VIRTUAL EXECUTION ENVIRONMENTS

Jan Vitek

PURDUE UNIVERSITY

with material from Nigel Horspool and Jim Smith

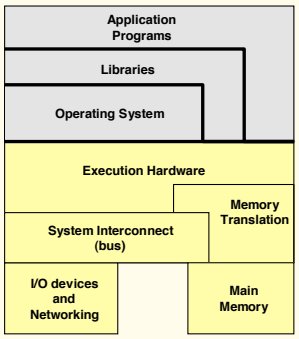
Virtualization

The "Machine"

- Different perspectives on what the *Machine* is:
- OS developer

Instruction Set Architecture

- ISA
- Major division between hardware and software



VEE '05 (c) 2005, J. E. Smith 8

- slides from Jim Smith's talk at VEE'05

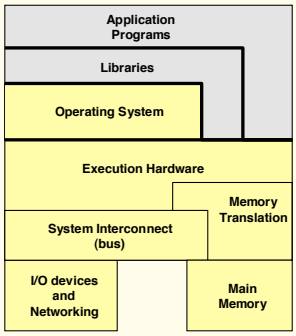
Virtualization

The "Machine"

- Different perspectives on what the *Machine* is:
- Compiler developer

Application Binary Interface

- ABI
- User ISA + OS calls



VEE '05 (c) 2005, J. E. Smith 9

- slides from Jim Smith's talk at VEE'05

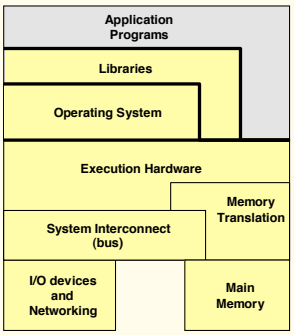
Virtualization

The "Machine"

- Different perspectives on what the *Machine* is:
- Application programmer

Application Program Interface

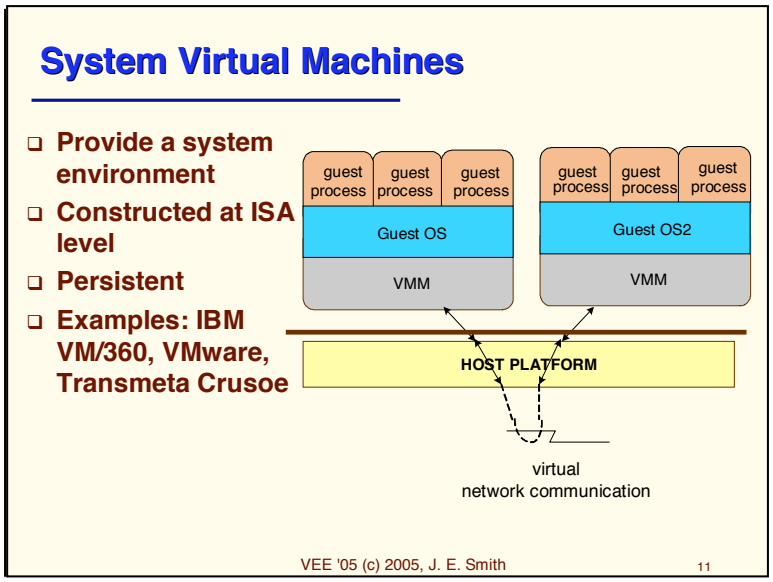
- API
- User ISA + library calls



VEE '05 (c) 2005, J. E. Smith 10

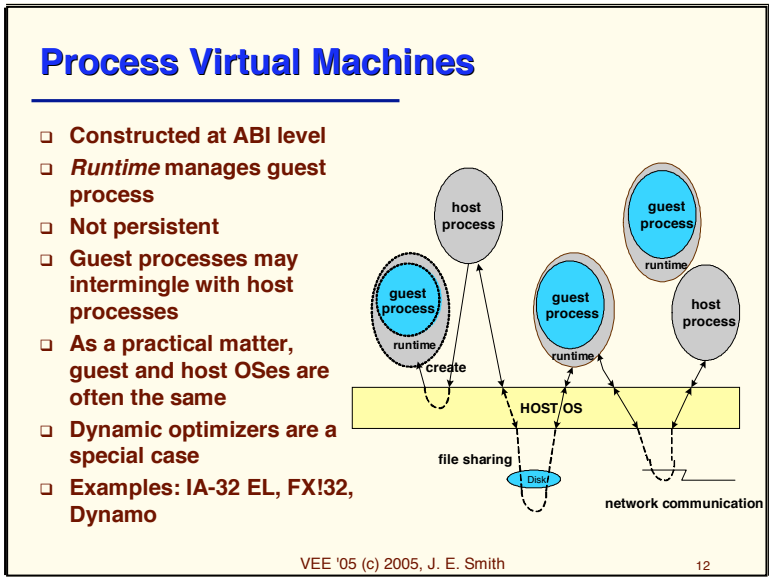
- slides from Jim Smith's talk at VEE'05

