Virtual Machines Should Be Invisible

Stephen Kell

stephen.kell@cs.ox.ac.uk

joint work with

Conrad Irwin (University of Cambridge)
Spot the virtual machine (3) (Hint: they’re all invisible)
Hey, you got your VM in my Programming Experience™!

VMs don’t support programmers; they *impose on* them:

- limited language selection
- “foreign” code must conform to FFI
- debug with *per-VM* tools (jdb? pdb?)
- developing *across* VM boundaries? forget it!

Wanted:

- an end to FFI coding in the common case (assuming…)
- tools that work *across* VM boundaries

Focus on dynamic languages (→ Python for now)…
How we’re going to do it

Conventional VMs: “cooperate or die!”

■ you will conform
■ you will use my tools

“Less obtrusive” VMs:

■ “Describe yourself, alien!”
■ . . . and I’ll describe myself (to whole-process tools)

In particular:

■ extend underlying infrastructure: libdl, malloc, . . .
■ . . . and a shared descriptive metamodel—DWARF!
■ never (re)-invent opaque VM structures / protocols!
CPython, typical JVM, or similar

hand- or tool-generated FFI-based wrapper code

C library

operating system

instruction set architecture
Implementation tetris (2)

C library

operating system

native libs

user code

generic support libraries:
libunwind, libffi, libdl, ...

DwarfPython VM

compiler-generated debugging information

Virtual machines should be... – p.8/20
DwarfPython: an unobtrusive Python VM

DwarfPython is an ongoing implementation of Python which

- can import native libraries as-is
- can share objects directly with native code
- support debugging with native tools

Key components of interest:

- unified notion of function as *entry point(s)*
- extended `libdl` sees *all* code; entry point generator
- extensible objects (using DWARF + extended malloc)
- interpreter-created objects described by DWARF info

No claim to fully-implementedness (yet)...
### What is DWARF anyway?

```bash
$ cc -g -o hello hello.c && readelf -wi hello | column
```

<table>
<thead>
<tr>
<th><code>&lt;b&gt;</code>:TAG_compile_unit</th>
<th><code>&lt;7ae&gt;</code>:TAG_pointer_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_language : 1 (ANSI C)</td>
<td>AT_byte_size: 8</td>
</tr>
<tr>
<td>AT_name : hello.c</td>
<td>AT_type : <code>&lt;0x2af&gt;</code></td>
</tr>
<tr>
<td>AT_low_pc : 0x4004f4</td>
<td><code>&lt;76c&gt;</code>:TAG_subprogram</td>
</tr>
<tr>
<td>AT_high_pc : 0x400514</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>&lt;c5&gt;</code>: TAG_base_type</th>
<th><code>&lt;791&gt;</code>: TAG_formal_parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_byte_size : 4</td>
<td>AT_type : <code>&lt;0xc5&gt;</code></td>
</tr>
<tr>
<td>AT_encoding : 5 (signed)</td>
<td>AT_low_pc : 0x4004f4</td>
</tr>
<tr>
<td>AT_name : int</td>
<td>AT_high_pc : 0x400514</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>&lt;2af&gt;</code>:TAG_pointer_type</th>
<th><code>&lt;79f&gt;</code>: TAG_formal_parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_byte_size: 8</td>
<td>AT_name : argc</td>
</tr>
<tr>
<td>AT_type : <code>&lt;0x2b5&gt;</code></td>
<td>AT_type : <code>&lt;0xc5&gt;</code></td>
</tr>
<tr>
<td>AT_location : fbreg - 20</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>&lt;2b5&gt;</code>:TAG_base_type</th>
<th><code>&lt;79&gt;</code>: TAG_formal_parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_byte_size: 1</td>
<td>AT_name : argv</td>
</tr>
<tr>
<td>AT_encoding : 6 (char)</td>
<td>AT_type : <code>&lt;0x7ae&gt;</code></td>
</tr>
<tr>
<td>AT_name : char</td>
<td>AT_location : fbreg - 32</td>
</tr>
</tbody>
</table>

Virtual machines should be… – p.10/20
Functions as black boxes

Functions are *loaded, named* objects:

- extend libdl for dynamic code: `dlcreate()`, `dlbind()`, ...
- no functions “foreign” (our impl.: always use `libffi`)

```python
def fac:
    if n == 0: return 1
else: return n * fac(n−1)
```

```
0x2aaaaaf640000 <fac>:
  00: push %rbp
; -- snip
  23: callq *%rdx
; -- snip
  2a: retq
```

Virtual machines should be... – p.11/20
What have we achieved so far?

