Virtual Machines Should Be Invisible

Stephen Kell
Department of Computer Science
University of Oxford
stephen.kell@cs.ox.ac.uk

Conrad Irwin *
Corpus Christi College
University of Cambridge
ctji2@cantab.net

Abstract

Current VM designs prioritise implementor freedom and performance, at the expense of other concerns of the end programmer. We motivate an alternative approach to VM design aiming to be unobtrusive in general, and prioritising two key concerns specifically: foreign function interfacing and support for runtime analysis tools (such as debuggers, profilers etc.). We describe our experiences building a Python VM in this manner, and identify some simple constraints that help enable low-overhead foreign function interfacing and direct use of native tools. We then discuss how to extend this towards a higher-performance VM suitable for Java or similar languages.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—run-time environments, debuggers, compilers

General Terms Languages

1. Introduction

Virtual machines exist to support language implementations.[1] As Cliff Click noted in last year's workshop keynote, a modern virtual machine (VM) is packed with services, covering most of what interests a language implementor: code generation, memory management, linking, loading, optimisation, profiling, and more.

End programmers care about languages and libraries, not about virtual machines. While the languages a VM supports can bring valuable abstractions to the programmer, virtual machines themselves bring costly distractions. Each class of VM brings its own set of invocation interfaces, configuration mechanisms, foreign function interfacing (FFI) conventions, and suite of tools for debugging and profiling. Developers are familiar with VMs because their peculiarities are entangled with practicalities of various languages. Ideally, however, VMs would be invisible: they would exist as libraries silently supporting the languages and libraries required by programmers, as unobtrusively as possible.

In this paper we argue that VMs may be made far less obtrusive than they currently are. Our conception of an "invisible" VM is neither fully achievable nor precisely defined, but is intended to draw attention to a general phenomenon: that the concerns of end programmers fail to align with those of implementors. Implementors are usually motivated to build a system which executes a single language (or a suite of benchmarks) as fast as possible. By contrast, end programmers want a system which is *fast enough* (which varies according to deployment scenario), which lets them analyse their code using familiar and powerful tools, and which lets them re-use whatever existing code will shorten their task.

For most of this paper we focus on two specific issues where current VMs are especially obtrusive: foreign function interfacing and run-time tool support. By the latter, we refer to debuggers, profilers, race detectors and similar dynamic analysis tools. For the former problem, our goal is to give programmers the ability to treat language as a *per-function implementation choice*, with negligible interfacing effort in most common cases. For the latter, our goal is a design that bestows on existing *whole-program analysis tools*, given only minor modifications, a first-class understanding of code running on our VMs.

Our approach is to consider a minimal set of "reasonable constraints" on VM authors, in the form of conventions and skeleton structures, that can allow cooperation across native–VM and VM–VM boundaries. This includes sharing of code, data and metadata, and a shared metamodel. Where possible, we "embrace and extend" existing facilities in operating systems and native-code compilers in preference to reinventing them at the VM level. By contrast, current designs have arisen from giving implementors free rein to reinvent, customise and optimise.

---

* Now with Rapportive, Inc.

[1] We consider this one of two necessary properties of VMs. The other is that they define some representation of executable code.

Specifically, this paper presents the following contributions.

- We describe our efforts implementing the Python language in invisible fashion, detailing various techniques for minimising FFI overhead and cooperating with debugging tools.
- We generalise these experiences to identify three design invariants: supporting linkage, data representation, and runtime metadata. Together these allow VMs to share a core object model and descriptive framework with native code, while retaining freedom to support diverse source languages, intermediate representations, and code generation.
- We discuss the evolution of our design towards *whole-program dynamic optimisation*, arguing that the wealth of code transformations within modern VMs should be broken out into a system-wide service integrated into the dynamic linker.

2. Building an invisible Python

Predating the grandiose ideal of invisible VMs, our original goal was to build an implementation of our chosen dynamic language, Python, which could be used to write scripts against native libraries interactively and dynamically. Whereas conventional wisdom was that interfacing with native libraries required glue code, generated either from annotated header files (the approach of Swig [1]) or metaprogramming (as with other tools such as the approach of Boost.Python), we considered how to make our implementation less obtrusive by dispensing with this step.

2.1 Parathon

Our insight was that compiled-generated debugging information necessarily offers descriptions of native libraries' interfaces at run time. The burden of interpreting native objects could be shouldered by the interpreter itself, dynamically interpreting this information much like a debugger, rather than by ahead-of-time glue coding. We adopted the DWARF format [6], *de facto* standard on Unix platforms, and include a brief introduction as an Appendix.

