Optimising Virtual Machines

Lachlan Patrick
loki@research.canon.com.au

Who is Loki?

- Has a Ph.D. from Sydney University
- Met Rob Pike (Inferno – Dis VM – Limbo) (see http://vitanuova.com/ for code)
- Software Engineering / Virtual Machines
- Has written four virtual machines

Overview

- A brief definition of virtual machines
- Stack-based vs Register-based VMs
- An implementation of a stack-based VM
- An implementation of a register-based VM
- Performance improvement tricks
- Probably no time for JITs (but we’ll see).

What are virtual machines?

- *Virtual machine* vs *Interpreter* – what’s the difference?
- I define an *interpreter* as “being capable of running source code, perhaps interactively”
- I define a *virtual machine* as “being capable of running code, possibly a compiled form of the code”
- An *emulator* is a virtual machine where the instruction set is based on a real CPU.

Examples of *Interpreters*

- Postscript interpreters, e.g. Ghostscript
- Lisp, Pascal interpreters
- Perl and Python languages
- DOS Batch language?
- Unix shells have a scripting language

Examples of *Virtual Machines*

- The Java VM (runs an object-oriented stack-based virtual machine language)
  - Java
  - JPython
  - Lisp
  - Prolog
- The C# VM
- Parrot is the Perl register-based VM

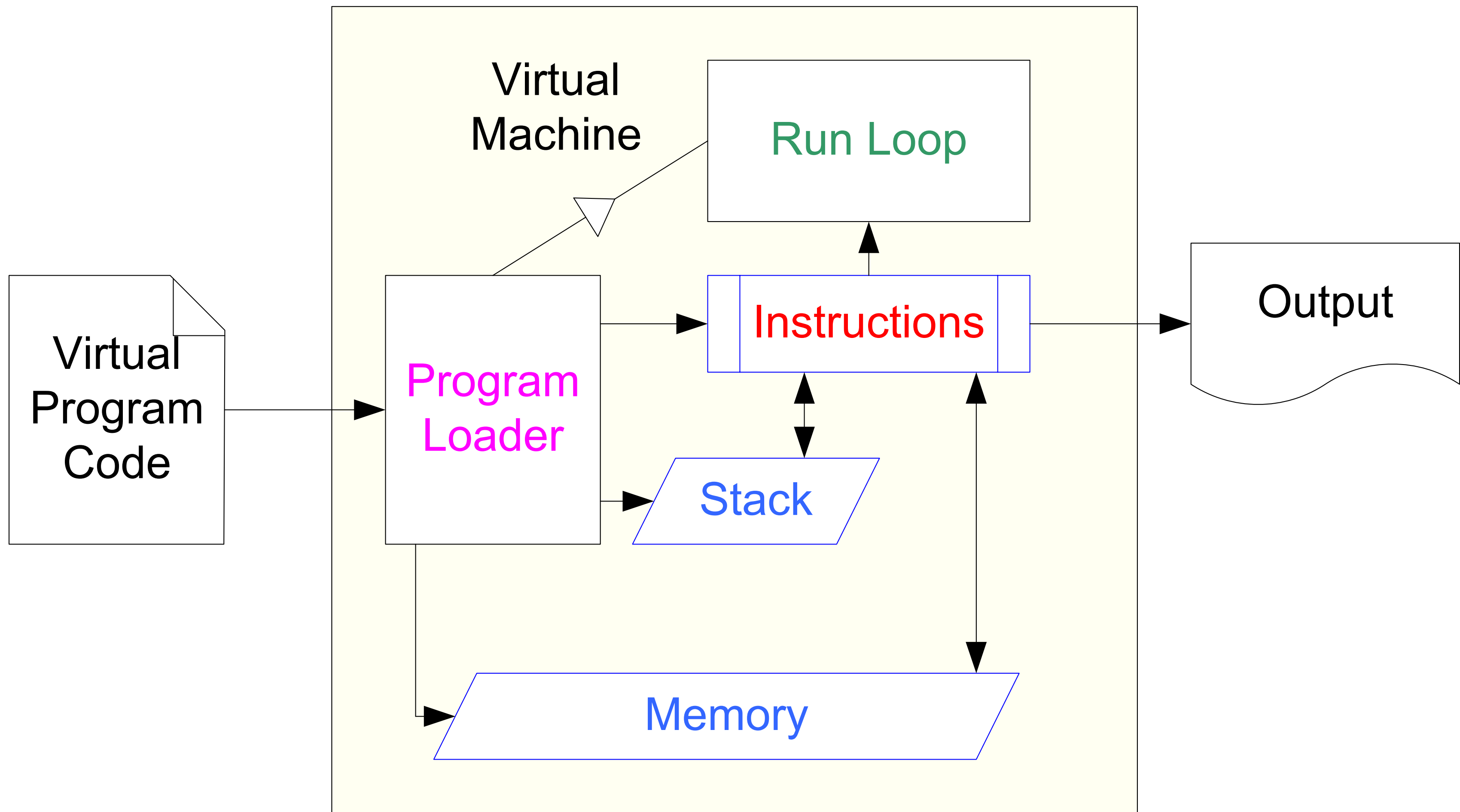
The Problem

- Speed
- Speed is a problem because a virtual program is:
  - *software* (the program) *running on*
  - *software* (the VM) *running on*
  - *hardware* (the CPU)
- VM performance is therefore important (but not always)

Why do people use VMs?

- Virtual machines are used for many reasons:
  - Portability (think Java and web applets)
  - Versatility (gluing disparate code together, e.g. Python, Perl, ICI)
  - String matching or regular expression parsing (e.g. Perl, ICI)
  - Emulation (I don’t have the hardware, but this software VM does the same thing…)

Stack-Based VM architecture



Java VM op-code examples

- iconst_0 — push 0 (int)
- dload_3 — push local double variable 3
- saload — push short int array elem b[a]
- iadd — push added int (a+b)
- lsub — push subtracted long (ab-cd)
- frem — push remainder float (a%b)
- ifne label — branch if int a != 0
- ireturn — return int a from method

A simple Stack-based VM

- An instruction ‘action’ function:

```
void add_int() {
    push(pop() + pop());
}
```

- The execution loop:

```
void run() {
    while (! finished) {
        Inst *i = fetch_inst();
        i->action();
    }
}
```

gcc –S output of C code on left

```
add_int:
   call pop
   copy r1,r3
   call pop
   copy r1,r4
   add  r3,r4,r2
   call push
   ret

run:
   jump L7test
L7:
   call fetch_inst
   copy r1[12],r5
   call r5
L7test:
   jeqz r9,L7
```

Tricks

- Execute as few instructions as possible within loops (include what that loop calls).
- Execute faster equivalent instructions.
- Avoid functions calls where possible.

