CS107: Register Allocation and Virtual Machines

Guannan Wei
guannan.wei@tufts.edu

March 31, 2026

Spring 2026

Tufts University

Last Time

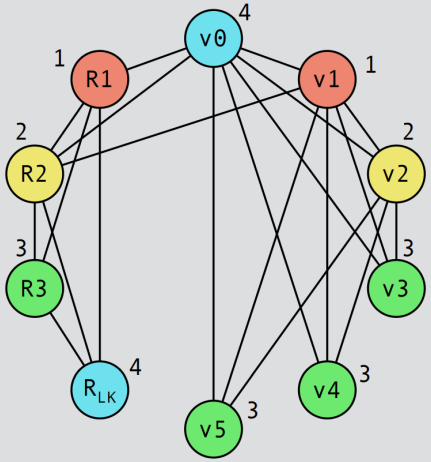
- Register allocation: assign virtual registers to physical registers
- Simplification and heuristics for graph coloring
  - Liveness analysis
  - Interference graph
- Spilling: move some virtual registers to memory when there are not enough physical registers
- Coalescing: eliminate move instructions by assigning the same register

Graph Coloring

Original prog.

```
gcd:
  v0 ← RLK
  v1 ← R1
  v2 ← R2
loop:
  v3 ← done
  if v2=0 goto v3
  v4 ← v2
  v2 ← v1 % v2
  v1 ← v4
  v5 ← loop
  goto v5
done:
  R1 ← v1
  goto v0
```

Colored interference graph



Rewritten prog.

```
gcd:
  RLK ← RLK
  R1 ← R1
  R2 ← R2
loop:
  R3 ← done
  if R2=0 goto R3
  R3 ← R2
  R2 ← R1 % R2
  R1 ← R3
  R3 ← loop
  goto R3
done:
  R1 ← R1
  goto RLK
```

Today

- Additional constraints on register allocation
- Linear scan register allocation
- Interpreters and virtual machines

Assignment Constraints

Until now, we assume that a virtual register can be assigned to any physical register, as long as it is free.

In practice, this is often not the case, as various architectural characteristics impose assignment constraints, e.g.:

Assignment Constraints

Until now, we assume that a virtual register can be assigned to any physical register, as long as it is free.

In practice, this is often not the case, as various architectural characteristics impose assignment constraints, e.g.:

- some architectures classify registers in several classes, with different capabilities (e.g. address vs. data registers, integer vs. floating-point registers, etc.),
- some instructions require some of their arguments/result to be in specific registers,
- calling conventions require function arguments and results to be in specific registers.

A realistic register allocator has to be able to satisfy these constraints.

Register Classes

Most architectures separate the registers in several classes. Even in modern RISC architectures, there is typically one class for floating-point values and another one for integers and pointers.

Register classes can easily be taken into account in a coloring-based allocator:

- If a variable must be put in a register of some class, then its node can be made to interfere with all pre-colored nodes corresponding to registers of other classes (preventing to assign it to a register of the wrong class).

Calling Conventions

Many calling conventions pass arguments in registers.

At the beginning of all functions, move instructions have to be inserted to copy the arguments to new virtual registers:

```
fact:
  v1 ← R1 // save first argument in v1
```

Calling Conventions

Many calling conventions pass arguments in registers.

At the beginning of all functions, move instructions have to be inserted to copy the arguments to new virtual registers:

```
fact:
  v1 ← R1 // save first argument in v1
```

Similarly, before any function call, move instructions have to be inserted to save the arguments into the appropriate registers:

```
R1 ← v2 // load first argument from v2
CALL fact
```

Whenever possible, theses move instructions will be removed by coalescing.

Caller/Callee-Saved Registers

Calling conventions distinguish two kinds of registers:

- **caller-saved** registers are saved by the caller before a call and restored after it,
- **callee-saved** registers are saved by the callee at function entry and restored before function exit.

