Interpreters & virtual machines

Michel Schinz

Advanced Compiler Construction / 2006-03-24

Interpreters

An **interpreter** is a program which executes another program, represented as some kind of data-structure. Common program representations include:

- raw text (source code),
- trees (AST of the program),
- linear sequences of instructions.

Why interpreters?

Interpreters enable the execution of a program without requiring its compilation to native code.

They simplify the implementation of programming languages and – on modern hardware – are efficient enough for most tasks.

Text-based interpreters

Text-based interpreters directly interpret the textual source of the program.

They are very seldom used, except for trivial languages where every expression is evaluated at most once (*i.e.* no loops or functions).

Plausible example: a calculator program, which evaluates arithmetic expressions while parsing them.

Tree-based interpreters

Tree-based interpreters walk over the abstract syntax tree of the program to interpret it.

Their advantage compared to string-based interpreters is that parsing (and name/type analysis, if applicable) is done only once.

Plausible example: a graphing program, which has to repeatedly evaluate a function supplied by the user to plot it.

Virtual machines

Virtual machines behave in a similar fashion as real machines (*i.e.* CPUs), but are implemented in software. They accept as input a program composed of a sequence of instructions.

Virtual machines often provide more than the interpretation of programs: they manage memory, threads, and sometimes I/O.

Virtual machines history

Perhaps surprisingly, virtual machines are a very old concept, dating back to ~1950.

They have been (and still are) used in the implementation of many important languages, like SmallTalk, Lisp, Forth, Pascal, and more recently Java and C#.

Why *virtual* machines?

Since the compiler has to generate code for some machine, why prefer a virtual over a real one?

- for simplicity: a VM is usually more high-level than a real machine, which simplifies the task of the compiler,
- for portability: compiled VM code can be run on many actual machines,
- to ease execution monitoring.

Drawback of virtual machines

The only drawback of virtual machines compared to real machines is that the former are slower than the latter.

This is due to the overhead associated with interpretation (fetching and decoding instructions, etc.).

Moreover, the high number of indirect jumps in interpreters causes pipeline stalls in modern processors.

Kinds of virtual machines

There are two kinds of virtual machines:

- stack-based VMs, which use a stack to store intermediate results, variables, etc.
- register-based VMs, which use a limited set of registers for that purpose, like a real CPU.

There is some controversy as to which kind is better, but most VMs today are stack-based.

Virtual machines input

Virtual machines take as input a program expressed as a sequence of instructions.

Each instruction is identified by its **opcode** (**op**eration **code**), a simple number. Often, opcodes occupy one byte, hence the name **byte code**.

Some instructions have additional arguments, which appear after the opcode in the instruction stream.

Basic implementation techniques

overhead

Virtual machines are implemented in much the same way as a real processor:

- the next instruction to execute is fetched from memory and decoded,
- the operands are fetched, the result computed, and the state updated,
- the process is repeated.

VMs implementation languages

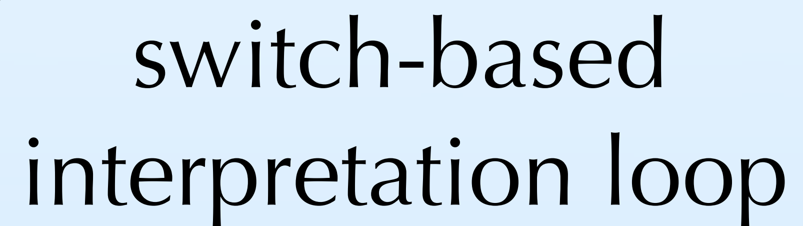
Many VM today are written in C(++), because these languages are at the right abstraction level for the task, and fast.

As we will see later, an extension of `gcc` enables one particularly interesting optimisation, which means that several VMs are written for `gcc`.

Implementing a VM in C

```
typedef enum {
  add, /* ... */
} Instruction;

void interpret() {
  static Instruction program[] = { add /* ... */ };
  Instruction* ip = program;
  int* sp;
  for (;;) {
    switch (*ip++) {
    case add:
      sp[1] = sp[0] + sp[1];
      sp++;
      break;
      /* ... */
    }
  }
}
```



Optimising VMs

The basic implementation of a virtual machine presented earlier can be made faster using several techniques. We will now look at the following:

- threaded code,
- top of stack caching,
- super-instructions,
- JIT compilation.