Virtualization



- slides from Jim Smith's talk at VEE'05

Virtualization

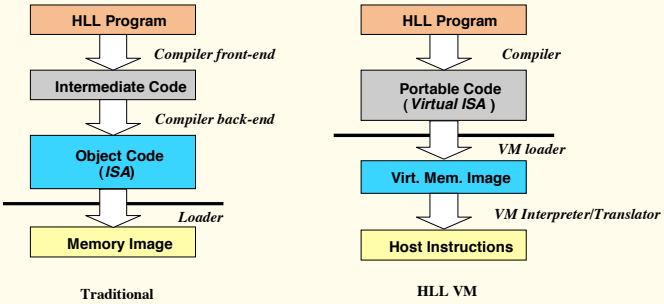


- slides from Jim Smith's talk at VEE'05

Virtualization

High Level Language Virtual Machines

- Raise the level of abstraction
  - User higher level virtual ISA
  - OS abstracted as standard libraries
- Process VM (or API VM)



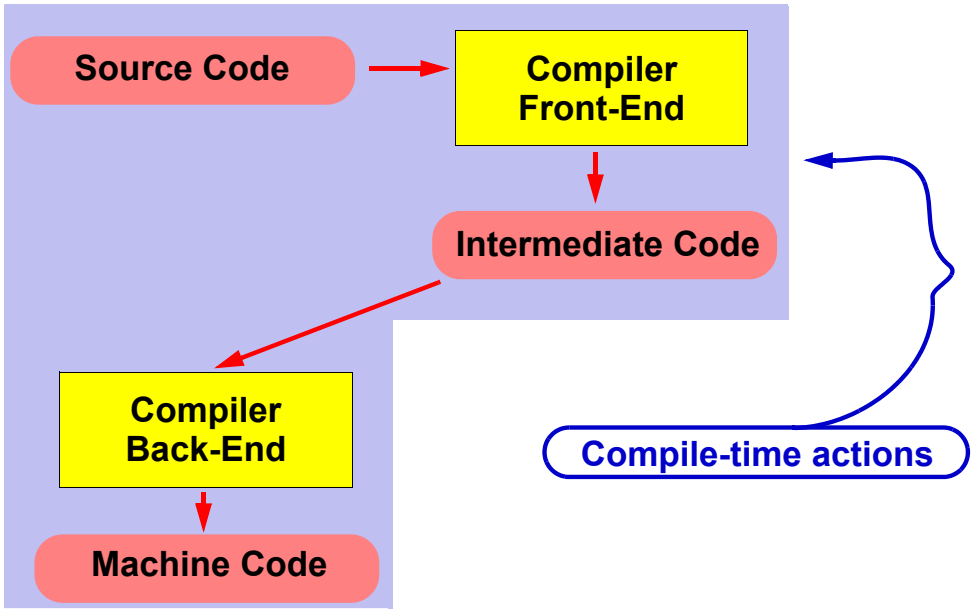
VEE '05 (c) 2005, J. E. Smith 13

- slides from Jim Smith's talk at VEE'05

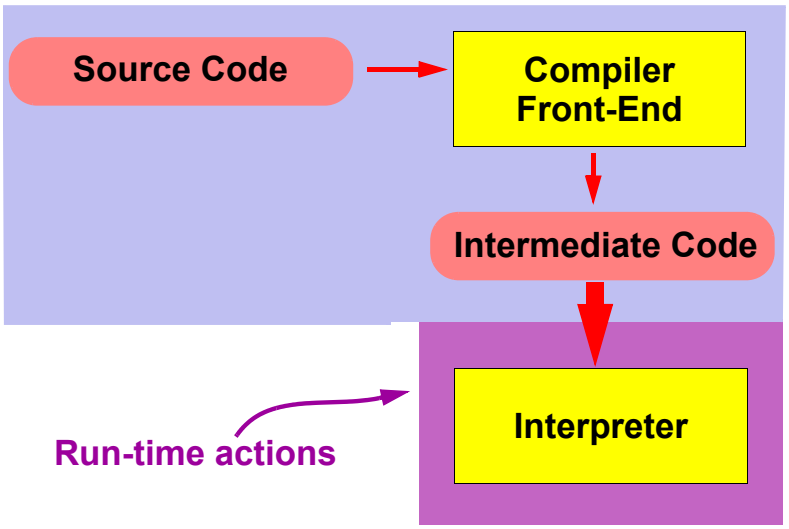
Implementation of Virtual Machines

Introduction

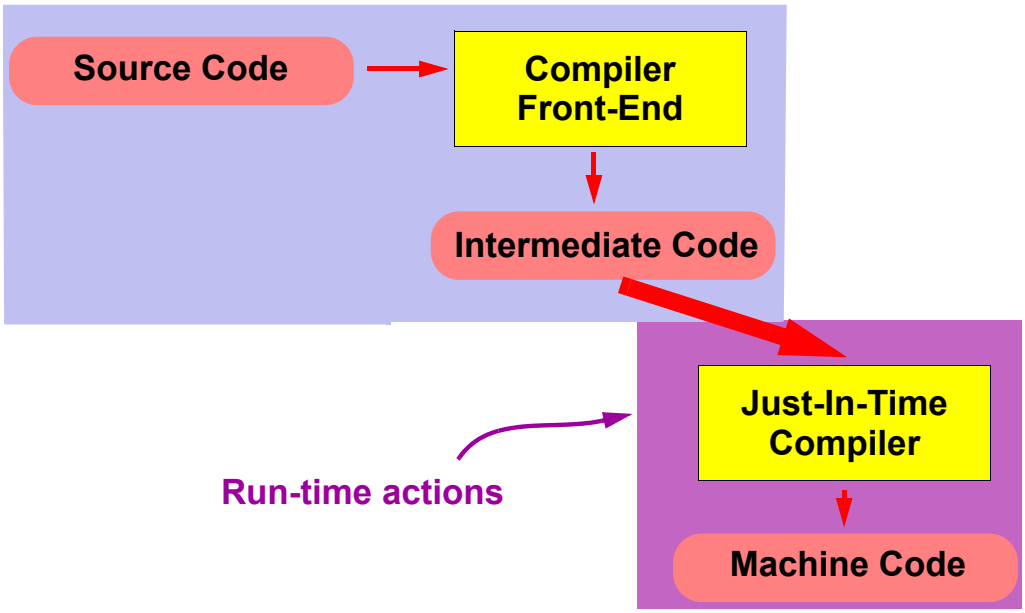
Usual Programming Language Implementation



Another Programming Language Implementation



And Another Implementation



An Overview

- Source code is translated into an intermediate representation, (IR)
- The IR can be processed in these different ways:
  1. compile-time (static) translation to machine code
  2. emulation of the IR using an interpreter
  3. run-time (dynamic) translation to machine code = JIT (Just-In-Time) compiling

What is IR?

IR is code for an idealized computer, a *virtual machine*.

Examples:

Language	IR	Implementation(s)
Java	JVM bytecode	Interpreter, JIT
C#	MSIL	JIT (but may be pre-compiled)
Prolog	WAM code	compiled, interpreted
Forth	bytecode	interpreted
Smalltalk	bytecode	interpreted
Pascal	p-code --	interpreted compiled
C, C++	--	compiled (usually)
Perl 6	PVM Parrot	interpreted interpreted, JIT
Python	--	interpreted
sh, bash, csh	original text	interpreted

Toy Bytecode File Format

We need a representation scheme for the bytecode. A simple one is:

- to use one byte for an opcode,
- four bytes for the operand of LDI,
- two bytes for the operands of LD, ST, JMP and JMPF.

As well as 0 for STOP, we will use this opcode numbering:

LDI	LD	ST	ADD	SUB	EQ	NE	GT	JMP	JMPF	READ	WRITE
1	2	3	4	5	6	7	8	9	10	11	12

The order of the bytes in the integer operands is important. We will use *big-endian* order.

The Classic Interpreter Approach

It emulates the fetch/decode/execute stages of a computer.

```
for( ; ; ) {
    opcode = code[pc++];
    switch(opcode) {
      case LDI:
          val = fetch4(pc);  pc += 4;
          push(val);
          break;
      case LD:
          num = fetch2(pc);  pc += 2;
          push( variable[num] );
          break;
      ...
      case SUB:
          right = pop();  left = pop();
          push( right-left );
      ...
```


The Classic Interpreter Approach, cont'd

```
      case JMP:
          pc = fetch2(pc);
          break;
      case JMPF:
          val = pop();
          if (val)
              pc += 2;
          else
              pc = fetch2(pc);
          break;
      ...
    }  /* end of switch */
}  /* end of for loop */
```

Critique

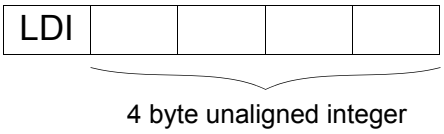
- The classic interpreter is easy to implement.
- It is flexible – it can be extended to support tracing, profiling, checking for uninitialized variables, debugging, ... anything.
- The size of the interpreter plus the bytecode is normally much less than the equivalent compiled program.
- But interpretive execution is *slow* when compared to a compiled program.