The result was Parathon, an implementation of a usable subset of Python which understands two kinds of object: those it created, described by an internal metamodel, and those created by native code, described by DWARF. Garbage was collected by the conservative Boehm collector [2], coexisting well with the C library's heap.

Numerous limitations remained: a Pythonic rendering of functions' output parameters was impossible without extra annotation; lists and arrays remained largely incompatible; Pythonic structural treatment of objects was conspicuously unsupported when making native calls. However, Parathon was sufficient to prove the concept of supporting native Python coding using debugging information. Fig. 1 show a simple C API that became an early test-case, and the CPython wrapper code that would ordinarily be used. In Parathon, our interpreter performs these operations, or their analogues, without any such direction, using only the DWARF information: a DWARF database replaces explicitly managed type objects (comment 1); underlying functions are called through libffi[2] (2), and garbage collection makes reference count adjustments unnecessary (3). Arguments are extracted directly from the calling stack frame, and marshalling is either unnecessary or inferred by comparing DWARF types. Fig. 2 shows a sample session, which runs without any wrapper code generation.

```
typedef struct buffer_s {          – data type definition
  char* string;
  unsigned int length;
  unsigned int used;
} Buf;
Buf* buffer_new();                 – creates an empty buffer
/* ... + more calls to get/put data into the buffer... */

/* Using CPython (but not Parathon) we would have to write: */
static PyObject* Buf_new(          – constructor function
 PyTypeObject* type, PyObject*
 args, PyObject* kwds) {
  BufferWrap* self;
  self = (BufferWrap*)type->       – allocate type object (1)
   tp_alloc(type, 0);
  if (self != NULL) {
   self->b = new_buffer();         – call underlying function (2)
   if (self->b == NULL) {
    Py_DECREF(self);               – adjust refcount (3)
    return NULL;
   }
  }
  return (PyObject*)self; }
}
```

Figure 1. Example API and a now-redundant CPython wrapper

However, Parathon was not entirely satisfactory. There was no way to debug code at the Python source level. Backtraces did not exhibit the Python call stack. Internally, a lot of complexity derived from the split between the two kinds of object. Passing callbacks to native code involved allocating closures generated by libffi, but it was not clear when these could be deallocated. Native objects were not first-class: for example, Python-style dynamic field insertions or removals could not be performed on them. To build a proper Python without huge escalation in complexity, a more uniform approach was required.

2.2 Towards DwarfPython

Our next insight was that the same DWARF metamodel used to describe native code and its data could also de-

[2] http://sourceware.org/libffi/

```
>>> import c                          – ensure libc (+DWARF) loaded
>>> s = stat()                        – construct a stat object
>>> stat("/etc/passwd", s)            – call through libffi
>>> print s
{ .st_dev = 42, ... (snipped) }       – access fields using DWARF
>>> def bye():                        – defining a Python function
...     print "Goodbye, world!"
...
>>> atexit(bye)                       – construct libffi closure
>>> import m                          – import another library
>>> print log2(s.st_size)             – call some more functions
10.6465587102
>>> exit(0)
Goodbye, world!
```

Figure 2. A trivial Parathon session

```
// original Parathon version -- "standard design" ↘
ParathonValue* FunctionCall::evaluate(ParathonContext& c)
{   return call_function (this->base_phrase->evaluate(c),
                          this->parameter_list->asArgs(c)); }

// less obtrusive DwarfPython version
val FunctionCall::evaluate() // ← context is the call stack
{   return call_function (this->base_phrase->evaluate(),
                          this->parameter_list->asArgs()); }
```

Figure 3. Using process context as interpreter context

scribe Python code and its data. Moreover, the dynamism of the Python language could be supported modern operating systems' and compilers' existing debugging infrastructure. (Any gaps in this would also be weaknesses encountered during debugging, hence worth fixing *within* DWARF). In other words, Python's main distinction is not its machine model but its language semantics, and this can be isolated within our Python *interpreter*, where it is invisible to other code in the process—Python becomes an implementation detail that can be hidden inside a function's implementation.

One illustration of the shift from Parathon to DwarfPython is in its notion of execution context. Like many interpreters, Parathon threads an environment and other shared state through its internal calls, by a ParathonContext* pointer. In DwarfPython, there is no such environment; to a first approximation, "the process context is the context".[3] The local name environment is discovered by examining the stack, looking up the DWARF information for the current frame, and discovering the bindings recorded for frames of this type. Fig. 3 illustrates this contrast.

