The Inevitable Death of VMs: A Progress Report

Stephen Kell
University of Cambridge
Cambridge, United Kingdom
stephen.kell@cl.cam.ac.uk

ABSTRACT

Language virtual machines (VMs), as implementation artifacts, are characterised by hard boundaries which limit their conduciveness to language interoperability, whole-system tooling, and other interactions with the 'world outside'. Since the VM paradigm emerged, it has become increasingly clear that no single language or VM can succeed to the exclusion of others. This motivates a different approach in which languages are no longer implemented as VMs per se, but as participants in certain shared system-wide protocols, shared across diverse collection of languages and constituting a more porous boundary. One means of achieving such a shift is to evolve the underlying infrastructure from an essentially Unix-like environment to one that incorporates VM-like services, including memory management primitives, as a core protocol shared between many language implementations. Ongoing work towards these goals within the liballocs runtime is discussed, specifically concerning pointer identification, process-wide garbage collection, and speculative optimisations.

CCS CONCEPTS

• **Software and its engineering** → *Operating systems*; **Interoperability**; **Compilers**;

KEYWORDS

Unix, virtual machines, debugging, garbage collection, linking

ACM Reference Format:
Stephen Kell. 2018. The Inevitable Death of VMs: A Progress Report. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3191697.3191728

1 INTRODUCTION

The paradigm of language virtual machines has given us many things: high performance for even highly dynamic languages; convenient and fast garbage collection; and reliable high-level tooling. All are enabled by the internal uniformity of a virtual machine. Here the 'virtual machine' is not only a specification artifact, used to define a platform or a language, but is also how the implementation is designed and 'packaged', as a more-or-less freestanding box. But this paradigm also takes from us: through removing access to certain abstractions offered by the host system, supporting only one or a subset of the languages of interest, and providing only prescriptive tool interfaces, it create barriers and external needs which must be worked around. These workarounds are different for each VM (e.g. different FFIs), and often must be reinvented or re-learned by each end programmer unfortunate enough to need them. For example, custom tooling on the JVM is surprisingly commonplace and is usually done via onerous bytecode manipulation. Other VMs offer other mechanisms, many of them not even language-standard, but rather specific to a given implementation.

This paradigm is limiting. It made sense in the early days, of Lisp and Smalltalk, when the creator of a new language could earnestly believe that it had a chance of fully and quickly obsoleting what came before. Those days are long gone—yet the insular, self-contained implementation paradigm remains. As I have argued previously, we should instead adopt different principles in how we implement high-level languages, adopting a pluralist outlook and an attitude not unlike what Noble and Biddle [5] have called 'postmodern'. Specifically, we should aspire to package language implementations in a way that renounces 'one true VM', instead allowing first-class interoperability with the host environment (perhaps at modest drop in performance), the same interoperability with other VMs past and present, and tool support which 'sees across' these boundaries. One basis for this would be a common substrate that exposes high-level VM-like primitives sufficient to allow rich communication, including data sharing and cross-calling, between multiple VMs and host-native code. Such a substrate might be realised by extending a pre-existing platform that is widely used and embodies a pluralist attitude to languages and tools. Such a platform does exist: it is the Unix(-like) process.

This shift in principles does not necessarily mean throwing away what we have built. One appeal of adopting Unix as a base is that most VMs already target Unix-like abstractions. The shift that is necessary is therefore one of repositioning, or what I call 'retrofitting', so that the VM can make use of the services of the extended substrate, beyond what Unix currently offers. This would partly replace hand-rolled code with calls to the substrate, and partly add descriptive information and . The working hypothesis is that only small fraction of code must be changed, and that this code is mostly quite low-lying (e.g. concerning memory allocation from the operating system); compilation, object layout, or even the principal garbage collector (that on whose performance the language implementation critically depends) need not be much affected. As such, the goal is not to 'add one more' VM or design to the pile, but to embrace and underpin a broad collection of co-existing systems.

In what follows I will first recap and elaborate this position and previous work, then outline some recent or ongoing technical work on further challenges. Three particular challenges of interest are pointer identification, process-wide garbage collection (GC), and speculative optimisations.

<Programming'18> Companion, April 9–12, 2018, Nice, France

ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3191728

2 EXTENDING LIBALLOCS

The vehicle for this work is my liballocs runtime, first described at Onward! 2015 [3]. The concept of liballocs is to 'embrace and extend' existing Unix APIs towards a unified meta-level interface. This interface admits a diversity of implementations which, collectively, reach across the entirety of a Unix process—spanning not only VM-hosted code but also unsafe C, C++ or even assembly code, and all data created by such code. This interface provides query operations (get type, get bounds, etc), mutations (setting type information and/or other metadata) and allocation management operations (move, resize, etc). The key idea is to view the process address space not as a flat collection of memory mappings but more generally as a hierarchical collection of *allocations*, spanning from the large (mappings) down to the small (individual objects, each with type information). Allocations are managed by reified *allocators* (stack, heap, etc); each allocator implements the shared meta-level protocol in its own way. Type information, in a form derived from native debugging information is capable of describing a very large variety of object layouts, without precluding certain VMs (or allocators, or language implementations) primarily working with their own internal format, as currently. The fixed overheads of maintaining this information for conventional C heap, stack and static allocators are close to the noise.

