Virtual Machines & Interpretation Techniques

Advanced Compiler Techniques 2005
Erik Stenman

Virtual Machines

- A virtual machine is an abstract computing architecture independent of any hardware.
- They are software machines that run on top of real hardware, providing an abstraction layer for language implementers.
  - There are other types of virtual machines intended to emulate or simulate some real hardware (e.g., Virtutech-Simics, VMware, Transmeta), but they are not the focus of this course.

Characteristics of a VM

- A VM has its own instruction set independent of the host system.
- A VM usually has its own memory manager and can also provide its own concurrency primitives.
- Access to the host OS is usually limited and controlled by the VM.

Advantages of VMs

- A VM bridges the gap between the high level language and the low level aspects of a real machine.
- It is relatively easy to implement a VM, and it is easier to compile to a VM than to a real machine.
- A VM can be modified when experimenting with new languages.
- Portability is enhanced.
- Support for dynamic (down-)loading of software.
- VM code is usually smaller than real machine code.
- Safety features can be verified by the VM.
- Profiling and debugging are easy to implement.

Disadvantages of VMs

- Lower performance than with a native code compiler.
  - Overhead of interpretation.
  - Modern hardware is not designed for running interpreters.

Some VM History

- VMs have been built and studied since the late 1950s.
- The first Lisp implementations (1958) used virtual machines with garbage collection, sandboxing, reflection, and an interactive shell.
- Forth (early 70s) uses a very small and easy to implement VM with high level of reflection.
- Smalltalk (early 70s) is a very dynamic language where everything can be changed on the fly, the first truly interactive OO system.
- USCD Pascal (late 70s) popularized the idea of using pseudocode to improve portability.
- Self (late 80s) a prototype-based Smalltalk flavor with an implementation that pushed the limits of VM technology.
- Java (early 90s) made VMs popular and well known.

VM Design Choices

- When designing a VM one has some design choices similar to the choices when designing intermediate code for 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 design choices are similar to those of the compiler backend:
  - Is performance more important than portability?
  - Is reliability more important than performance?
  - Is (smaller) size more important than performance?
- And some design choices are similar to when designing an OS:
  - How to implement memory management, concurrency, IO…
  - 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" (in the sense that compilers for most architectures already exists) programming languages such as C or C++.
- For performance critical components assembly language can be used.
- Some VMs (Lisp, Forth, Smalltalk) are largely written in the language itself.
- Many VMs are written specifically for gcc, for reasons that will become clear in later slides.

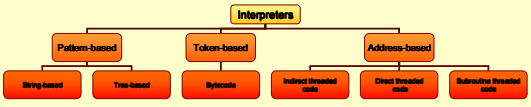
Interpreters

- Language runtime systems often uses two kinds of interpreters:
  1. Command-line interpreter.
     - Reads and parses instructions in source form.
     - Used in interactive systems.
  2. Instruction interpreter.
     - Reads and executes instructions in some intermediate form such as bytecode.

Implementing Interpreters

- There are several ways to implement an interpreter.
  - Pattern (or string) based interpretation.
    - Interpreting source code (strings) directly is inefficient since most of the time is spent in lexical analysis.
    - A better alternative is to compile the source into e.g., an abstract syntax tree and then do the interpretation over that tree. (Jumps and calls are expensive.)
  - Token-based interpretation.
    - Compiling the code into a linear representation of instructions, where each instruction is represented by a token, e.g., bytecode.
  - Address-based interpretation.
    - Compiling the code into a linear representation where each instruction is represented by the address that implements the instruction.
    - There are several variants: Indirect threaded code, direct threaded code and subroutine threading.

Taxonomy of Interpreters



Implementing Interpreters

- We will now look at some details of how to implement an interpreter.
- We will start with a complete but simple string based interpreter for a very simple language. Then extend the language and the interpreter to show the different ways to implement interpreters.

Interpreting while Parsing (String-based Interpretation)

- For some really simple languages the interpretation can be done during parsing.
- We can e.g., implement a simple calculator directly in a parser generator.
- A parser generator is a program that takes a description of a grammar and generates a program that can parse the grammar.
- We will use CUP a parser generator for Java:
  - http://www.cs.princeton.edu/~appel/modern/java/CUP/
  - I will not go into the details of CUP.