A simpler Stack-based VM

```
void add_int() {
    *sp++ = (*(--sp) + *(--sp)); // dodgy
}
void halt() {
    exit(0);
}
void run() {
    while (true)
        (*pc++).action();
}
```

A simpler Stack-based VM

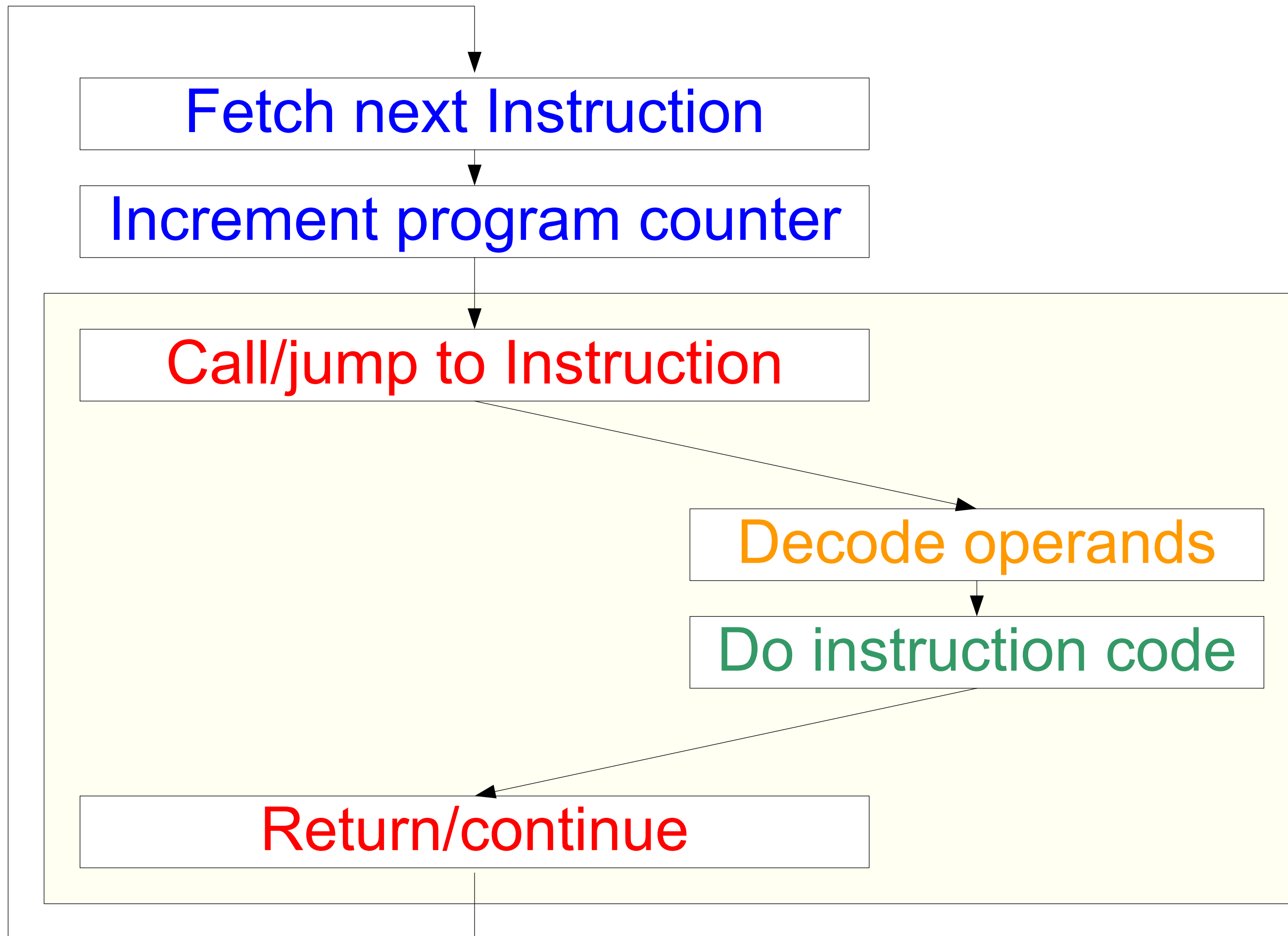
```
void add_int() {
    int src1 = *(--sp);
    int src2 = *(--sp);
    *sp++ = src1 + src2;
}

void run() {
    while (true)
        (*pc++).action();
}
```