Another key difference is our notion of data. In Parathon, we had a class ParathonValue representing all objects in the program, comprising 9 fields and 25 methods, and ParathonValue* pointers were ubiquitous. When modifying Parathon towards DwarfPython, one of our first changes replaced this with typedef void ParathonValue;—instead of defining our own notion of object, now an "object" is simply the referent of any pointer, and we rely on run-time availability of debugging information to support interaction with these objects.

A key property of the DWARF metamodel is its inherent flexibility, arising from the need to accommodate diverse compilers and peculiar architectures. This allows it to accommodate quirky structures, such as noncontiguous objects and functions with multiple start addresses, which turn out to prove useful. Most importantly, however, DWARF is understood by debuggers and other tools, so by maintaining a dynamic metamodel of our program as it executes, these tools are able to understand our program's state with only minor modifications.

2.3 Unifying object models (and metamodels)

A key property of our design is that it unifies the "native" object model with that adopted by a VM (DwarfPython in this case). It does so using several techniques, and these also contribute towards the debuggability of DwarfPython using native tools.

Native entry points All functions defined in Python have one or more *native entry points* generated for them. This makes them indistinguishable from native functions in a backtrace (assuming that symbols can be located—we discuss this in §3.2). In fact, *all calls* made by our interpreter, regardless of target, are implemented the same way: using libffi to call a native entry point.

Heap instrumentation We instrument the C library's allocator to record the allocation site of each heap block. Using heuristics, we map this to the DWARF type allocated by a particular site.[4] This is sufficient to recover a precise DWARF description of dynamically allocated objects, without relying on imprecise static type information. For objects allocated by Python, type information (not the allocation site) is stored directly in the heap metadata, but treatment is otherwise similar.

Tree-structured object storage Python has an atomic notion of objects, where substructure is pushed out into the heap using references to other objects. By contrast, the native world, exemplified by C and C++, adopts a more general model where objects are tree-structured: they may be contained recursively within another object. We unify these models by considering a tree-structured object to contain implicit references to its subobjects. In languages with a Python-like flat object model, these fields have the semantics (but not the representation) of read-only pointers to the contained objects. This means our Python implementation must hide the distinction between these "implicit pointers" and the usual kind. Fig. 4 illustrates the two views.

[3] The only significant exception is the list of top-level imported namespaces.

[4] These could more properly be implemented as an extension to compiler-generated DWARF information, perhaps DW_TAG_allocation_site, recording the source-level type allocated by a particular call in the text. This would also allow debuggers to perform dynamic type identification.

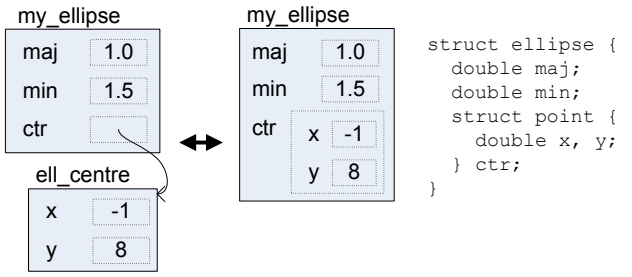

Figure 4. Viewing tree-structured objects as heaps

2.4 Making it dynamic

Debugging implements read-only dynamism over native objects, but *mutability* of those objects is lacking. (This is essentially the same problem as both edit-and-continue debugging and dynamic software update.) DwarfPython uses some additional techniques and runtime infrastructure to fill this gap. These aspects are a work-in-progress. (Henceforth in the paper we do not claim a working implementation of systems we describe, unless stated otherwise.)

Dynamic* DWARF *information Just as dynamic and reflective languages keep a *mutable* model of their own objects' structures, so DwarfPython keeps a mutable database of debugging information. To accommodate dynamic code definition, we require a protocol much like that for notifying debuggers of code generated by a JIT compiler[5], notifying the debugger of accommodate dynamic updates to the available metadata. We have developed a cleaner approach to this than current protocols, and describe it in §3.2. To allow per-object layout changes, such as field additions and removals, data types are treated in a copy-on-write fashion: modifications to an object's schema fork its descriptive information. This allows sharing in the common case, but allows unique objects to be treated uniformly.