  The slowdown is 1 to 3 orders of magnitude (depending on the language).

What can we do to speed up our interpreter?

Improving the Classic Interpreter

1. Verification – verify that all opcodes and all operands are valid before beginning execution, thus avoiding run-time checks.
   We should also be able to verify that stacks cannot overflow or underflow.
2. Avoid unaligned data.



3. We can eliminate one memory access per IR instruction by expanding opcode numbers to addresses of the opcode implementations ...

Classic Interpreter with Operation Addresses

The bytecode file ... as in our example

READ; ST 0; READ; ST 1; LD 0; LD 1; NE; JMPF 54; LD 0; LD 1; GT; JMPF 41;
LD 0; LD 1; SUB; ST 0; JMP 51; LD 1; LD 0; SUB; ST 1; JMP 8; LD 0; WRITE; STOP

would be expanded into the following values when loaded into the interpreter's bytecode array.

&READ	&ST	0	&READ	&ST	1	&LD	...

and so on.

Each value is a 4 byte address or a 4-byte operand.

Classic Interpreter, cont'd

Now the interpreter dispatch loop becomes:

```
      pc = 0;   /* index of first instruction */
DISPATCH:
      goto *code[pc++];

LDI:
      val = *code[pc++];
      push(val);
      goto DISPATCH;
LD:
      num = *code[pc++];
      push( variable[num] );
      goto DISPATCH;
      ...
```

The C code can be a bit better still ...

Classic Interpreter, cont'd

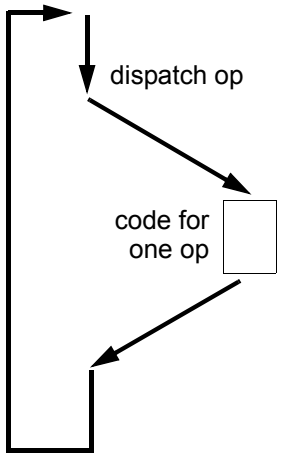
Recommended C style for accessing arrays is to use a pointer to the array elements, so we get:

```
        pc = &code[0];   /* pointer to first instruction */
DISPATCH:
        goto *pc++;
LDI:
        val = *pc++;
        push(val);
        goto DISPATCH;
LD:
        num = *pc++;
        push( variable[num] );
        goto DISPATCH;
        ...
```

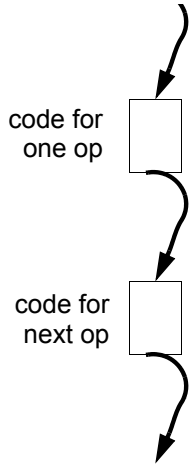
But let's step back and see a new technique –

(Direct) Threaded Code Interpreters

Reference: James R. Bell, Communications of ACM 1973



Classic Interpreter



Threaded Code Interpreter

Threaded Code Interpreters, cont'd

As before the bytecode is a sequence of addresses (intermixed with operands needed by the ops) ...

&LDI	99	&LDI	23	&ADD	&ST	5	...

The interpreter code looks like this ...

```
    /* start it going */
    pc = &code[0];
    goto *code[pc++];

LDI:
    operand = (int)*pc++;
    push(operand);
    goto *code[pc++];
```

```
ADD:
    right = pop();
    left = pop();
    push(left+right);
    goto *code[pc++];

    ...
```


Threaded Code Interpreters, cont'd

As before, better C style is to use a *pointer* to the next element in the code ...

```
    /* start it going */
    pc = &code[0];
    goto *(*pc++);

LDI:
    operand = (int)(*pc++);
    push(operand);
    goto *(*pc++);
```

```
ADD:
    right = pop();
    left = pop();
    push(left+right);
    goto *(*pc++);

    ...
```

This makes the implementation very similar to Bell's, who programmed for the DEC PDP11.

Further Improvements to Interpreters ...

A problem still being researched. (See the papers in the IVME annual workshop.)

Speed improvement ideas include:

1. Superoperators (see Proebsting, POPL 1995)
2. Stack caching (see Ertl, PLDI 1995)
3. Inlining (see Piumarta & Riccardi, PLDI 1998)
4. Branch prediction (see Ertl & Gregg, PLDI 2003)

Space improvement ideas (for embedded systems?) include:

1. Huffman compressed code (see Latendresse & Feeley, IVME 2003)
2. Superoperators – if used carefully (ibid)