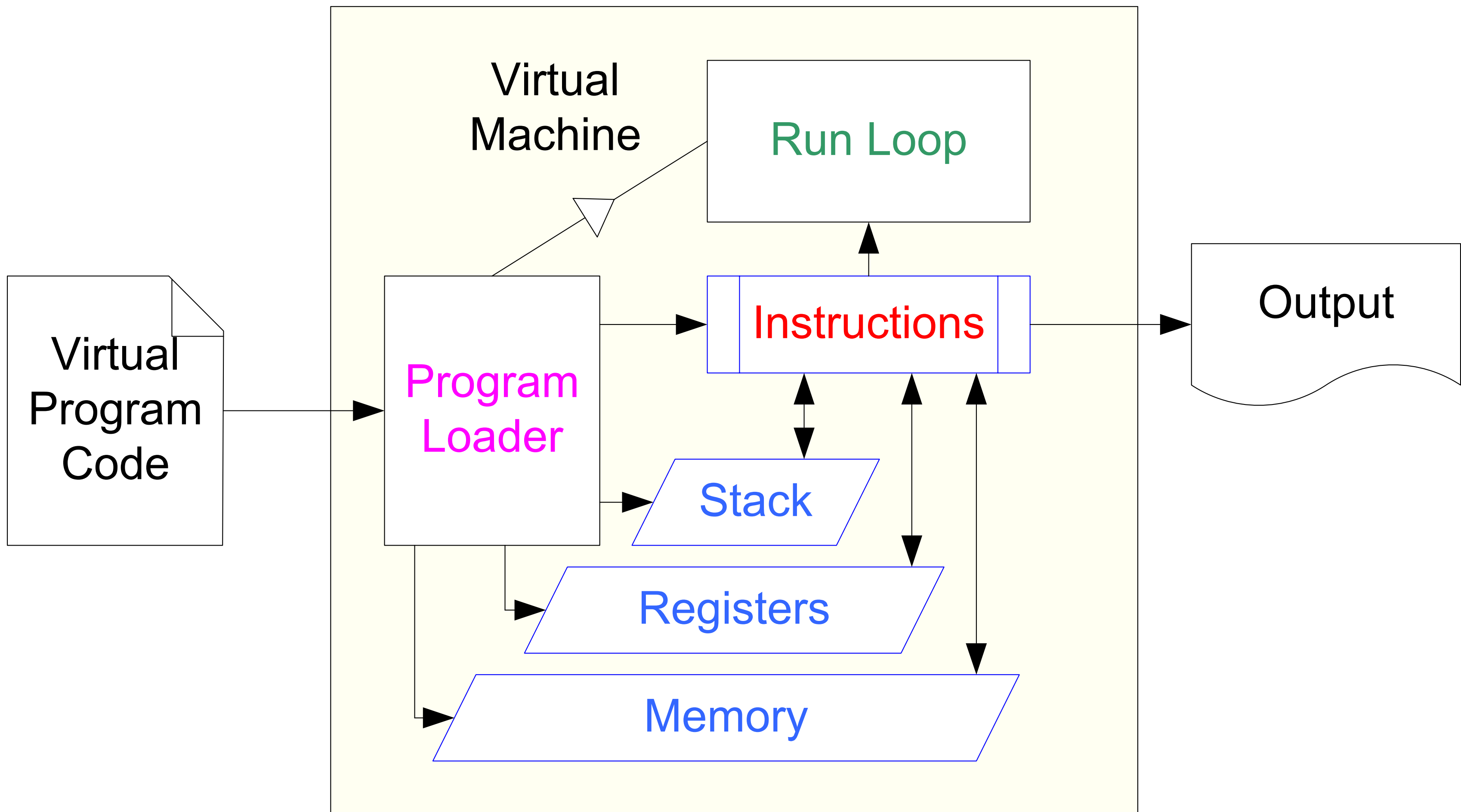
```
add_int:
   dec  sp
   copy *sp,r3
   dec  sp
   copy *sp,r4
   add  r3,r4,*sp
   inc  sp
   ret

run:
L7:
   copy pc[12],r5
   inc  pc
   call r5
   jump L7
```

VM Execution Loop



Register-Based VM architecture



A simple Register-based VM

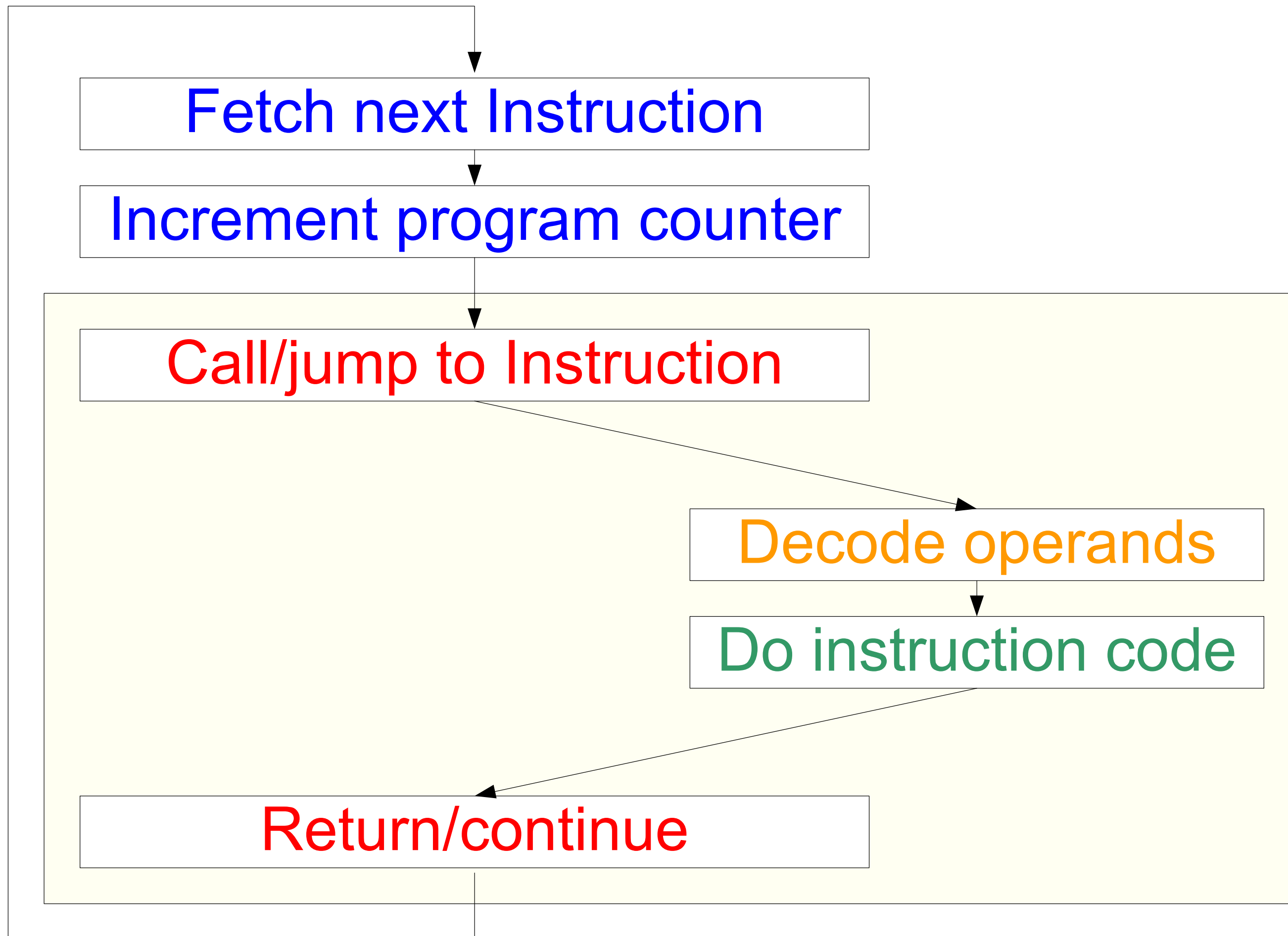
```
void add_int(int src1, int src2, int *dst) {
    *dst = src1 + src2;
}

int reg[32]; // array of registers
void run() {
  while (true) {
    Inst i = *pc++;
    i.action(reg[i.s1], reg[i.s2], &reg[i.d]);
  }
}
```