A Calculator Language

- Grammar:

```
Expr ::= Expr MINUS Term
       | Expr PLUS Term
       | Term
Term ::= Term TIMES Factor
       | Term DIV Factor
       | Factor
Factor ::= NUMBER | LPAR Expr RPAR
```


Simple Interpreter .cup

```
terminal PLUS, MINUS, TIMES, DIV, LPAR, RPAR;
terminal Integer NUMBER;

non terminal Program;
non terminal Integer Expression, Term, Factor;

precedence left PLUS, MINUS;
precedence left TIMES, DIV;

start with Program;
```


Interpreter .cup

```
Program ::= Expression:e
        {: System.out.println(e.intValue()); :}
        ;
Expression ::= Expression:e PLUS Term:t
        {: RESULT = new Integer(e.intValue() +
                                t.intValue()); :}
        |  Expression:e MINUS Term:t
        {: RESULT = new Integer(e.intValue() -
                                t.intValue()); :}
        |  Term:t
        {: RESULT = t; :}
```


Interpreter .cup

```
Term ::= Term:t TIMES Factor:f
        {: RESULT = new Integer(t.intValue() *
                                f.intValue()); :}
        |  Term:t DIV Factor:f
        {: RESULT = new Integer(t.intValue() /
                                f.intValue()); :}
        |  Factor:f
        {: RESULT = f; :}
Factor ::= NUMLIT:n {: RESULT = n; :}
        |  LPAR Expression:e RPAR
        {: RESULT = e; :}
```

Control Flow

- This approach works fine for simple expressions.
- Control flow constructs such as 'if' and 'while' are harder to handle.
- For 'while' we would need to "reparse" the statement that is to be repeated.
- Let us extend the language with control flow, variables, and boolean values.

Tree-based (pattern-based) Interpretation

- By representing the code by a data structure we can "reexecute" the same piece of code several times.
- This will lead to a slightly more complicated interpreter, which will require at least two passes over the code.
- The code will first be parsed and stored in the internal representation, then the interpretation will be performed.
- We can use an abstract syntax tree for representing the code.

Design choices

- How is the program represented?
  - As an Abstract Syntax Tree (AST) with the class `Tree`.
- How is data represented?
  - We have different types of values, integers and Booleans.
  - The value of each expression is either an `IntValue` or a `BoolValue`, subclasses of `Value`.
- How are variables represented?
  - With a symbol table where each symbol can have a value.

The Implementation

- The Interpreter itself can be implemented by a Visitor on the AST.
- We need a Value class:

```
class Value {
  static class IntValue extends Value {
    int i;
    public IntValue(int i) { this.i = i; }
  }
  static class BoolValue extends Value {
    boolean b;
    public BoolValue(boolean b) { this.b = b; }
  }
}
```


Interpreting Expressions

```
public void caseOp(Op tree) {
  switch (tree.op) {
  case TRUE:
    result = new BoolVal(true);
    break;
  case FALSE:
    result = new BoolVal(false);
    break;
  case PLUS:
    IntValue lval = (IntValue) interpret(tree.left);
    IntValue rval = (IntValue) interpret(tree.right);
    result = new IntValue(lval.i + rval.i);
    break;
  …
```


Semantic Analysis Needed

- This assumes that types are correct.
  - We could either have a prepass that does the type analysis.
  - Or we could do the type checking at the same time as interpreting.

Analyzing While Interpreting

```
public void caseOp(Op tree) {
  switch (tree.op) {
  case PLUS:
    Value lval = interpret(tree.left);
    Value rval = interpret(tree.right);
    if ((lval instanceof IntValue) &&
       (rval instance of IntValue)) {
       result = new IntValue(
            ((IntValue)lval).i +
          ((IntValue)rval).i);
    } else error();
    break;
...
```


Control Flow

- Now we can try to interpret a control flow construct.
- It turns out to be very easy, since we are writing our interpreter in Java which supports the same control flow constructs.
- It becomes a bit complicated if the type analysis has to be done at the same time.