Caller/Callee-Saved Registers

Calling conventions distinguish two kinds of registers:

- **caller-saved** registers are saved by the caller before a call and restored after it,
- **callee-saved** registers are saved by the callee at function entry and restored before function exit.

Ideally, all virtual registers that have to survive at least one call should be assigned to callee-saved registers, while other virtual registers should be assigned to caller-saved registers.

How can this be obtained in a coloring-based allocator?

Caller/Callee-Saved Registers

The contents of **caller-saved** registers do not survive a function call.

- Edges are added to the interference graph between all virtual registers that are live across at least one call and (physical) caller-saved registers.
- These edges ensure that virtual registers that are live across at least one call will not be assigned to caller-saved registers, and will therefore either be spilled or allocated to callee-saved registers!

Saving Callee-Saved Registers

Callee-saved registers must be preserved by all functions.

This can be achieved by copying them to fresh temporary registers at function entry and restoring them before exit.

Saving Callee-Saved Registers

For example, if R_8 is a callee-saved register, a function could look like:

```
entry:
  v_1 ← R_8 // save callee-saved R_8 in v_1
  ...       // function body
  R_8 ← v_1 // restore callee-saved R_8
  goto R_LK
```

- If the register pressure is low, then R_8 and v_1 will be coalesced, and the two move instructions removed.
- If register pressure is high, v_1 will be spilled, thereby making R_8 available in the function body, e.g. to store a virtual register live across a call.

Technique #2: Linear Scan

The basic linear scan technique is very simple:

- the program is linearized — i.e. represented as a linear sequence of instructions, not as a graph,
- a unique live range is computed for every variable, going from the first to the last instruction during which it is live,
- registers are allocated by iterating over the intervals sorted by increasing starting point: each time an interval starts, the next free register is allocated to it, and each time an interval ends, its register is freed,
- if no register is available, the active range ending last is chosen to have its variable spilled.

Linear Scan Example

Let's try to allocate registers for our `gcd` procedure using linear scan. Let's assume there are 4 allocable registers.

Program

```
1  gcd:   v0 ← RLK
2         v1 ← R1
3         v2 ← R2
4  loop:  v3 ← done
5         if v2=0 goto v3
6         v4 ← v2
7         v2 ← v1 % v2
8         v1 ← v4
9         v5 ← loop
10        goto v5
11 done:  R1 ← v1
12        goto v0
```

Live ranges

v_0: $[1^+, 12^-]$

v_1: $[2^+, 11^-]$

v_2: $[3^+, 10^+]$

v_3: $[4^+, 5^-]$

v_4: $[6^+, 8^-]$

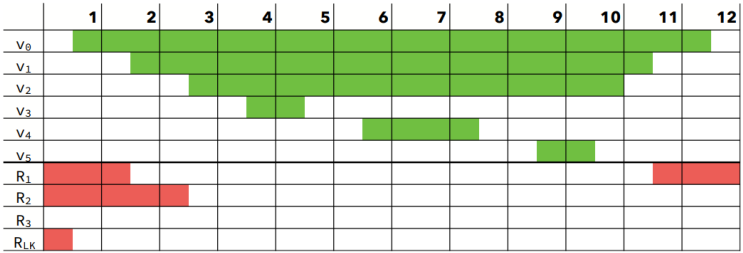
v_5: $[9^+, 10^-]$

Notation:

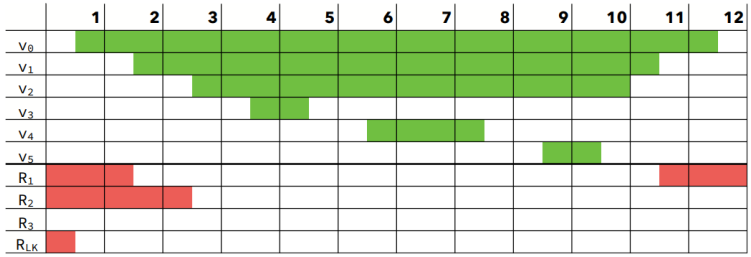
i^+ entry of instr. i

i^- exit of instr. i

Linear Scan Example (K = 4)