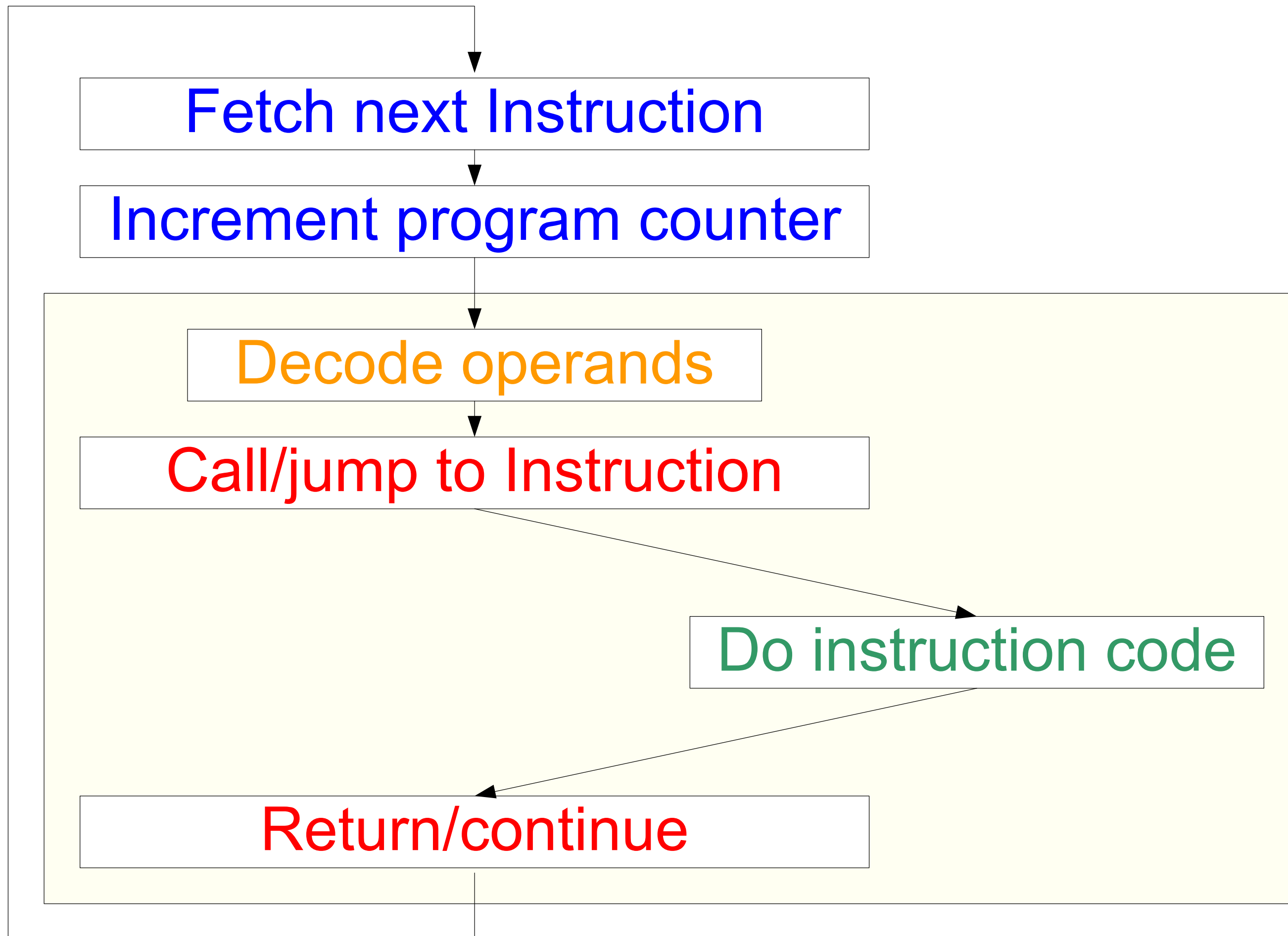
Registers vs Stacks

- Stacks:
  - Simple data model (stack = function args + locals).
  - Need extra commands to bring stack data to top.
  - JIT may be complex (real CPUs have registers).
- Registers:
  - Random access to local variables is convenient.
  - Fixed instruction function signature?
  - Real CPUs use registers. Compilers use ’em well.
  - JIT may be simpler (map VM registers onto CPU).

VM Execution Loop



VM Execution Loop 2



Decoding inside the Instruction

- Pros:
  - Flexibility (use only the operands you need)
  - Speed (use only the operands you need)
  - Speed (no parameters passed to the function)
- Cons:
  - Code size (decoding happens everywhere)
  - Complexity (decoding happens everywhere)
  - Correctness (e.g. stack overflow/underflow)

Decoding outside the Instruction

- Pros:
  - Security (e.g. avoid stack overflow/underflow)
  - Less code (decoding happens in one spot)
  - Correctness (decoding happens in one spot)
  - Faster to write (i.e. only the instruction)
- Cons:
  - Slower (introduces an if-statement into run loop?)
  - Slower (fixed number of operands to decode?)
  - Inflexibility (how many things to pop? types?)