While (assuming type analysis)

```
public void caseWhile(While tree){
  while(((BoolValue)
   interpret(tree.cond)).b) {
     interpret(tree.body);
  }
}
```


Interpreting While, While Analyzing

```
public void caseWhile(While tree) {
  Value cond=interpret(tree.cond);
  while((cond instanceof BoolValue)
        && ((BoolValue) cond).b) {
    interpret(tree.body);
    cond=interpret(tree.cond);
  }
}
```


Variables

- We need to keep track of the values of variables somehow. A simple solution is to store these values with the symbols in the symbol table.
- If we interpret an assignment we store the value in the symbol.
- If we interpret an identifier we read the value from the symbol.

Functions

- These techniques can handle simple languages without functions or more than one scope.
- In order to handle functions and especially recursive functions and local scopes we will need an *environment*.

Environments

- In an *environment* we store all values of parameters (arguments) and local variables of a function for one specific call.
- We create a new environment when we call a function or enter a local scope.
  - We store actual arguments of the call in the environment.
  - We initialize local variables.
  - After returning from a function, or leaving the local scope, the environment is not needed any more.
- The environment can be implemented as an array of values, the position in the array of an identifier can be stored in the symbol table.

```
class Environment {
  Environment outer; // For nested scope.
  Value[] values;
}
```

- An environment is similar to how scopes are handled in the compiler.
- When compiling to native code the environment is stored on the stack as activation records.

Function Calls

```
void caseFunCall {
  // call interpreter recursively on
  //   function arguments;
  Arguments args = interpret_args(tree.args);

  // Create a new Environment
  currentEnv = new Environment(currentEnv);

  // Store the arguments in the new environment.
  insert_args(args, currentEnv);

  // Call the interpreter recursively on the
  //   body of the called function, using the new
  //   environment.
  result = interpret(find_code(tree.funName));

  // Restore the environment.
  currentEnv = currentEnv.outer;
}
```


Disadvantages with Tree-based Interpreters

- The tree representation has to be created somehow each time we want to run the program.
  - Parsing the source code each time is time consuming.
  - Storing the whole tree is space consuming.
- The tree representation uses a lot of space at runtime, which is infeasible for large programs.
- Using the stack of the host language adds to the space need at runtime.

Token-based Interpreters

- By compiling the program to a special instruction set of a virtual machine, and by adding tables that maps function names to offsets in the instruction sequence, some of the interpretation overhead can be reduced.
- Most VM instruction sets uses small integers to represent everything in the instruction stream (opcodes, registers, stack slots, functions, constants, etc.).
- By implementing the interpreter in C we can gain some speed, it also allows us to do nasty pointer tricks.

Token-based Interpreters

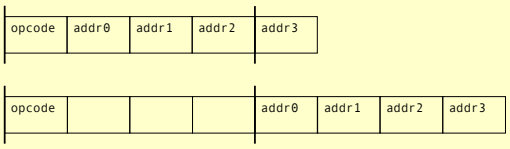
- The fundamental instruction unit is the *token*.
- A token is a predefined numeric value that represents a certain instruction.
  - E.g., BREAK=0, LOADLITERAL = 1, ADD=2.
- The most common case is *bytecode*:
  - The token width is 8 bits.
  - The total instruction set is limited to 256 tokens.

Basic Structure of a Token-based Interpreter

```
byte *pc = &program[0];
while(TRUE) {
  byte opcode = pc[0];
  switch(opcode) {
  …
    case LOADLITERAL:
      destReg = pc[1];
      value = getTwoBytes(&pc[2]);
      regs[destReg] = value;
      pc += 4;
      break;
    case JUMP:
      jumpAddress = getFourBytes(&pc[1]);
      pc = &program[jumpAddress]
      break;
  }
}
```

Alignment

- Most modern machines loads data at least one word at the time (usually 4 bytes). By making sure that instructions are aligned on word offsets we get better performance.



Note: The padding is done by the loader, no extra space is needed in the external representation.