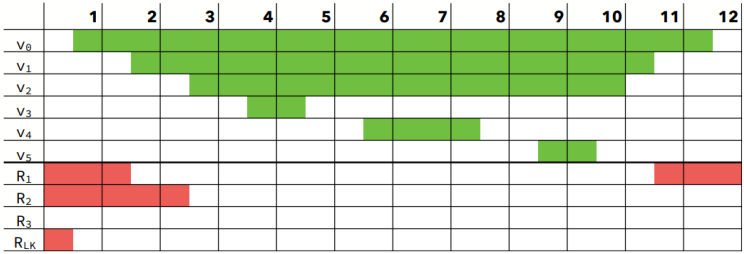


Linear Scan Example (K = 4)



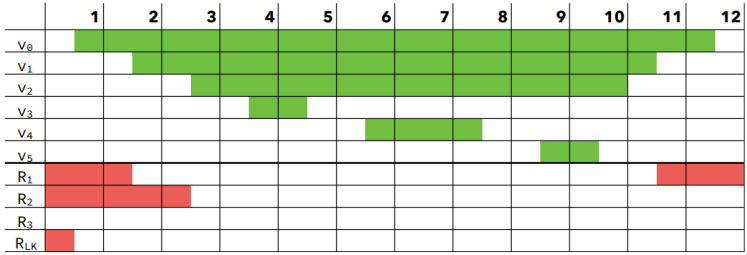
time	active intervals	allocation
1^+	$[1^+, 12^-]$	$v_0 \rightarrow R_3$

Linear Scan Example (K = 4)



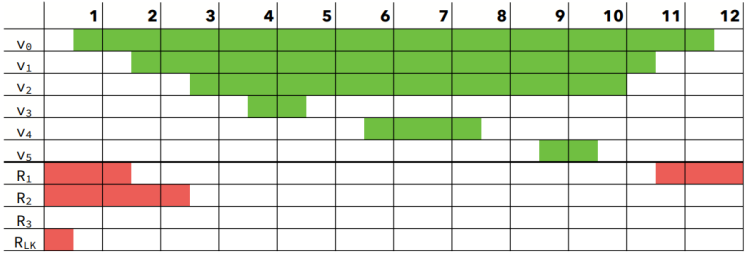
time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \rightarrow R_3$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1$

Linear Scan Example (K = 4)



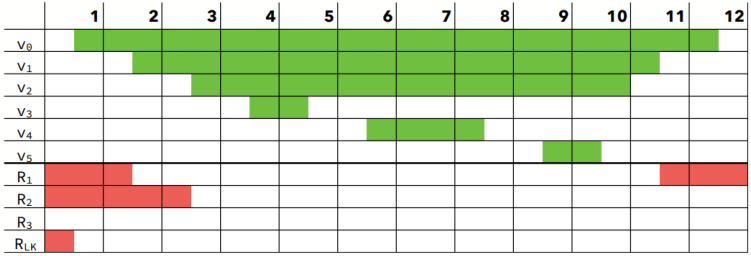
time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \rightarrow R_3$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1$
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2$

Linear Scan Example (K = 4)



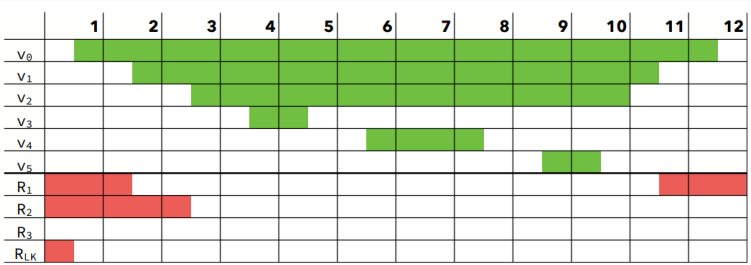
time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \rightarrow R_3$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1$
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2$
4^+	$[4^+,5^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_3 \rightarrow R_{LK}$