Non-contiguous objects To support field addition even on native objects, we must support *tied* storage. This is separately allocated heap storage whose lifetime is linked with that of a pre-existing object. Tying to manually-managed heap objects is easily implemented by interposing on free(). For GC'd heap objects, some cooperation with the collector may be required to prevent early reclamation of parts of an object (since there is no reference from the tied-to object to later-added storage). We can tie storage to stack allocations by redirecting their on-stack return address to a special handler. Since DWARF allows object locations to be non-contiguous, the resulting object layouts remain describable.[6]

DWARF *extensions* DWARF is not always expressive enough for our needs. One example is source code locations: DWARF encodes a mapping from program counters to source code coordinates, but this is insufficient since any location in the interpreter might map to any Python source file. A small extension to DWARF solves this by effectively pushing additional arguments (in our case the current AST node pointer) into the line-number lookup key. Another extension is required to capture output parameters written through stack pointers. In general, this kind of DWARF extension (which we envisage exploiting through programmer-supplied annotations) invariably helps debugging use-cases too. For example, an extended line-number lookup assists with source-level debugging across code generators (e.g. generalising `yacc`'s use of `#line` directives), while capturing output parameters enables more meaningful "value returned" reports when stepping through a function exit.

[5] A notable example is the LLVM–gdb __jit_debug_register_code protocol, http://llvm.org/docs/DebuggingJITedCode.html

[6] Another implementation of non-contiguous objects is virtual inheritance in C++, although lacking the *gradual* and *oblivious* properties of ours.

3. Generalising the approach

DwarfPython is an ongoing effort, but seems promising enough that we may wonder whether it transfers to other settings. We consider a Java-like setting. Clearly, the same benefits of a low FFI coding overhead and native debuggability could be useful here. This raises several questions which we consider in this section.

- What are the principles underlying the approach?
- What generic shared infrastructure is required?
- How can we deal with the constrained object models offered by Java-like languages, e.g. the requirement that each object implements a monitor and a suite of virtual calls? Can we share the resulting objects across VM–VM boundaries? What happens to statically-enforced invariants on such objects?
- What are the implications for garbage collection? Can we still obviate the need for FFI code in the presence of higher-performance, less conservative garbage collection?
- What are the implications for traditional (dynamic) optimisations done by VMs? Can we optimise code across VM–VM and VM–native boundaries?

3.1 Principles

In essence, the whole design of DwarfPython rests on a few simple invariants.

Invariant 1. There is a shared concept of functions.

This is embodied in the fact that all functions have at least one *native entry point*. Functions are *named*, belong to a *loaded* module, and may have multiple entry points corresponding to alternative calling conventions (such as C versus Pascal versus fastcall). There is no separate concept of "foreign" functions.

(Confusingly, "foreign" and "native" are often used synonymously. We will use "native" to mean code compiled

ahead-of-time to the host architecture, and "foreign" as a relative term: to a given VM, both native code and *other* VMs' code are foreign.)

Multiple entry points may arise not only from alternative calling conventions, but alternative signatures (such as pass-by-reference or pass-by-value of a given argument) and contracts (such as "arg0 is not null" or "arg1 points to at least a Widget"). Multiple signatures and contracts accommodate differing expectations of diverse callers. Dynamic code naturally accepts arguments by reference, using this run-time indirection to dynamically discover the concrete objects pointed to, and assuming minimal precondition (instead raising exceptions dynamically when errors occur). Static-typed and/or optimised callers, by contrast, may wish to pass arguments immediately on the stack (for speed) and to call through a faster path which elides dynamic checks on the strength of static reasoning (such as Java-style bytecode verification enforcing type bounds on particular arguments). Our approach relies on dynamically generating distinct entry points to suit such diverse callers.

Invariant 2. There is a shared descriptive metamodel spanning native code and all VMs.

This is embodied in our pervasive use of DWARF, and is necessary for tool support to span VMs and native code. It is also an enabler of the final invariant.

Invariant 3. An implementation of a particular language on a particular VM will define mappings between its data types and their representations in the common DWARF-based metamodel.

This is an obligation on language implementations, in order to preserve the usefulness of a shared metamodel. In the Python case, the mapping is straightforward, since essentially any native object may be interpreted as a Python object (modulo the nontrivial treatment of tree-structured objects). We must consider how to apply our approach to more constrained scenarios, e.g. in Java where java.lang.Object brings certain requirements.

3.2 Shared infrastructure

Each of our invariants is maintained by some piece of run-time infrastructure. Encouragingly, each piece generalises from some familiar infrastructure.

Dynamic loader The first invariant entails a run-time service for tracking what code is loaded. This already exists; it is the dynamic loader provided (essentially) by the operating system, such as libdl on Unix platforms. We extend this in the same spirit as other extensions, such as dlvsym() (which adds symbol versioning on GNU and Solaris systems). Our dlcreate(), dldestroy() and dlbind() calls allow guest VMs to dynamically manage named "objects" containing entry points. We also define a four-argument dldsym() analogous to dlsym() but providing also for a token describing calling convention and signature requirements, and for multiple namespaces. This extended dynamic loader obsoletes the ad-hoc protocols for registering dynamically generated code as described in §2.4, since debuggers already track dynamic changes to the link map on dlopen() and dlclose() events; our extensions generalise this support in minimally invasive fashion.[7] We have a prototype of libld for GNU/Linux which can create new objects with a fixed-size text segment and dynamically populate them (using Linux-specific libdl options). This is sufficient for backtraces to show symbols for dynamic code. (A full implementation would lift the fixed-size constraint, likely requiring a modified ld.so.)