Make VMs responsible for generating entry points; then

- in-VM code is not special: can call, dlsym, …
- host VM and impl. language are “hidden” details

What’s left?

- exchanging data, sharing data
- making debugging tools work
- selection and generation of entry points… (ask me)
Accessing and sharing objects

Objects don’t “belong” to any VM. They are just memory…

… described by DWARF.

Jobs for VMs and language implementations:

- Map each language’s data types to DWARF (as usual)
- Make sense of arbitrary objects, dynamically.
  - Python: mostly easy enough (like a debugger)
  - Java: need to `java.lang.Objectify`, dynamically

Assumption: can map any pointer to a DWARF description.

- use some (fast) `malloc` instrumentation (ask me)
Java-ifying an object created by native code

- object extension
- ... dynamically
- non-contiguous
- tree-structured
- “fast” entry pts skip this

```
my_ellipse

  | maj | 1.0 |
  | min | 1.5 |
  | ctr |

  | x   | -1  |
  | y   |  8  |

vtable base

mon

extensions constructed by Java VM after receiving object
```
Wrapping up the object model

Summary: invisible VMs take on new responsibilities:

- describe objects they create; accommodate others
- register functions with libdl (→ generate entry points!)

Lots of things I haven’t covered; ask me about

- garbage collection
- dispatch structures (vtables, . . .
- reflection (but you can guess)
- extensions to DWARF
- memory infrastructure
- abstraction gaps between languages
Doing without FFI code: a very simple C API

```c
static PyObject* Buf_new(
    PyTypeObject* type, PyObject* args, PyObject* kwds) {
    BufferWrap* self;
    self = (BufferWrap*)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->b = new_buffer();
        if (self->b == NULL) {
            Py_DECREF(self);
            return NULL;
        }
    } return (PyObject*)self; }
```

– CPython wrapper

VM can do all this **dynamically**!

- … given ABI *description*

Familiar slogan: Make the dynamic case work…
What about debugging?

(gdb) bt
#0 0x0000003b7f60e4d0 in __read_nocancel () from /lib64/libpthread
#1 0x00002aaaace3f7c5 in ?? ()
#2 0x00002aaaaaa3b7b3 in ?? ()
#3 0x000000000000443064 in main (argc=1, argv=0x7fffffffd828) at

We need to fill in the question marks. Easy!

■ handily, everything is described using DWARF info
■ ... with a few extensions
■ ... just tell the debugger how to find it!
■ anecdote / contrast: LLVM JIT + gdb protocol
Why it works: the dynamism–debugging equivalence

<table>
<thead>
<tr>
<th>debugging-speak</th>
<th>runtime-speak</th>
</tr>
</thead>
<tbody>
<tr>
<td>backtrace</td>
<td>stack unwinding</td>
</tr>
<tr>
<td>state inspection</td>
<td>reflection</td>
</tr>
<tr>
<td>memory leak detection</td>
<td>garbage collection</td>
</tr>
<tr>
<td>altered execution</td>
<td>eval function</td>
</tr>
<tr>
<td>edit-and-continue</td>
<td>dynamic software update</td>
</tr>
<tr>
<td>breakpoint</td>
<td>dynamic weaving</td>
</tr>
<tr>
<td>bounds checking</td>
<td>(spatial) memory safety</td>
</tr>
</tbody>
</table>

A debuggable runtime is a dynamic runtime.

Dynamic reasoning is our fallback.

Even native code should be debuggable!
What about performance? What about correctness?

Achievable performance is an open question. However,

- our heap instrumentation is fast
- intraprocedural optimization unaffected

We can now do whole-program dynamic optimization!

- libdl is notified of optimized code
- VM supplies assumptions when generating code…

Correctly enforcing invariants is a whole-program concern!

- “guarantees” become “assume–guarantee” pairs
- e.g. “if caller guarantees $P$, I can guarantee $Q$”
- libdl is a good place to manage these too
Lots of implementation is not done yet! Some is, though.

- **libpmirror**, **DWARF foundations**: functional (but slow)
- memory helpers (**libmemtie**, **libmemtable**): similar
- extended **libdl**: proof of concept
- **dwarfpython**: can *almost* do fac!
- **parathon** (predecessor), usable subset of Python

Lots to do, but...  

...I think we can make virtual machines less obtrusive!

Thanks for listening. Any questions?