Linear Scan Example (K = 4)



time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \rightarrow R_3$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1$
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2$
4^+	$[4^+,5^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_3 \rightarrow R_{LK}$
6^+	$[6^+,8^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_4 \rightarrow R_{LK}$

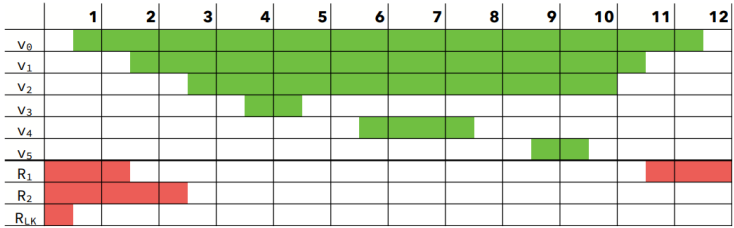
Linear Scan Example (K = 4)



time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \rightarrow R_3$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1$
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2$
4^+	$[4^+,5^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_3 \rightarrow R_{LK}$
6^+	$[6^+,8^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_4 \rightarrow R_{LK}$
9^+	$[9^+,10^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_5 \rightarrow R_{LK}$

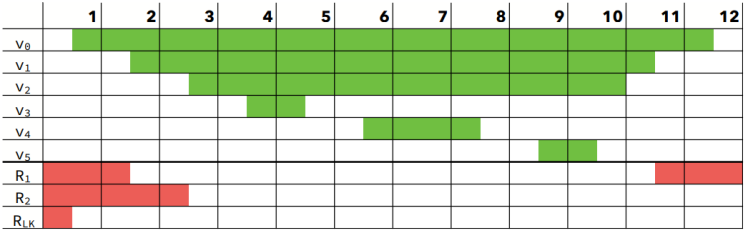
Result: no spilling

Linear Scan Example (K = 3)



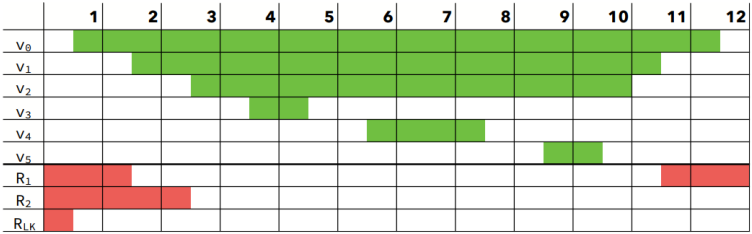
time **active intervals** **allocation**

Linear Scan Example (K = 3)



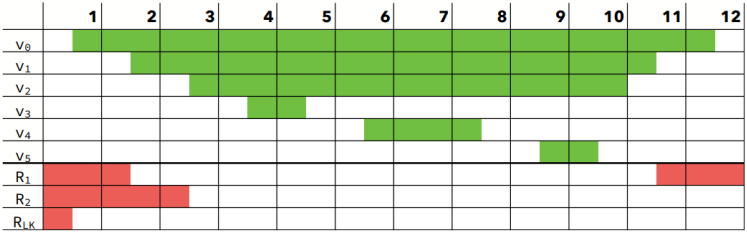
time	active intervals	allocation
1^+	$[1^+, 12^-]$	$v_0 \rightarrow R_{LK}$

Linear Scan Example (K = 3)



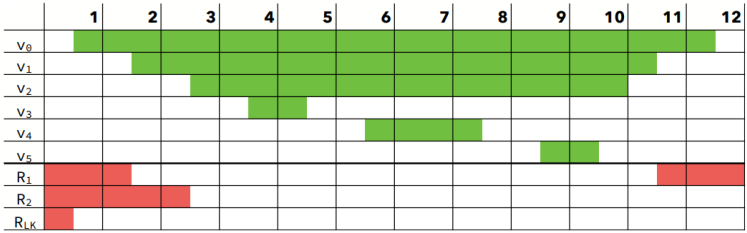
time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \rightarrow R_{LK}$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_{LK}$, $v_1 \rightarrow R_1$