Metadata interface We have described DwarfPython's use of heap instrumentation and run-time debugging information to understand the running program (§2.4). Our core interface for this is implemented by a library libpmirror, which we had already created for an earlier project. As its name suggests, this library conforms (mostly) to the design principles of *mirrors* [4], but reflects a whole process rather than a single VM. It is separately encapsulated from the process it describes; like DWARF debugging information generally, is *stratified* in that it may be omitted from processes not requiring it; and inherits the DWARF metamodel's fairly direct structural correspondence with the code it models. (This means libpmirror is a cross-language reflection facility, although predictably, it only unifies multiple languages to the extent that DWARF does, which is limited—see the Appendix.)

Memory infrastructure For tracking heap metadata, we have implemented a fast associative data structure called a *memtable*, which resembles a hash table but uses a *very large* linear region of lazy-committed virtual address space, rather than an array indexed by low-order hash bits, as its primary look-up. This exploits underlying virtual memory hardware's implementation of sparse, clustered address-keyed mappings; it is both more space-efficient and faster than a hash table in our experience.[8] Entries are chained by threading a list through heap blocks (but could be kept less invasively in a separate shadow heap, or more efficiently in reclaimed malloc header space). Chains are short since each lookup entry covers a small (1KB) region of address space. Memtables are also used for tracking *tied* storage regions; a small library libmemtie provides a runtime interface for this, and adds the necessary instrumentation to the host C library's free() call (but currently no collector cooperation, cf. §2.4).

Language implementations The third invariant is handled by the language implementations themselves. The ap-

[7] We note that standard library interfaces to code loading, such as classloaders in Java, may fulfil three distinct functions: dynamic loading (including from network and other disparate sources), namespacing, and run-time code transformation. Only the first two of these are handled by our libdl extension; the third, being a metaprogramming facility, is best handled in VMs' code generation subsystems.

[8] We hope to detail this experience in a future paper or technical report.

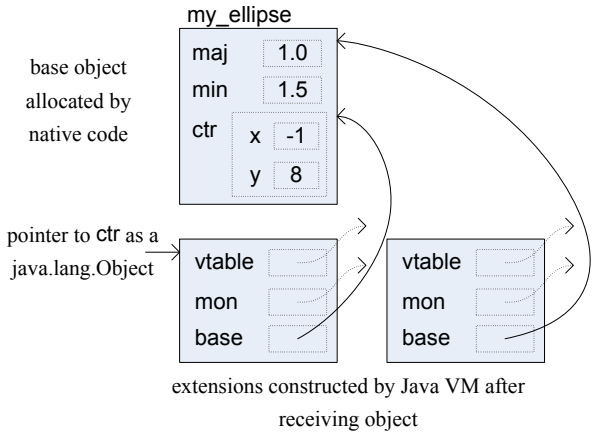

Figure 5. Language implementations may extend objects

proach of mapping languages' data-types to and from the shared metamodel is what allows separate VMs to share data. Rather than marshalling data between separate objects, as done by traditional FFI wrapper code, we can use support for non-contiguous objects to dynamically extend objects with additional data required to satisfy per-language requirements. We consider this in more detail in the next subsection.

3.3 Object layout, and other constraints

In contrast to Python, languages such as Java place specific requirements on objects: every object owns a monitor and holds a vtable pointer (or other means to dispatch a particular set of calls). It is the language implementation's responsibility to resolve these requirements by generating a hybrid layout. As with field addition in Python, these may be noncontiguous by exploiting tied storage. This results in object layouts akin to C++ virtual inheritance, but constructed lazily.[9] Fig. 5 illustrates this for our ellipse data-type being made Java-accessible. Two logical Object instances result, because of the internal tree structure, but we consider these one single non-contiguous tree-structured object. This extension is bootstrapped by the entry point of "most liberal" contract (§3.1): objects received through this code path are dynamically checked and extended as necessary, whereas a more restricted entry point assumes that this has already happened.

Since Java has a nominal typing, interfaces for foreign objects will have to be generated at compile-time. This may be done transparently from the user, given a Java compiler which can locate and load debugging information and map it to Java classes and interfaces, which is intended by Invariant 3.