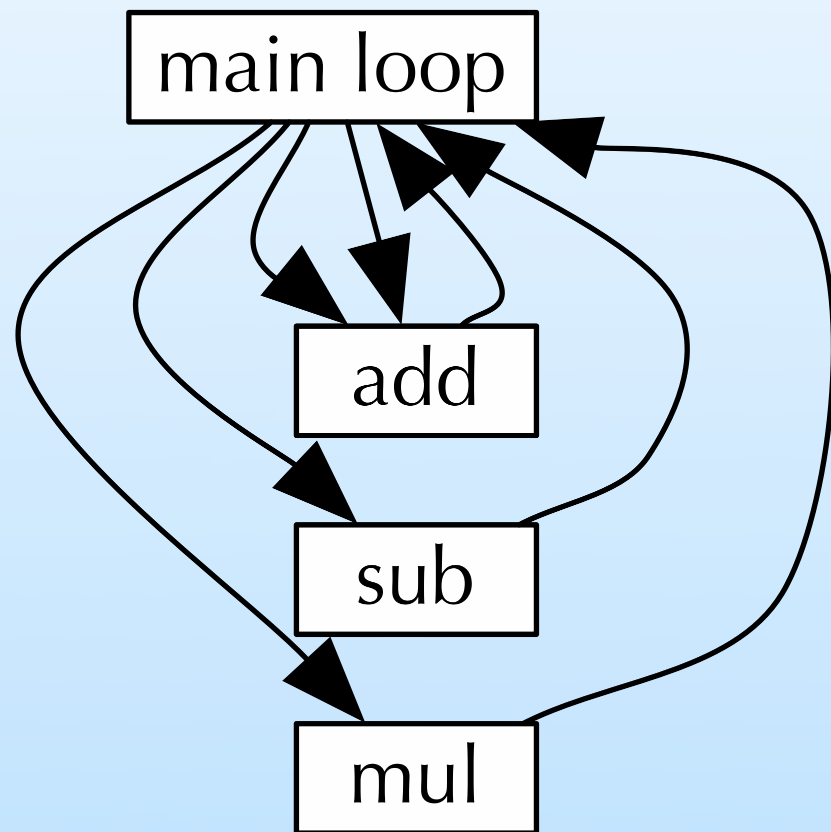
Threaded code

In a `switch`-based interpreter, each instruction requires two jumps:

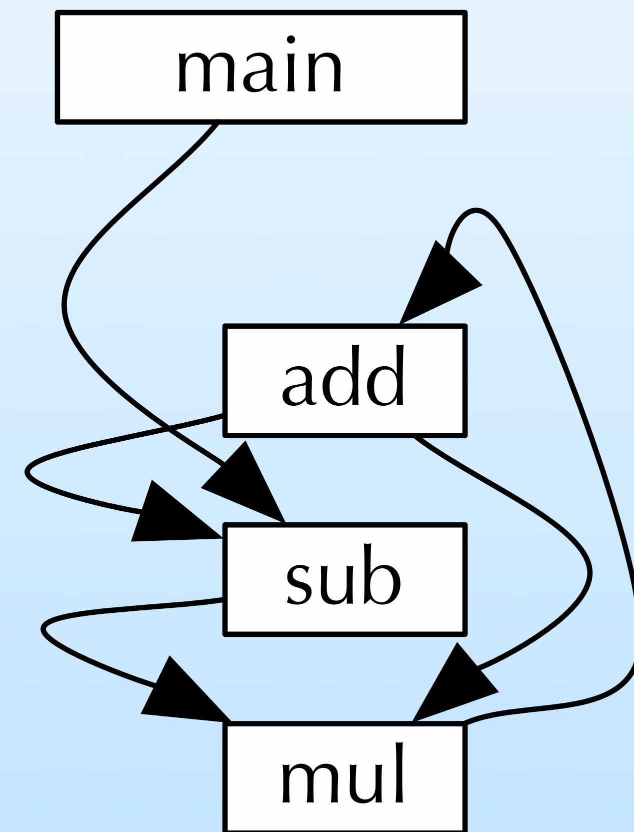
- one indirect jump to the branch handling the current instruction,
- one direct jump from there to the main loop.

It would be better to avoid the second one, by jumping directly to the code handling the next instruction. This is called **threaded code**.

Non-threaded and threaded code



Non-threaded
(`switch`-based)



Threaded

Implementing threaded code

To implement threaded code, there are two main techniques:

- with **indirect threading**, instructions index an array containing pointers to the code handling them,
- with **direct threading**, instructions are pointers to the code handling them.

Threaded code in C

To implement threaded code, it must be possible to manipulate code pointers.

In ANSI-C, the only way to do this is to use function pointers.

The Gnu C compiler (`gcc`) allows the manipulation of labels as values, which is much more efficient!

Direct threaded code in ANSI C

```
typedef void (*Instruction)();

int* sp;

void add() {
  sp[1] = sp[0] + sp[1];
  sp++;
}

Instruction program[] = { add /* ... */ };
Instruction* ip = program;

void interpret() {
  for (;;)
    (*ip++)();
}
```

Warning: quite inefficient in practice!