Token-based Interpreter with Aligned Instructions

```
byte *pc = &program[0];
while(TRUE) {
  byte opcode = pc[0];
  switch(opcode) {
  …
    case LOADLITERAL:
      destReg = pc[1];
      value = getTwoBytes(&pc[2]);
      regs[destReg] = value;
      pc += 4;
      break;

    case JUMP:
      jumpAddress = getFourBytes(&pc[4]);
      pc = &program[jumpAddress]
      break;
  }
}
```


Token-based Interpreter with Abstract Encoding

```
#define LOADLITTERAL_SIZE 4
#define JUMP_SIZE         8
#define LOADLITTERAL_ARG1 1
#define LOADLITTERAL_ARG2 2
#define JUMP_ARG1         4
```

```
byte *pc = &program[0];
while(TRUE) {
  byte opcode = pc[0];
  switch(opcode) {
  …
    case LOADLITERAL:
      destReg = pc[LOADLITTERAL_ARG1];
      value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);
      regs[destReg] = value;
      pc += LOADLITTERAL_SIZE;
      break;

    case JUMP:
      jumpAddress = getFourBytes(&pc[JUMP_ARG1]);
      pc = &program[jumpAddress]
      break;
  }
}
```


Token-based Interpreter with Abstract Control

```
#define NEXT goto loop
```

```
byte *pc = &program[0];
while(TRUE) {
loop:
  byte opcode = pc[0];
  switch(opcode) {
  …
    case LOADLITERAL:
      destReg = pc[LOADLITTERAL_ARG1];
      value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);
      regs[destReg] = value;
      pc += LOADLITTERAL_SIZE;
      NEXT;

    case JUMP:
      jumpAddress = getFourBytes(&pc[JUMP_ARG1]);
      pc = &program[jumpAddress]
      NEXT;
  }
}
```


Indirectly Threaded Interpreter

- In an *indirectly threaded* interpreter we do not switch on the tokens. Instead we use the tokens as indices into a table containing the addresses of the 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 assembler.
- In GNU C (gcc) we can use *labels as values* 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 gcc, and uses a switch for compatibility when compiled with other compilers.

Indirectly Threaded Interpreter

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

```
byte *pc = &program[0];
while(TRUE) {
loop:
  byte opcode = pc[0];
  switch(opcode) {
  …
    case LOADLITERAL:
    loadlitteral_label:
      destReg = pc[LOADLITTERAL_ARG1];
      value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);
      regs[destReg] = value;
      pc += LOADLITTERAL_SIZE;
      NEXT;

    case JUMP:
    jump_label:
      jumpAddress = getFourBytes(&pc[JUMP_ARG1]);
      pc = &program[jumpAddress]
      NEXT;
  }
}
```

Directly Threaded Interpreter

- In a directly threaded interpreter we do not use tokens at all during runtime.
- Instead the loader replaces each token with the address of the implementation of the instruction.
- This means the opcodes will take one word or four bytes at runtime, slightly increasing the code size.

Directly Threaded Interpreter

```
byte *pc = &program[0];
while(TRUE) {
loop:
  byte opcode = pc[0];
  switch(opcode) {
  …
    case LOADLITERAL:
    loadlitteral_label:
      destReg = pc[LOADLITTERAL_ARG1];
      value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);
      regs[destReg] = value;
      pc += LOADLITTERAL_SIZE;
      NEXT;

    case JUMP:
    jump_label:
      jumpAddress = getFourBytes(&pc[JUMP_ARG1]);
      pc = &program[jumpAddress]
      NEXT;
  }
}
```

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


Subroutine Threaded Interpreter

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

Subroutine Threaded Interpreter (with tail-calls)

```
byte *pc = &program[0];
NEXT;

…
void loadlitteral(void) {
  destReg = pc[LOADLITTERAL_ARG1];
  value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);
  regs[destReg] = value;
  pc += LOADLITTERAL_SIZE;
  NEXT;
}
void jump(void) {
  jumpAddress = getFourBytes(&pc[JUMP_ARG1]);
  pc = &program[jumpAddress];
  NEXT;
}
```

```
static void *label_tab[] {
  ...
  &loadlitteral;
  &jump;
}
#define NEXT ((void (*)())*pc)()
```