[9] Since earlier parts of the structure reserve no space for forward pointers, "forward" navigation between non-contiguous parts of an object can be supported by associative look-up through a memtable, the same structure used for heap metadata.

Language implementations are also concerned with selecting which function to call, i.e. with dispatch. Dispatch occurs through data structures; we consider these structures logically part of an object layout. Moreover, their contents may logically be defined by queries over the DWARF metamodel. For example, "the vtable for class C contains all functions declared lexically within C, left-merged with like-signature methods in inherited classes, transitively, excluding methods with the final attribute". Although this elides some details (e.g. allocation of vtable slots), in general such queries can embody the overriding rules of a particular language, while subtly separating them from the data structure's core definition. Imagine that the debugging information for our ellipse data type lexically includes some nonvirtual C++ method declarations. Our Java query would populate a vtable with these methods, whereas in C++ these calls would be early-bound. (This seems reasonable in preserving the dynamism trade-offs of different languages, but arguably weakens encapsulation by risking misuse of the original ellipse implementation. We would welcome discussion of this issue in the workshop.)

We discussed the illusion of internal references within tree-structured objects in §2.3. In Python this entailed runtime overhead to distinguish an implicit pointer from a stored field. In languages with nominal subtyping it incurs no such overhead because the distinction is apparent statically in the defining type's layout.

3.4 FFI coding and garbage collection

As in DwarfPython, our design pushes the load of foreign function interfacing away from hand-coding done by the end programmer, and towards code generation done by the VM. We believe this to be appropriate: whereas APIs such as JNI are invariably convoluted by the desire to accommodate all conceivable implementations of the VM, VM implementors are uniquely aware of their own implementations, so are best placed to bear this effort. We have considered already the construction and accessing of VM-specific data representations. The other major source of FFI code is interaction between garbage collection and foreign code.

In JNI [10], several calls exist to cooperate with moving collectors, namely calls to "get" and "release" array contents and manage long-lived references (GlobalRef) to objects. Several techniques allow relieving the programmer of this burden. In the simplest, for deployments (such as DwarfPython) where a moving collector is not used, these operations are simply redundant. In a semi-conservative approach, we may sweep a widened set of roots (e.g. including the malloc-managed heap) and only move objects having no ambiguous references (at some cost in fragmentation). Alternatively we may dynamically trap the escape of these pointers into imprecise roots, by memory-protecting these regions before calling out to native code. To optimise this, we may accept annotations (perhaps derived by analysis) that a given function saves no pointers, then omit memory protection on

such calls. Without experimental results we cannot propose a definitive technique; we are arguing that some combination of these techniques is likely to allow VMs to shoulder the burden at reasonable cost. (If this seems unpalatable, we remind the reader that this is, after all, the spirit of garbage collection: using dynamic analyses to take the place of burdensome programmer effort.)

This meshes well with generational approaches. For example, we might have a conservatively- or semiconservatively-collected heap shared with C and C++ code, but then use a single precise compacting collector for objects that have not been shared, so can still be relocated. Since foreign code is typically "distant" code, we hypothesise that objects that need to be moved into the conservative heap are probably long-lived; short-lived objects may stay in heaps that are collected precisely.

3.5 Optimisation

Most intraprocedural optimisations are unaffected by our approach because they are hidden by the implementation of a particular language. Meanwhile, most interprocedural optimisations are also unaffected (or trivially *uneffected*) because JITs only optimise across code which they themselves generated. We consider two "interesting" cases as (unimplemented) thought experiments. Firstly, there are optimisations which are textually intraprocedural, but whose correctness relies on program-wide knowledge. Secondly, there are optimisations which we would like to support but currently do not: those that cross VM–VM and VM–native boundaries.

Program-wide knowledge Consider devirtualization by class hierarchy analysis. This relies on whole-program knowledge (namely the value set of a vtable entry). Since these optimisations are performed on a per-call-site basis, using a particular dispatch infrastructure, they appear to be local to a VM. However, recalling our approach (§3.3) to generating dispatch structures from shared DWARF information, the queries which were used to generate these structures are open to invalidation by code loading. This forces our shared dynamic loading infrastructure (§3.2) to get involved: queries must be persistent, and when their results are affected by code loading events, this should trigger reoptimisation.