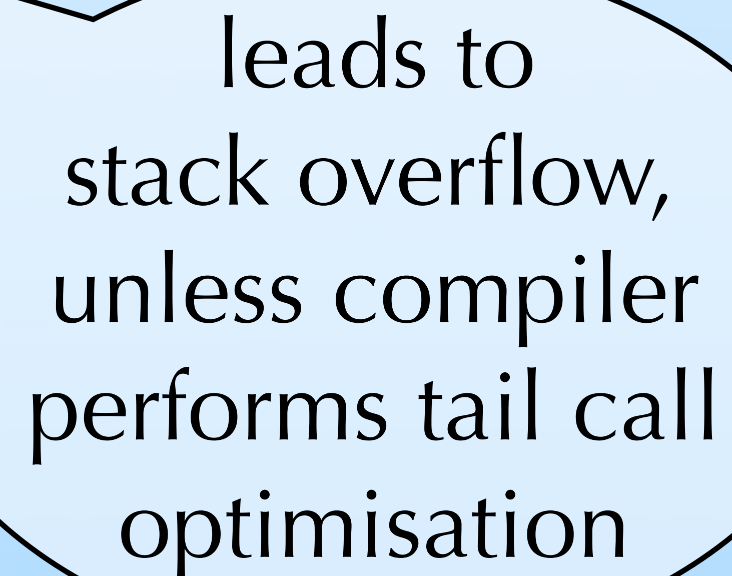
Direct threaded code in ANSI C (with tail calls)

```
typedef void (*Instruction)();

int* sp;

void add(Instruction* ip, int* sp) {
  sp[1] = sp[0] + sp[1];
  (*ip)(ip + 1, sp + 1);
}

Instruction program[] = {
  add /* ... */
};
```



Direct threaded code with gcc's labels as values

```
typedef void* Instruction;

void interpret() {
  static Instruction program[] = { &&add /* ... */ };
  Instruction* ip = program;
  int* sp;

  goto *ip++;

 add:
  sp[1] = sp[0] + sp[1];
  sp++;
  goto *ip++;
}
```

Top of stack caching

In a stack-based VM, the stack is typically represented as an array in memory. Since almost all instructions access that array, it can be interesting to store some of it in registers.

Keeping a *fixed* number of stack elements in registers is usually a bad idea: imagine what happens when an instruction pops one stack element, and the next one pushes one element.

Caching a varying number of elements is better.

Top of stack caching

Caching a varying number of stack elements in registers complicates the implementation of instructions.

There must be one implementation of each VM instruction per cache state – defined as the number of stack elements currently cached in registers.

(Some instructions can have the same implementation for several states)

Top of stack caching

For example, if we admit that between zero and two stack elements can be cached in registers, there are three implementations of the `add` instruction, one per cache state:

- the first fetches both operands from the stack,
- the second fetches one operand from the stack, the other from registers,
- the last fetches both operands from registers.

(Static) super-instructions

Since instruction dispatch is expensive in a VM, one way to reduce its cost is to dispatch less...

This can be done by grouping several instructions which often appear in sequence into a **super-instruction**.

Profiling is typically used to determine which sequences should be transformed into super-instructions, and the instruction set of the VM is then modified accordingly.

Dynamic super-instructions

It is also possible to generate super-instructions at run time, to adapt them to the program being run. This is the idea behind **dynamic super-instructions**.

It is possible to push this technique to its limits, and generate one super-instruction for *every basic block* of the program! This effectively transform all basic blocks into single (super-)instructions.

Just-in-time compilation

Virtual machines can be sped up through the use of **just-in-time** (or dynamic) **compilation**.

The basic idea is relatively simple: instead of interpreting the code, first compile it to native code, and then run that.

In practice, there are several difficulties to solve.

JIT: how to compile?

JIT compilers have one constraint that off-line compilers do not have: they must be fast – fast enough to make sure the time lost compiling the code is regained during its execution.

For that reason, JIT compilers usually do not use costly optimisation techniques – HotSpot server being one exception.

JIT: what to compile?

Some code is executed only once over the whole run of a program. It is usually faster to interpret that code than go through JIT compilation.

Therefore, it is better to start by interpreting all code, monitoring execution to see which code is executed often – a so-called *hot spot*.

Once a hot spot is identified, it gets compiled to native code.

Virtual machine generators

Several tools have been written to automate the creation of virtual machines based on a high-level description.

`vmgen` is such a tool, which we will briefly examine.