Linear Scan Example (K = 3)



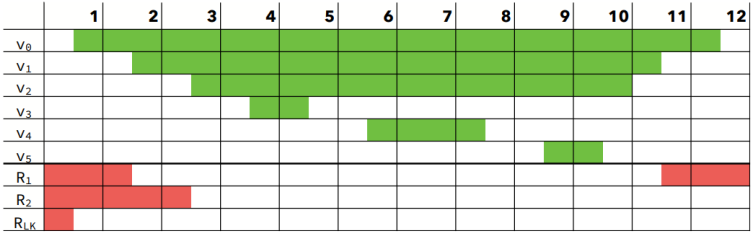
time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \rightarrow R_{LK}$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_{LK}, v_1 \rightarrow R_1$
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_{LK}, v_1 \rightarrow R_1, v_2 \rightarrow R_2$

Linear Scan Example (K = 3)



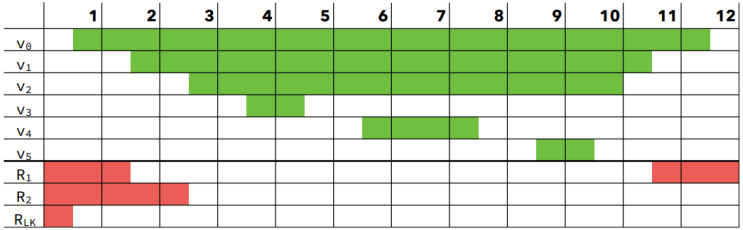
time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \rightarrow R_{LK}$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_{LK}$, $v_1 \rightarrow R_1$
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_{LK}$, $v_1 \rightarrow R_1$, $v_2 \rightarrow R_2$
4^+	$[4^+,5^-],[3^+,10^+],[2^+,11^-]$	$v_0 \rightarrow S$, $v_1 \rightarrow R_1$, $v_2 \rightarrow R_2$, $v_3 \rightarrow R_{LK}$

Linear Scan Example (K = 3)



time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \to R_{LK}$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \to R_{LK}, v_1 \to R_1$
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \to R_{LK}, v_1 \to R_1, v_2 \to R_2$
4^+	$[4^+,5^-],[3^+,10^+],[2^+,11^-]$	$v_0 \to S, v_1 \to R_1, v_2 \to R_2, v_3 \to R_{LK}$
6^+	$[6^+,8^-],[3^+,10^+],[2^+,11^-]$	$v_0 \to S, v_1 \to R_1, v_2 \to R_2, v_4 \to R_{LK}$

Linear Scan Example (K = 3)



time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \rightarrow R_{LK}$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_{LK}, v_1 \rightarrow R_1$
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \rightarrow R_{LK}, v_1 \rightarrow R_1, v_2 \rightarrow R_2$
4^+	$[4^+,5^-],[3^+,10^+],[2^+,11^-]$	$v_0 \rightarrow S, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_3 \rightarrow R_{LK}$
6^+	$[6^+,8^-],[3^+,10^+],[2^+,11^-]$	$v_0 \rightarrow S, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_4 \rightarrow R_{LK}$
9^+	$[9^+,10^-],[3^+,10^+],[2^+,11^-]$	$v_0 \rightarrow S, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_5 \rightarrow R_{LK}$

Result: v_0 is spilled *during its whole life time*!

Linear Scan Improvements

The basic linear scan algorithm is very simple but still produces reasonably good code. It can be - and has been - improved in many ways:

- the liveness information about virtual registers can be described using a sequence of disjoint intervals instead of a single one,
- virtual registers can be spilled for only a part of their whole life time,
- more sophisticated heuristics can be used to select the virtual register to spill,

Runtime Systems

Runtime Systems

- Interpreters
- Virtual machines
- Garbage collection

Interpreters

An **interpreter** is a program that executes another program, represeted as some kind of data structure.