Whole-program dynamic optimisation In last year's keynote, Cliff Click observed that profile-guided optimization in ahead-of-time compilers is trapped in a cycle of underuse and immaturity. In stark contrast, many JVMs contain a wealth of complex dynamic compilation techniques which are continuously exercised and improved. The infrastructure we have outlined is an ideal platform for breaking out this complexity into a shared service of profile-guided dynamic optimization across *whole programs*. Our dynamic loader tracks loaded code; a whole-program profiler built on this can track hot paths across multiple VMs and native code. For example, consider optimising some native code by inlining some VM-generated code which itself rests on some change-prone class hierarchy analysis. It would not normally be safe to perform this inlining because if the analysis is invalidated, the native optimiser will not be notified. Given persistent queries, we can solve this by *propagation of dependencies*: the native-code optimiser registers (with the dynamic loader) a dependency on the VM-generated code. When the analysis underlying the latter is invalidated, the native optimiser is also notified, and can replace the now-unsafe inlined code. At the heart of this approach is the separation of whole-program facts (query output, and data gathered by analysis and profiling), which are concerns managed by the shared infrastructure, from language and code generation, which remain concerns of individual VMs. To dynamically optimise native code we may build on the link-time optimisation and low-level JIT compilation pioneered by the LLVM project [8]. Finally, by discouraging premature optimisation of native code, this may help with the currently poor deoptimisation support in native toolchains—familiar to gdb users as frustrating "value optimized out" messages.

4. Related work

Many tools exist for making FFI code easier to write, but few provide direct sharing of data, and none addresses debugging the results. Swig [1] is a popular tool for generating wrappers from C APIs; Boost.Python[10] and SIP[11] are similar Python-specific tools focussing more on C++. Java Native Access[12] offers lower overheads but still requires programmers to transcribe native interfaces into Java (rather than generating them from a unified metamodel).

The GNU implementation of Java [3] integrates Java into an existing compiler infrastructure, and allows native libraries to be accessed using a much more usable interface (CNI) than Java's usual JNI. Roughly, our approach generalises this towards multiple VMs and dynamic languages.

One implementation of Scheme [12] is an interesting relative of DwarfPython. It provides a similar degree of wrapper-free integration, but no specific contribution to tool support, is C-specific, and does not support dynamism such as object schema update.

Cross-language debugging tools overlap somewhat with our goals. Blink [9] uses a controlling master process to provide a consistent interface onto multiple runtime-specific debuggers, at a cost of per-environment integration effort (since each new environment brings another debugger which must be integrated by hand). In essence, Blink embraces diversity of environments, whereas we attempt to synthesise a single underlying environment.

[10] http://www.boost.org/doc/libs/1_35_0/libs/python

[11] http://riverbankcomputing.co.uk/software/sip/

[12] http://github.com/twall/jna

There is a clear demand for cross-language and cross-VM tool support, as witnessed by extant patches to Valgrind[13], and gdb[14], machine-level Python heap profiling[15], cross-language Java debugging information[16], per-VM "providers" for the DTrace tool [5] and many others. These approaches are "point fixes" for some pairing of tool and VM, rather than direct solutions.

There is also considerable demand for sharing objects across VMs; the most relevant existing system is CoLoRS [13]. This extends stock VMs with shared objects, but does not support sharing with native code, nor unifying run-time tool support.

VMKit [7] has a similar approach of factoring managed runtimes, but instead of providing for sharing across multiple colocated VMs, considers constructing and experimenting with individual specialised VMs.

A philosophically similar approach is that of subject-oriented composition [11], which considers reconciling multiple overlapping views of the same application domain model. Our approach is essentially its analogue at machine- rather than application-level.

5. Concluding remarks

We have argued that virtual machines can and should be made far less obtrusive for end programmers to use. We have focused on FFI and debugging issues; there remain other ways in which VMs are obtrusive, especially their configuration (e.g. code search paths, resource limits, security models), which are worth rethinking. In any case, our immediate plans are: to produce a complete, optimised implementation of DwarfPython; then to apply our techniques within VMKit's `j3` JVM [7], including whole-program dynamic optimizations. We are also interested in embracing functional languages (especially with lazy evaluation, which remain difficult to debug), moving a wide range of VMs closer to the invisible ideal.

Acknowledgments

The authors acknowledge Manuel J. Simoni for an idea which seeded the work in this paper, and helpful comments from Max Bolingbroke, Nishanth Sastry and Jukka Lehtosalo. Stephen Kell was supported in part by the Oxford Martin School Institute for the Future of Computing.

References

[1] D. Beazley. Swig: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, 1996.

[2] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice & Experience*, 18(9):807–820, 1988.

[3] P. Bothner. Compiling Java with GCJ. *Linux Journal*, 2003.

[4] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 331–344. ACM, 2004.

[5] B. Cantrill. Hidden in plain sight. *ACM Queue*, 4(1):26–36, 2006.

[6] Free Standards Group. *DWARF Debugging Information Format version 3*, December 2005.

