Virtual Machines and Interpretation Techniques

Kostis Sagonas
kostis@it.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
- Profiling and debugging are easier to implement

Virtual Machines: Cons

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

Some History of VM Development

- VMs have been built and studied since the late 1950's
- The first Lisp implementations (1958) used VMs with garbage collection, sandboxing, reflection, and an interactive shell
- Forth (early 70's) used a very small and easy to implement VM with high level of reflection
- Smalltalk (late 70's) allowed changing code on the fly (first truly interactive OO system)
- USCD Pascal (late 70's) popularized the idea of using pseudocode to improve portability
- Self (late 80's), a language with a Smalltalk flavor, had an implementation that pushed the limits of VM performance
- Java (early 90s) made VMs popular and well known

VM Design Choices

- Some design choices are similar to the choices when designing the intermediate code format of a compiler:
  - Should the machine be used on several different physical architectures and operating systems? (JVM)
  - Should the machine be used for several different source languages? (CLI/CLR (.NET))
- Some other design choices are similar to those of the compiler backend:
  - Is performance more important than portability?
  - Is reliability more important than performance?
  - Is (smaller) code size more important than performance?
- And some design choices are similar to those in an OS:
  - How to implement memory management, concurrency, exceptions, I/O, …
  - Is low memory consumption, scalability, or security more important than performance?

VM Components

- The components of a VM vary depending on several factors:
  - Is the language (environment) interactive?
  - Does the language support reflection and/or dynamic loading?
  - Is performance paramount?
  - Is concurrency support required?
  - Is sandboxing required?

(In this lecture we will only talk about the interpreter of the VM.)

VM Implementation

- Virtual machines are usually written in “portable” programming languages such as C or C++.
- For performance critical components, assembly language is often used.
- VMs for some languages (Lisp, Forth, Smalltalk) are largely written in the language itself.
- Many VMs are written specifically for GNU C, for reasons that will become apparent in later slides.

Forms of Interpreters

- Programming language implementations often use two distinct kinds of interpreters:
  - Command-line interpreter
    - Reads and parses language constructs in source form
    - Used in interactive systems
  - Virtual machine instruction interpreter
    - Reads and executes instructions in some intermediate form such as VM bytecode

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 Instruction 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

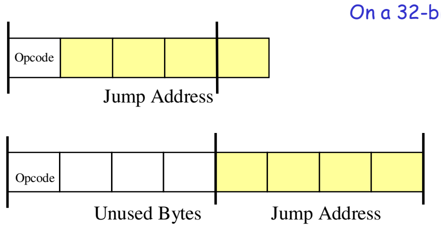
- Program store (code area)
  - Program is a sequence of instructions
  - Loader
- State (of execution)
  - Stack
  - Heap
  - Registers
    - Special register (program counter) pointing to the next instruction to be executed
- Runtime system component
  - Memory allocator
  - Garbage collector

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]);
       … // get_const2 code
       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?

On a 32-bit machine



NOTE: Padding of instructions can be 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]);
       … // get_const2 code
       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

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

```
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]);
       … // get_const2 code
       pc += GET_CONST2_SIZEOF;
       break;
     …
     case JUMP: // aligned version
       jump_addr = get_4_bytes(&pc[JUMP_ARG1]);
       pc = &byte_program[jump_addr];
       break;
     …
   }
}
```

Interpreter with Abstracted Control

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

```
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]);
       … // get_const2 code
       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;
     …
   }
}
```

Indirectly Threaded Interpreters

- In an indirectly threaded interpreter we do not switch on the opcode encoding. Instead we use the bytecodes as indices into a table containing the addresses of the VM instruction implementations
- The term *threaded code* refers to a code representation where every instruction is implicitly a function call to the next instruction
- A threaded interpreter can be very efficiently implemented in assembly
- In GNU CC, we can use the labels as values C language extension and take the address of a label with &&labelname
- We can actually write the interpreter in such a way that it uses indirectly threaded code if compiled with GNU CC and a switch for compatibility

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]);
        … // get_const2 code
        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])
```

Directly Threaded Interpreter

- In a directly threaded interpreter, we do not use the bytecode instruction encoding at all during runtime
- Instead, the loader replaces each bytecode instruction encoding (opcode) with the address of the implementation of the instruction
- This means that we need one word for the opcode, which slightly increases the VM code size

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]);
        … // get_const2 code
        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)
```

```
#define GET_CONST2_SIZEOF 8
#define JUMP_SIZEOF 8
#define GET_CONST2_ARG1 5
#define GET_CONST2_ARG2 6
#define JUMP_ARG1 4
```

Threaded Interpreter with Prefetching

```
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; // prefetching
        … // get_const2 code
        NEXT_INSTRUCTION;
     …
     case JUMP: // aligned version
     jump_label:
        pc = get_4_bytes(&pc[JUMP_ARG1]);
        NEXT_INSTRUCTION;
     …
   }
}
```

Subroutine Threaded Interpreter

- The only portable way to implement a threaded interpreter in C is to use subroutine threaded code
- Each VM instruction is implemented as a function and at the end of each instruction the next function is called

Stack-based vs. Register-based VMs

- A VM can either be *stack-based* or *register-based*
  - In a stack-based machine most operands are (passed) on the stack. The stack can grow as needed.
  - In a register-based machine most operands are passed in (virtual) registers. The number of registers is limited.
- Most VMs are stack-based
  - Stack machines are simpler to implement
  - Stack machines are easier to compile to
  - Less encoding/decoding to find the right register
  - Virtual registers are no faster than stack slots

Virtual Machine Interpreter Tuning

Common VM interpreter optimizations include:

- Writing the interpreter loop and key instructions in assembly
- Keeping important VM registers (pc, stack top, heap top) in hardware registers
    - GNU C allows global register variables
- Top of stack caching
- Splitting the most used set of instruction into a separate interpreter loop

---

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