Register-based pre-decoding VM

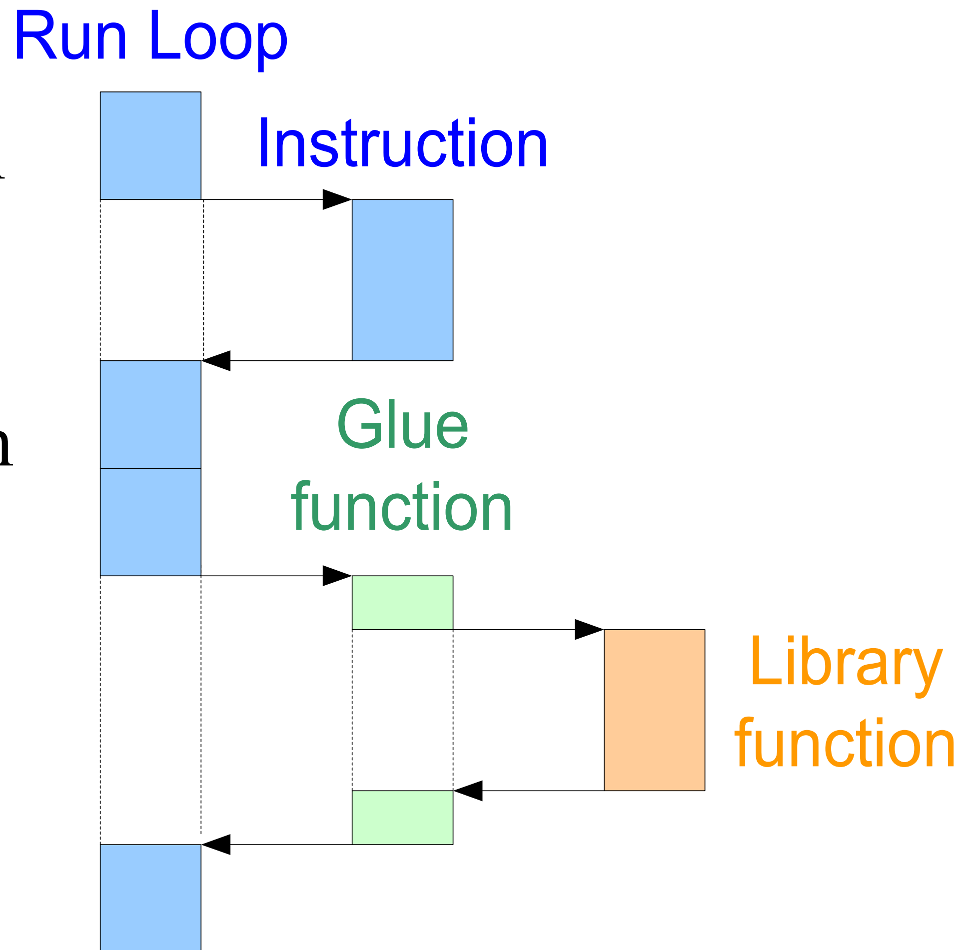
```
int reg[32];     // array of registers
int memory[BIG]; // memory of the VM
void run() {
  while (true) {
    Inst i = *pc++;
    int src1 = *(i.ptr1); // points to register!
    if (i.src1_in_memory)
      src1 = memory[src1]; // allow indirection
    i.action(src1, src2, dst);
  }
}
void add_int(int src1, int src2, int *dst) {
      *dst = src1 + src2;
}
```

Tricks

- Develop a sense of relative speeds:
  opcode < if < switch << function call
  (i.e. isub < jeq < jumptable << call)
- Pointers are faster than using array[index].
- Do as much work at load-time as possible, and as little at run-time as possible (e.g. static type-checking, point to registers not register indexes, Java doesn’t need to check for stack overflow at run-time!)
- Beware: loop unrolling and inline macros don’t always speed up code!

Glue Functions

- Instructions (built into the VM) are not enough to do all the work.
- We need a way to call additional libraries from our VM.



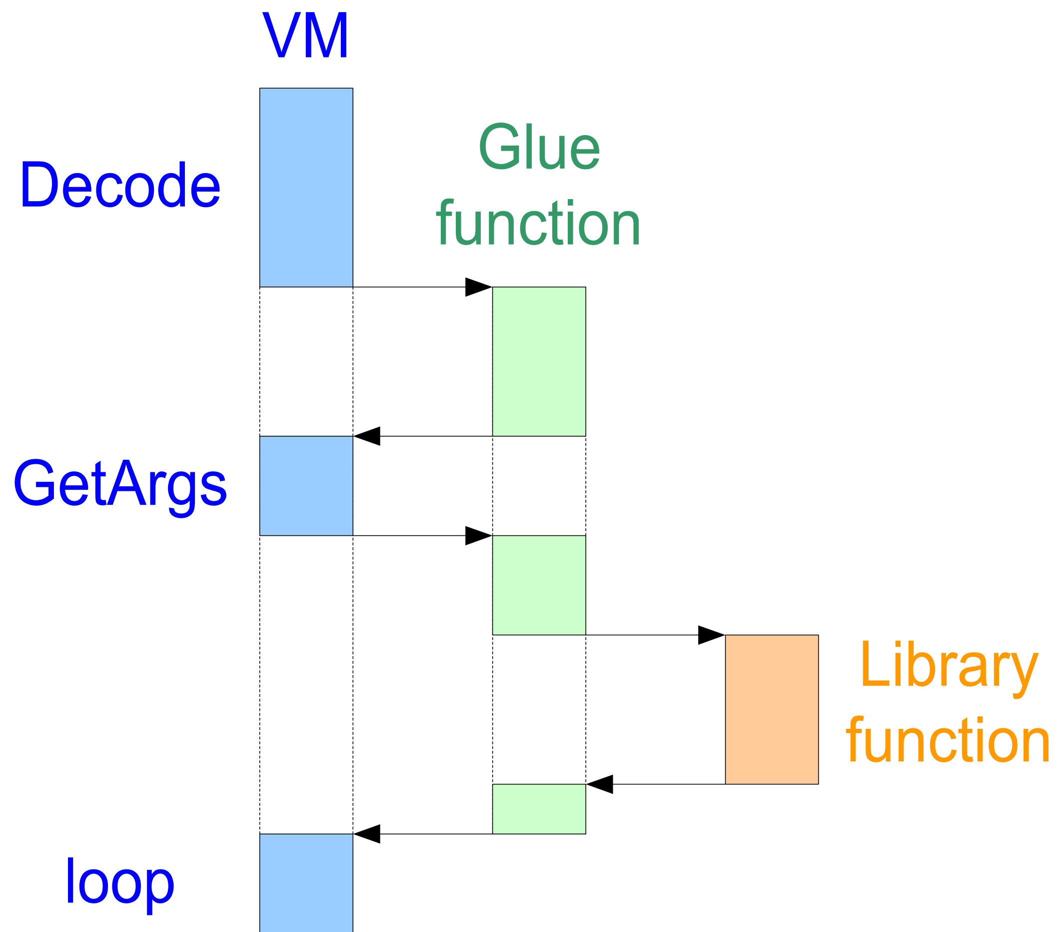
Python glue functions

```
static PyObject *
my_print(PyObject *self, PyObject *args)
{
  char *str;
  int retval;
  if (!PyArg_ParseTuple(args, "s", &str))
     return NULL;
  retval = printf("%s", str);
  return Py_BuildValue("i", retval);
}
```