`vmgen`

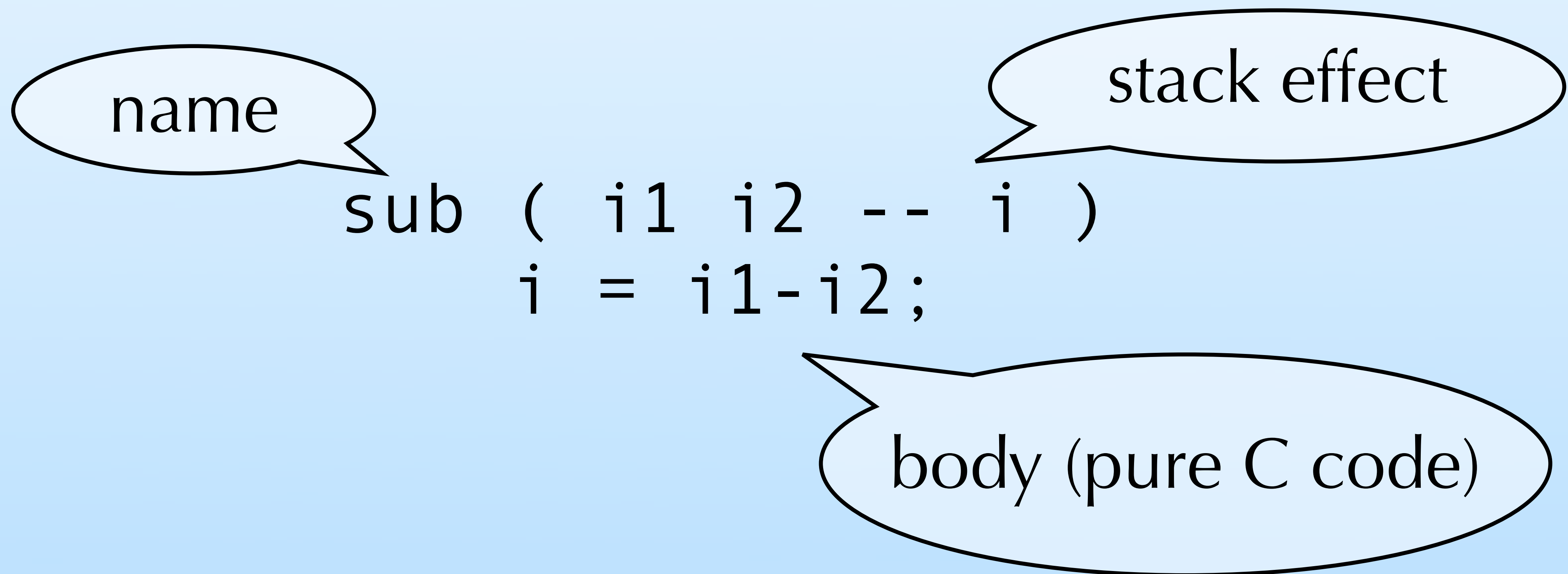
Based on a single description of the VM, `vmgen` can produce:

- an efficient interpreter, with optional tracing,
- a disassembler,
- a profiler.

The interpreter includes all the optimisations we have seen – threaded code, super-instructions, top-of-stack caching – and more.

vmgen

Example of instruction description:



Real-world virtual machines

As an example of a real virtual machine, we will now look at the Java Virtual Machine.

Other virtual machines worth mentioning include:

- Microsoft's CLR,
- the Parrot VM, designed for Perl 6,
- etc.

The Java Virtual Machine (JVM)

The Java Virtual Machine is a stack-based VM targeted towards the execution of compiled Java programs.

It is quite high-level, and has almost all concepts of Java 1.0: classes, interfaces, methods, exceptions, monitors, etc.

While the Java language has evolved over the years, the JVM has remained the same.

The JVM model

The JVM is composed of:

- a stack, which stores intermediate values,
- a set of local variables, private to the method being executed,
- a heap, from which memory is allocated.

It accepts **class files** as input, each of which contain the definition of a single class/interface.

The language of the JVM

The JVM has 201 instructions to perform various tasks like loading of values on the stack, arithmetic operations, conditional jumps, etc.

One interesting feature of the JVM is that all instructions are *typed*. This features is used to support verification.

Example instructions: `iadd` (integer addition), `invokevirtual` (method invocation), etc.

JVM bytecode verification

A novel feature of the JVM is that it verifies programs before executing them, to make sure that they satisfy some safety requirements.

To enable this, all instructions are typed, and several restrictions are put on programs, *e.g.*:

- the stack size at any point in a method must be computable in advance,
- jumps must target statically known locations.

HotSpot JVM

HotSpot is the name of Sun's implementation of the JVM. Main features:

- interpreter which monitors execution to detect hot spots – later compiled to native code,
- two separate JIT compilers:
  1. a client compiler, fast but non-optimising,
  2. a server compiler, slower but optimising (based on SSA).

Summary

Interpreters enable the execution of a program without having to compile it to native code, thereby simplifying P.L. implementation.

Virtual machines are the most common kind of interpreters, and are a good compromise between ease of implementation and speed.

Several techniques exist to make VMs fast: threaded code, top-of-stack caching, super-instructions, JIT compilation, etc.