

Virtual Machines, Interpretation Techniques, and Just-In-Time Compilers

Kostis Sagonas

kostis@csd.uu.se

Virtual Machines

Virtual machines (VMs) provide an intermediate stage for the compilation of programming languages.

- VMs are **machines** because they permit a step-by-step execution of programs.
- VMs are **virtual** (**abstract**) because typically they
 - are not implemented in hardware
 - omit many details of real (hardware) machines
- VMs are tailored to the particular operations required to implement a particular (class of) source language(s)

Virtual Machines: Pros

- Bridge the gap between the high level of a programming language and the low level of a real machine.
- Require less implementation effort
- Easier to experiment and modify (crucial for new PLs)
- Portability is enhanced
 - VM interpreters are typically implemented in C
 - VM code can be transferred over the net and run in most machines
 - VM code is (often significantly) smaller than object code
- Easier to be formally proven correct
- Various safety features of VM code can be verified

Virtual Machines: Cons

- Inferior performance of VM interpreters compared with a native code compiler for the same language
 - Cost of interpretation
 - Significantly more difficult to take advantage of modern hardware features (e.g. hardware-based branch prediction)

Implementation of Interpreters

There are various ways to implement interpreters:

1. Direct string interpretation

Source level interpreters are very slow because they spend much of their time in doing lexical analysis

2. Compilation into a (typically abstract syntax) tree and interpretation of that tree

Such interpreters avoid lexical analysis costs, but they still have to do much list scanning (e.g. when implementing a 'goto' or 'call')

3. Compilation into a virtual machine and interpretation of the VM code

Virtual Machine Interpreters

- By compiling the program to the instruction set of a virtual machine and adding a table that maps names and labels to addresses in this program, some of the interpretation overhead can be reduced
- For convenience, most VM instruction sets use integral numbers of bytes to represent everything
 - opcodes, register numbers, stack slot numbers, indices into the function or constant table, etc.

Opcode	Reg #	CONSTANT
--------	-------	----------

Example: The **GET_CONST2** instruction

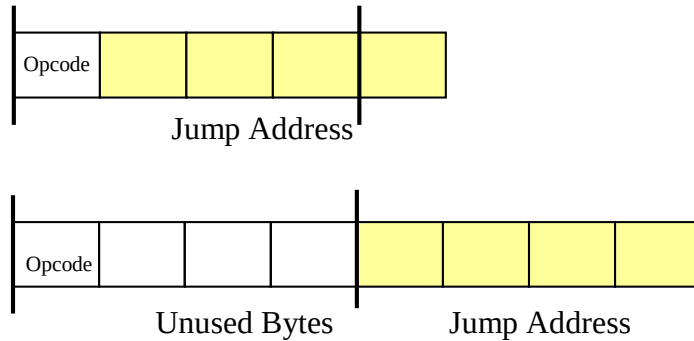
Components of Virtual Machine Implementations

- Program store (code area)
 - Program is a sequence of instructions
 - Loader
- State (of execution)
 - Stack
 - Heap
 - Registers
 - Special register (**program counter**) pointing to the next instruction to be executed
- Runtime system component
 - Memory allocator
 - Garbage collector

Basic Structure of a Bytecode Interpreter

```
byte *pc = &byte_program[0];
while(TRUE) {
    opcode = pc[0];
    switch (opcode) {
        ...
        case GET_CONST2:
            source_reg_num = pc[1];
            const_num_to_match = get_2_bytes(&pc[2]);
            ...
            pc += 4;
            break;
        ...
        case JUMP:
            jump_addr = get_4_bytes(&pc[1]);
            pc = &byte_program[jump_addr];
            break;
        ...
    }
}
```

To align or to not align VM instructions?



NOTE: Padding of instructions is done by the loader;
The size of the bytecode files need not be affected.