ICI glue functions

```
static int
f_myfunc()
{
  long i;
  double f;
  if (ici_typecheck("if", &i, &f))
      return 1;
  printf("Got %ld, %lf.\n", i, f);
  return int_ret(i + 1);
}
```

Pulling data from the VM



Java glue functions

```
JNIEXPORT void JNICALL
Java_MyApplication_copyarray
 (JNIEnv *env, jclass myclass,
  jintArray i1, jintArray i2, jint size)
{
    long *iarr = malloc(sizeof(long)*size);
    env->GetIntArrayRegion(i1,0,size,iarr);
    env->SetIntArrayRegion(i2,0,size,iarr);
    free(iarr);

}
```

But which is best?

- I want numbers! I want answers!
- There’s only one way to get real answers:
- Test!

One test in a test suite

```
int main(int argc, char *argv[])
{
   long repeat;
   long r0, r4;

   scanf("%ld", &repeat);
   r0 = repeat;
   r4 = 7;
   while (0 < r0) {
        r4 = r4 / 2;
        r4 *= 13;
        r4 += repeat;
        r4 %= 11;
        r0 --;
   }
   printf("%ld", r4);
   printf(" was the result\n");
   return 0;
}
```

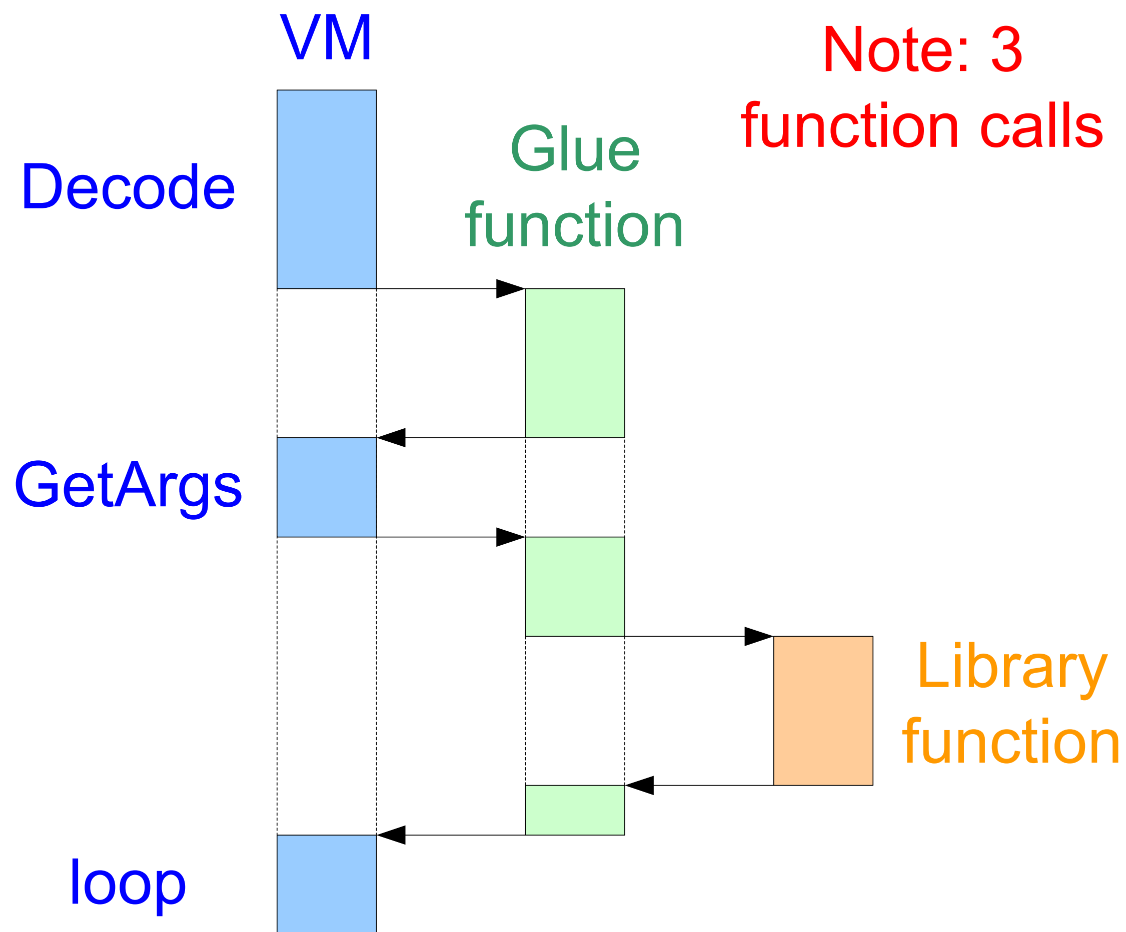
Integer instruction speed

	2 million		20 million
TECHNOLOGY	TIME	RAM	TIME
C (gcc -O2):	0.04s	336K	0.45s
C (gcc):	0.15s	336K	1.54s
Java 1.3.1:	0.57s	6204K	2.76s
LokiVM2002:	0.81s	400K	8.20s
Java no JIT:	0.91s	6668K	7.18s
LokiVM2000:	1.28s	416K	12.60s
AWK	2.72s	496K	27.10s
Lua 4.0:	4.58s	508K	(Wrong output!)
Perl 5.005:	4.70s	988K	46.43s
ICI 4.1.0:	5.65s	812K	56.45s
Python 2.1:	5.68s	1448K	55.93s
Python 1.5.2:	6.26s	1096K	62.44s