Common program representations include:

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

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 many 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 ever expression is evaluated at most once (i.e., languages without loops or functions).

Text-based Interpreters

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

They are very seldom used except for trivial languages where ever expression is evaluated at most once (i.e., languages without 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.

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.

All the interpreters included in the MiniScala compiler are also tree-based.

Virtual Machines

Virtual machines resemble real processors, but are implemented in software. They accept as input a program composed of a sequence of instructions.

Virtual Machines

Virtual machines resemble real processors, but are implemented in software. They accept as input a program composed of a sequence of instructions.

Virtual machines often provide more than the simple interpretation of programs: they also abstract the underlying system by managing memory, threads, and sometimes I/O.

Virtual Machines

Virtual machines resemble real processors, but are implemented in software. They accept as input a program composed of a sequence of instructions.

Virtual machines often provide more than the simple interpretation of programs: they also abstract the underlying system by managing memory, threads, and sometimes I/O.

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?

Why Virtual Machines?

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

- portability: compiled VM code can be run on many actual machines
- simplicity: a VM is usually more high-level than a real machine, which simplifies the task of the compiler
- simplicity, redux: a VM is easier to monitor and profile, which eases debugging

Virtual Machine Drawbacks

The only drawback of virtual machines compared to physical ones is slower performance.

Virtual Machine Drawbacks

The only drawback of virtual machines compared to physical ones is slower performance.

This is due to the overhead associated with interpretation:

- Fetching and decoding instructions, executing them, etc.
- Moreover, the high number of indirect jumps in interpreters causes pipeline stalls in modern processors.

Virtual Machine Drawbacks

The only drawback of virtual machines compared to physical ones is slower performance.

This is due to the overhead associated with interpretation:

- Fetching and decoding instructions, executing them, etc.
- Moreover, the high number of indirect jumps in interpreters causes pipeline stalls in modern processors.

To a (sometimes large) degree, this is mitigated by the tendency of modern VMs to compile the program being executed, and to perform optimizations based on its behavior (JIT compilation).

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.

For a compiler writer, it is usually easier to target a stack-based VM than a register-based VM, as the complex task of register allocation can be avoided.

Kinds of Virtual Machines

```
public static void main(String[] args) {
  int a = 1;
  int b = 2;
  int c = a + b;
  System.out.println(c);
}
```

JVM Bytecode:

```
...
4: iload_1
5: iload_2
6: iadd
...
```

Most widely-used virtual machines today are stack-based (e.g., the JVM, .NET's CLR, WebAssembly, etc.), but a few recent ones are register-based (e.g., Lua).

Virtual Machine Input

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

Each instruction is identified by its **opcode** (**op**eration **code**), which is 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.

VM Implementation

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

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

VM Implementation

Many VMs today are written in C or C++ because these languages are at the right abstraction level for the task, fast, and relatively portable.

VM Implementation

Many VMs today are written in C or C++ because these languages are at the right abstraction level for the task, fast, and relatively portable.

As we will see later, the GNU C compiler (GCC) has an extension that makes it possible to use labels as normal values. This extension can be used to write very efficient VMs, and for that reason, several of them are written specifically for GCC or compatible compilers.

Implementing a VM in C

```c
typedef enum {
  add, /* ... */
} instruction_t;

void interpret() {
  static instruction_t program[] = { add /* ... */ };
  instruction_t* pc = program;
  int* sp = ...; /* stack pointer */
  for (;;) {
    switch (*pc++) {
      case add:
        sp[1] += sp[0];
        sp++;
        break;
      /* ... other instructions */
    }
  }
}
```

Optimizing VMs

The basic, `switch`-based implementation of the virtual machine just presented can be made faster using several techniques:

- *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, and
- 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 the idea of **threaded code**.

Switch vs Threaded

Program: `add sub mul`