[7] N. Geoffray. *Fostering Systems Research with Managed Runtimes*. PhD thesis, Université Pierre et Marie Curie, Paris, France, September 2009.

[8] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Intl. Symp. on Code Generation and Optimization*, page 75. IEEE Computer Society, 2004.

[9] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley. Debug all your code: portable mixed-environment debugging. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 207–226. ACM, 2009.

[10] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[11] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 235–250. ACM, 1995.

[12] J. Rose and H. Muller. Integrating the Scheme and C languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1992.

[13] M. Wegiel and C. Krintz. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, SPLASH '10, pages 223–240, 2010.

[13] https://spideroak.com/code

[14] http://llvm.org/docs/DebuggingJITedCode.html

[15] http://us.pycon.org/2011/schedule/presentations/25/

[16] http://jcp.org/en/jsr/detail?id=45

A. A brief introduction to DWARF

Debugging information describes compiled code in sufficient detail to recover a source-level abstraction of a running program. This includes the ability to resolve source-level names, decode values, traverse data structures, walk the stack, and map instruction addresses back to source locations. Since it must support many machine architectures and compiler implementations, it is a rich and flexible medium. In this paper we focus on the DWARF format [6], which is common on contemporary Unix platforms.

Overall structure DWARF information is presented as a heterogeneous tree whose structure generally reflects the nesting relation in the source code. The top level records compilation units; under these are file-level definitions (e.g. classes and functions); similarly fields, formal parameters, variables, nested data types and nested functions all fall in the expected positions. Tree nodes are decorated with attributes which carry the descriptive payload: name, size and layout in memory, position in memory (relative to some implied base address such as a frame pointer or start-of-object), source code coordinates of their declaration, types of fields or variables, and so on. Although presented as a tree, the structure encodes cross-references (most commonly to data-type definitions, from definitions which instantiate them). It is therefore a graph and is frequently cyclic (e.g. for a recursive data type). Fig. 6 shows a skeleton C++ program together wiwth a schematic overview of its DWARF description.

Language features DWARF is mostly independent of language, and DWARF-emitting compilers exist for many languages including C, C++, Fortran, Objective-C, Java and others. Compilation units are tagged with their originating language. Most obvious deduplications among languages have been effected (e.g. `structure_type` includes C `structs`, Pascal Records and so on), but there is no deeper unification (e.g. `interface_type` is currently particular to Java-generated code).

Location descriptions DWARF is particularly flexible in mapping objects (including arguments, local variables, fields within objects, etc.) to memory locations. This is done with "location expressions" defined abstractly in terms of a stack machine. The expressions can source values from the running program (most typically register contents) and do arbitrary computations on them. Our implementation relies heavily on these expressions, notably to support objects split across multiple storage locations.

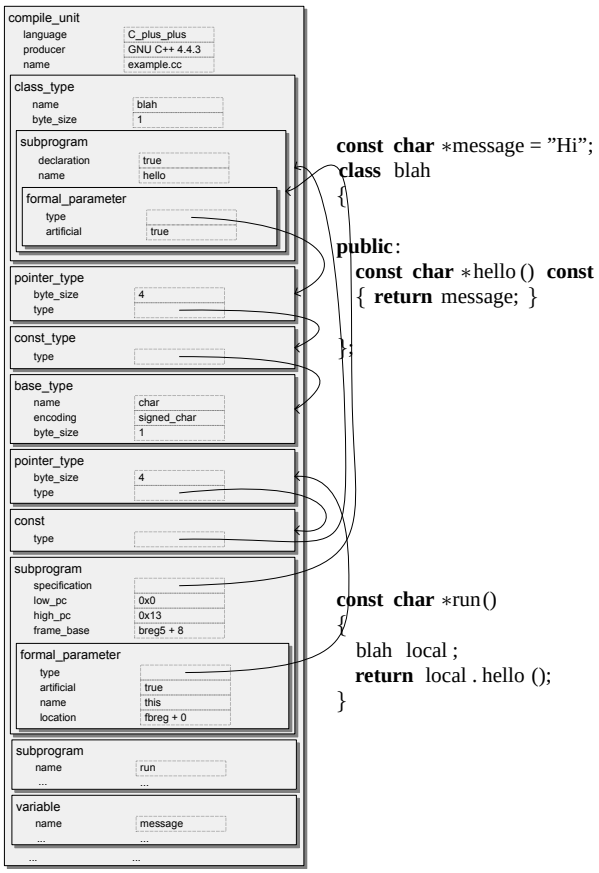

Figure 6. Simple C++ code and schematic DWARF description