String instruction speed

```
                  1 million
TECHNOLOGY          TIME    RAM
C (factored):      4.84s   292K (strlen outside)
LokiVM2002:        5.73s   404K
Perl 5.005:        7.61s  1100K
C (naïve):         8.57s   292K (used gcc -O2)
LokiVM2000:        8.83s   420K
AWK:               9.47s   484K
Python 2.1:       11.54s  1456K
Python 1.5.2:     13.54s  1096K
LokiVM1999:       14.14s   404K
Java 1.3.1:       15.51s  7936K (in 2 processes)
ICI 4.1.0:        21.40s   792K
Lua 4.0:          >12hrs   760K-12MB (!)
```

Pulling data from the VM



Tricks

- The 80/20 rule: “make the stuff you do 80% of the time faster, and don’t worry if the other 20% gets slower.”
- Weights: switch << call
- Corollary: Sometimes more code is faster than less code.

Switch statements

- Instead of just calling all instructions as functions:

```
i.action(src1, src2, dst);
```

- Do some instructions inside the execution loop:

```
switch(i.opcode) {
    case INST_HALT:
        return;
    case INST_ADDI:
        *dst = src1 + src2;
        break;
    case INST_JEQZ:
        if (src1 == 0)
            pc += src2 - 1;
        break;
    ...
```

Glue code: therein lies the rub

- The problem is glue code is slow, hard to write, and no-one wants to do it.
- Maybe that’s fine if glue code is part of the uncommon 20% which we don’t care about, i.e. if 80% of the time our programs are doing built-in instructions like integer addition.
- Glue code is one case in the switch statement. And we just made it slower.

The glue code case

```
while (true) {
    ... // fetch and decode instruction
    switch(i.opcode) {
        case INST_ADDI:
                *dst = src1 + src2;
                break;
        ...
        case INST_GLUE:
               i.action(src1, src2, &dst);
               break;
        ...
```

Tricks

- Observation: void * is versatile (more so than int).
- So, split the decoding between the execution loop and the glue code:
  - Use pointers inside the execution loop
  - Do type-checking per instruction
  - Let the glue code cast to the correct type

Attack of the void pointers

- Inside the execution loop:

```
void *src1, *src2, *dst;
... // decode and run-time type-check
i.action(src1, src2, dst);
```

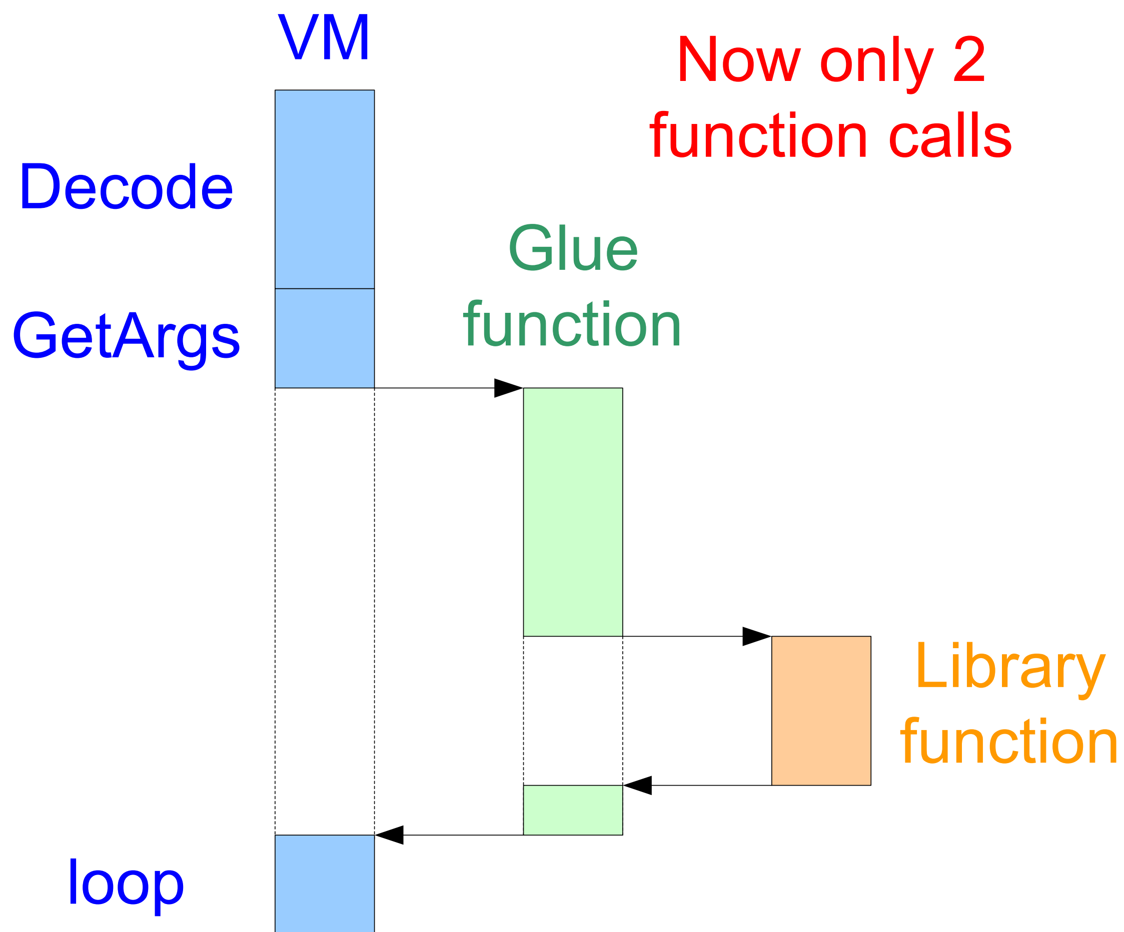
- The glue function:

```
void my_func(int *src1, int *src2,
             int *dst)
{
     *dst = *src1 | (*src2 >> 8);
}
```

Stocktaking

- Let’s take stock. What have we achieved?
  - Decoding now happens in the execution loop. Improves correctness, makes it easier to write glue code without fiddling with stack pointers or varargs.
  - Run-time type-checking is also in the run loop. Thus, we don’t need to call GetArgs (Yay! Faster!)
  - Built-in operations are also faster (80/20 rule).
- Problems remaining:
  - Inflexible three-parameter ‘action’ signature.