Subroutine Threaded Interpreter

```
byte *pc = &program[0];
while (TRUE) NEXT;

…
void loadlitteral(void) {
  destReg = pc[LOADLITTERAL_ARG1];
  value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);
  regs[destReg] = value;
  pc += LOADLITTERAL_SIZE;
}
void jump(void) {
  jumpAddress = getFourBytes(&pc[JUMP_ARG1]);
  pc = &program[jumpAddress];
}
```

```
static void *label_tab[] {
  ...
  &loadlitteral;
  &jump;
}
#define NEXT ((void (*)())*pc)()
```


Subroutine Threaded Interpreter

```
(void (*)()) pc = &program[0];
while (TRUE) *pc++;

…
void loadlitteral(void) {
  destReg = ((int *)pc)[LOADLITTERAL_ARG1];
  value = getTwoBytes(&pc[LOADLITTERAL_ARG2]);
  regs[destReg] = value;
  pc += LOADLITTERAL_SIZE;
}
void jump(void) {
  jumpAddress = getFourBytes(&pc[JUMP_ARG1]);
  pc = &program[jumpAddress];
}
```

```
#define LOADLITTERAL_SIZE 1
#define JUMP_SIZE 1
#define LOADLITTERAL_ARG1 0
#define LOADLITTERAL_ARG2 1
#define JUMP_ARG1 0
```

Stack-based vs. Register-based VM

- A VM can either be *stack-based* or *register-based*.
  - In a stack-based machine most operands are on the stack. The stack can grow as needed.
  - In a register-based machine most operands are 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.

Interpreter Tuning

- Common interpreter optimizations include:
  - Writing the interpreter loop and key instructions in assembler.
  - Keeping important variables in hardware registers (pc, stack-top, heap-top). (GNU C allow global register variables.)
  - Top of stack caching.
  - Splitting the most used instruction into a separate interpreter loop.

Interpreter Tuning

- More advanced interpreter optimizations includes:
  - Instruction merging: A common sequence of VM instructions is replaced by a single instruction.
    - Reduced interpretation overhead.
    - Enhances code locality.
    - More compact bytecode.
    - Gives C compiler bigger code block to optimize.
  - Instruction specialization: A special case VM instruction is created, typically with some arguments hard-coded.
    - Eliminates argument decoding cost.
    - More compact bytecode.
    - Reduces register pressure.

Just-in-time Compilation

- Native code is still faster than code interpreted in VMs. To get the best performance native code compilation is necessary. But bytecode is a nice format to distribute portable code.
- Solution: *dynamic compilation* or *just-in-time* (JIT) compilation.
- Native code takes more space than virtual machine code (4-8x). Don't compile everything to native code (some code is never executed).
- Compilation takes time, dynamic compilation has to be fast. No time for advanced optimization (unless the bytecode compiler has inserted hints in the bytecode).

JIT – What to Compile

- Only compile a method if the total execution time is reduced.
- How do we know this?
- Use the past to predict the future:
  - Use profiling to detect what and when to compile. There are two basic approaches:
    - Invocation counters.
    - Sample based profiling.

Invocation Counters

- Associate a counter with each function.
- When a function is called increment the counter.
- If the counter reaches a limit compile the function. Reset or use decay to only compile high-frequency functions.
- Hard to predict behavior, no control over time spent in compiler.

Sample Based Profiling

- Measure time spent in interpreter, compiler, and in compiled code.
- Harder to implement.
- Gives better picture of the hot-spots.

JIT Integration

- Integrating a JIT system where native code can coexist with interpreted code in the VM is not trivial.
- Context switches between native and interpreted code has to be fast. (They can occur at function calls, returns, and when exceptions are thrown.)
- Ensuring proper tail-calls with a mixed execution environment is also tricky.

Summary

- Virtual machines provides an abstraction from real hardware and make programming language implementation easier and languages more portable.
- A direct threaded interpreter gives the best performance.
- Virtual machines have been used for half a century but research didn't really take off until the JVM came along.