Three particular areas of interest are as follows.

Pointer identification. A classical distinction between VMs and lower-level execution environments is that VMs usually 'know where the pointers are'. This design property bootstraps not only precise garbage collection, but also object motion more generally, as well as feats of dynamic compilation such as on-stack replacement, and 'hot patching' services (which, in general, require resizing and reallocation of objects). Identifying pointers reliably across a whole process appears surprisingly feasible, albeit with challenges remaining. Ongoing work is progressing this on three fronts: for static objects, the use of link-time *relocation records* for pointer identification even at run time; for the stack, combining compiler-generated debug info (for locals) and frame information (describing saved registers) to eliminate gaps and ambiguities in stack metadata; and for dynamically created *pointer-derived integers*, such as inter-object offsets computed in unsafe code (lightly instrumented to catch this), generalising slightly the previous notion of relocation records in order to record these dynamically as these integers are created and stored. A theme here is to exploit the surprisingly close correspondence between 'moving' *garbage collection* of objects (at run time) and 'relocation'-based *linking* of code and data sections; as I will explain, this synonymy is not a coincidence.

Steps towards process-wide garbage collection. Given reliable pointer identification, many useful applications become possible. One would be a single tracing garbage collector covering the whole process. Such a collector would not be a good 'primary' collector, since it is not being tuned to particular languages or object representations. However, it is essential to provide a key 'pluralist' facility necessary for the desired degree of interoperability: inter-language references with semantically first-class status. This entails *cross-heap* references; a process-wide precise collector could run occasionally to detect inter-heap cycles, and/or as an improve on the performance and precision of the imprecise (conservative) Boehm collector [1] in applications which currently use that. An alternative and less performance-sensitive application for proving the robustness of pointer identification is dynamic software update (hot patching). This shares with GC the need to resize and reallocate arbitrary code and data allocations, rewriting pointers as necessary. However, it is overall a somewhat more forgiving application regarding both performance and the necessary frequency of safe execution points at which it may be invoked. Current dynamic-update systems require either ABI changes or extensive user-supplied guidance. A system based on strong pointer identification could likely offer common-case upgrades without these interventions—roughly following an UpStare-style stack rewinding approach [4] but offering substantial automation. To enable initial progress towards these goals, a key observation is that since native compilers often (avoidably) compromise metadata coverage in the course of performing optimisations, tools for quantifying and diagnosing such omissions are necessary. The toolchain must therefore incorporate a feedback cycle: analyse the compiler's debugging information to identify stack slots which remain unexplained or ambiguous, generate a warning, and re-run the compiler with fewer optimisations. This effectively falls back to precisely described (but less optimised) code, while triaging the defect, so stands also to help existing debuggers such as gdb.

Speculative optimisations. A VM-like host environment should allow optimisations which speculate—such as on the class or layout of a target object [2]. In a heterogeneous system allowing the sort of first-class interoperability I have posited, it also becomes important to speculate on *the allocator* of the object being accessed, to allow code to include language-specific fast paths (e.g. a JS-to-JS object access, also speculating on a particular object map) while still permitting other cases (e.g. a JS-to-C field access, object layout looked up dynamically). The earlier paper [3, §6] presented a sketch of this *allocator affinity* idea, but no implementation. A very early implementation exists in liballocs of speculation on an object's allocator, enabling inlining of a likely code path.

3 FURTHER MATTERS FOR DISCUSSION

A key hypothesis of the liballocs design is that existing VMs may be retrofitted onto it at fairly modest effort, rather than being thrown away or substantially rewritten. A previous partial retrofitting (of V8) exists, but is challenging to maintain; therefore, input is sought on alternative candidate VMs for use as retrofitting targets.

REFERENCES

[1] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.* 18, 9 (1988), 807–820. https://doi.org/10.1002/spe.4380180902

[2] C. Chambers, D. Ungar, and E. Lee. 1989. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.* 24, 10 (1989), 49–70. https://doi.org/10.1145/74878.74884

[3] Stephen Kell. 2015. Towards a Dynamic Object Model Within Unix Processes. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 224–239. https://doi.org/10.1145/2814228.2814238

[4] Kristis Makris. 2009. *Whole-Program Dynamic Software Updating*. Ph.D. Dissertation. Arizona State University.

[5] James Noble and Robert Biddle. 2002. *Notes on postmodern programming*. Technical Report CS-TR-02-9. Victoria University of Wellington, Wellington, New Zealand.