“GetArgs” now combined with execution loop



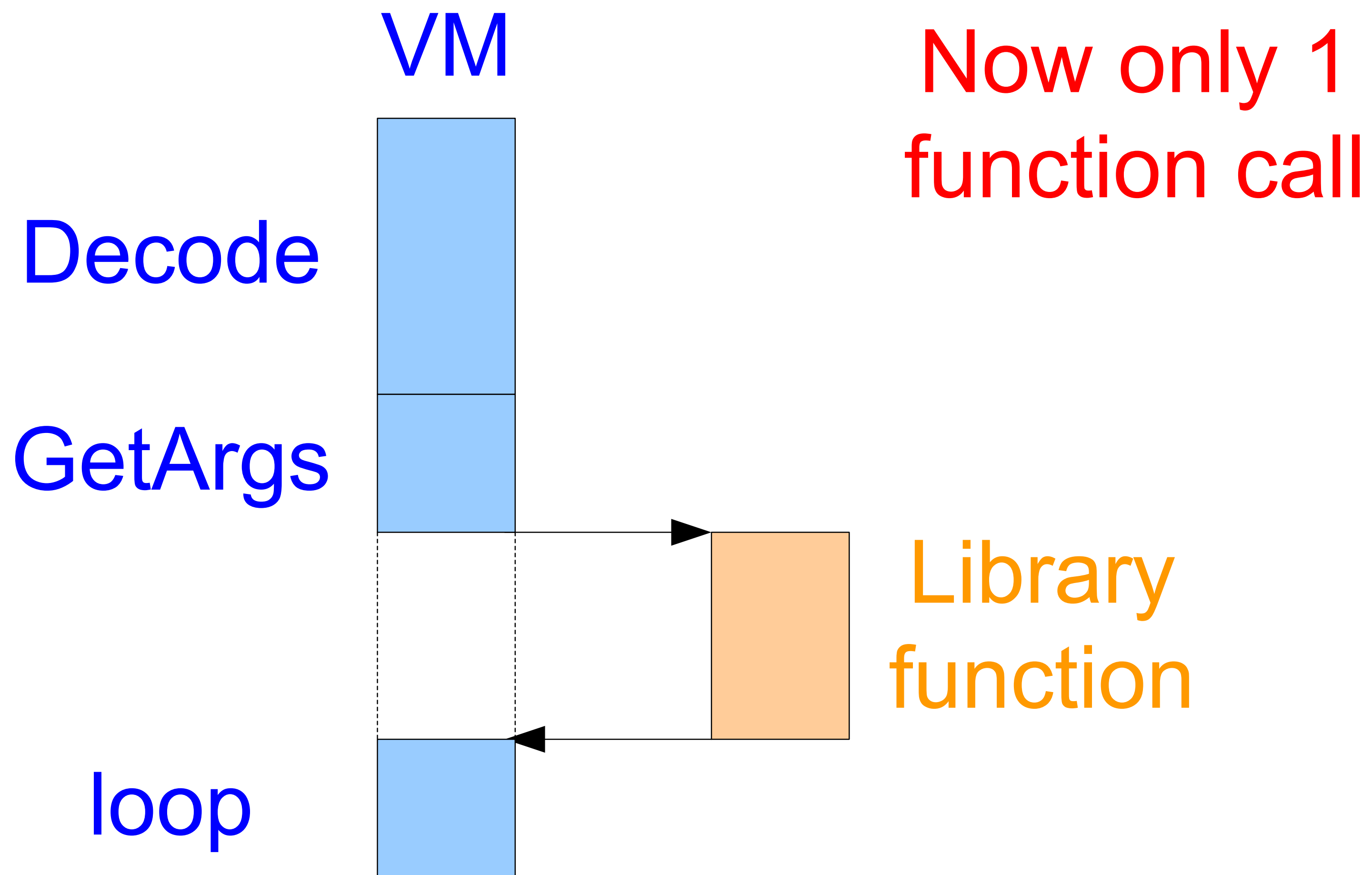
Tricks

- If in doubt, apply the 80/20 rule again.
  (We already used this rule to speed up the built-in instructions. Can we use it to make common glue code faster, yet still make uncommon glue code possible?)
- Observation: a jump table can become bigger without becoming slower.
- Corollary: “if you’ve already paid the toll, get your money’s worth.”

Multiple action signatures

```
while (true) {
   ... // fetch and decode first 3 params
   switch(i.opcode) {
     ... // built-in instructions cases
     case INST_GLUE3:
       i.action(arg1, arg2, arg3);
       break;
     case INST_GLUE4:
       ... // decode 4th parameter
       i.action(arg1, arg2, arg3, arg4);
       break;
     ...
```

Can we eliminate glue code *entirely*?



Just-In-Time compilers (JIT)

- A JIT converts virtual code at run-time into real CPU code. It’s a compiler in the VM.
- Advantage: speed!
- JITs are an old idea: regular expressions have been compiled internally to CPU code since 1968. In fact Ken Thompson wrote a paper on just that idea (K. Thompson, *Regular Expression Search Algorithm*, Comm. Assoc. Comp. Mach., Vol. 11, 6, pp. 419--422, 1968).

A JIT implementation

- The simplest JIT implementation inputs a virtual program, compiles it internally, then jumps to the start of that code (after flushing the CPU’s pipeline). On some platforms, it’s just a matter of casting a memory block to a function pointer and calling it!
- Smarter techniques only compile single virtual functions, or virtual loops, and only when that code is commonly used (don’t bother compiling single-use initialisation code).

Summary

- Function calls are modular, but switch statements are faster.
- Choose the trade-offs between speed and correctness.
- Sometimes tuning your code and thinking about data movements can produce radical improvements, even without using JITs/hardware/assembly language.
- Test and profile (it's the only way to be sure).