Bytecode Interpreter with Aligned Instructions

```
byte *pc = &byte_program[0];
while(TRUE) {
    opcode = pc[0];
    switch (opcode) {
        ...
        case GET_CONST2:
            source_reg_num = pc[1];
            const_num_to_match = get_2_bytes(&pc[2]);
            ...
            pc += 4;
            break;
        ...
        case JUMP: // aligned version
            jump_addr = get_4_bytes(&pc[4]);
            pc = &byte_program[jump_addr];
            break;
        ...
    }
}
```

Interpreter with Abstracted Instruction Encoding

```
byte *pc = &byte_program[0];
while(TRUE) {
    opcode = pc[0];
    switch (opcode) {
        ...
        case GET_CONST2:
            source_reg_num = pc[GET_CONST2_ARG1];
            const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
            ...
            pc += GET_CONST2_SIZEOF;
            break;
        ...
        case JUMP: // aligned version
            jump_addr = get_4_bytes(&pc[JUMP_ARG1]);
            pc = &byte_program[jump_addr];
            break;
        ...
    }
}
```

```
#define GET_CONST2_SIZEOF 4
#define JUMP_SIZEOF 8
#define GET_CONST2_ARG1 1
#define GET_CONST2_ARG2 2
#define JUMP_ARG1 4
```

Interpreter with Abstracted Control

```
byte *pc = &byte_program[0];
while(TRUE) {
    next_instruction:
    opcode = pc[0];
    switch (opcode) {
        ...
        case GET_CONST2:
            source_reg_num = pc[GET_CONST2_ARG1];
            const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
            ...
            pc += GET_CONST2_SIZEOF;
            NEXT_INSTRUCTION;
        ...
        case JUMP: // aligned version
            jump_addr = get_4_bytes(&pc[JUMP_ARG1]);
            pc = &byte_program[jump_addr];
            NEXT_INSTRUCTION;
        ...
    }
}
```

```
#define NEXT_INSTRUCTION \
    goto next_instruction
```

Structure of Indirectly Threaded Interpreter

```
byte *pc = &byte_program[0];
while(TRUE) {
next_instruction:
opcode = pc[0];
switch (opcode) {
...
case GET_CONST2:
get_const2_label:
source_reg_num = pc[GET_CONST2_ARG1];
const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
...
pc += GET_CONST2_SIZEOF;
NEXT_INSTRUCTION;
...
case JUMP: // aligned version
jump_label:
jump_addr = get_4_bytes(&pc[JUMP_ARG1]);
pc = &byte_program[jump_addr];
NEXT_INSTRUCTION;
...
}
}
```

```
static void *label_tab[] {
...
&&get_const2_label;
...
&&jump_label;
...
}
#define NEXT_INSTRUCTION \
goto **(void **)(label_tab[*pc])
```

Structure of Directly Threaded Interpreter

```
byte *pc = &byte_program[0];
while(TRUE) {
next_instruction:
opcode = pc[0];
switch (opcode) {
...
case GET_CONST2:
get_const2_label:
source_reg_num = pc[GET_CONST2_ARG1];
const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
...
pc += GET_CONST2_SIZEOF;
NEXT_INSTRUCTION;
...
case JUMP: // aligned version
jump_label:
pc = get_4_bytes(&pc[JUMP_ARG1]);
NEXT_INSTRUCTION;
...
}
}
```

```
static void *label_tab[] {
...
&&get_const2_label;
...
&&jump_label;
...
}
#define NEXT_INSTRUCTION \
goto **(void **)(pc)
```

Instruction Merging and Specialization

Instruction Merging: A sequence of VM instructions is replaced by a single (mega-)instruction

- Reduces interpretation overhead
- Code locality is enhanced
- Results in more compact bytecode
- C compiler has bigger basic blocks to perform optimizations on

Instruction Specialization: A special case of a VM instruction is created, typically one where some arguments have a known value which is hard-coded

- Eliminates the cost of argument decoding
- Results in more compact bytecode representation
- Reduces the register pressure from some basic blocks