Towards a Dynamic Object Model within Unix Processes

Stephen Kell
Computer Laboratory, University of Cambridge
Cambridge, United Kingdom
firstname.lastname@cl.cam.ac.uk

Abstract

Programmers face much complexity from the co-existence of "native" (Unix-like) and virtual machine (VM) "managed" run-time environments. Rather than having VMs replace Unix processes, we investigate whether it makes sense for the latter to "become VMs", by evolving Unix's user-level services to subsume those of VMs. We survey the (little-understood) VM-like features in modern Unix, noting common shortcomings: a lack of semantic metadata ("type information") and the inability to bind from objects "back" to their metadata. We describe the design and implementation of a system, liballocs, which adds these capabilities in a highly compatible way, and explore its consequences.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.4.7 [*Operating Systems*]: Organization and Design; D.2.12 [*Software Engineering*]: Interoperability

Keywords Unix, virtual machines, reflection, debugging

1. Introduction

Software today suffers from *fragmentation*: it comes tied to specific programming languages, libraries and infrastructures, creating a proliferation of isolated silos. One divergence spawning particularly many silos is that between Unix-like[1] "native" and virtual machine "managed" runtimes. By "virtual machine" we are referring to *language virtual machines*, not machines running on a hypervisor.

Historically, the earliest Lisp and Smalltalk virtual machines (VMs) were conceived to eventually supplant Unix-like operating systems. For good or ill, this hasn't happened: their descendents today (including Java VMs) nearly always occur *within* a Unix-like process. Although VMs' internal uniformity can simplify many tasks, interoperating with the wider world (including other VMs) remains an inevitable necessity—yet is a second-class concern to VM implementers. The apparent abstraction gap between VMs and the Unix "common denominator" leaves a wealth of programming tasks addressable only by manual or ad-hoc means: sharing code via brittle foreign function interfacing, sharing data (whether in memory or persistently) by manual translation between formats, and making do with a hodgepodge of per-silo tools for debugging, profiling and the like.

Instead of having VMs supplant Unix processes, is it practical to make Unix processes *become* VMs? This means evolving the basic interfaces offered by a Unix process so that they subsume those of a VM, while remaining backward-compatible. At first glance, this seems implausible: VMs offer a dynamically compiled, hardware-agnostic abstraction with late binding, whereas Unix processes seem stuck in the early 1970s: a minimally-augmented host machine, predisposed to early compilation and early binding. To turn them into VMs would amount to "adding dynamism", preserving compatibility, allowing existing VMs to be somehow *retrofitted*, and embracing the plurality of implementation alternatives that these VMs embody.

This paper argues that this is no longer too much to ask. It describes a system, liballocs, designed to achieve exactly this. Firstly, to set liballocs in context, we survey various evolutions Unix has undergone since the 1970s, including support for dynamic loading and source-level debugging. These features are not described in any textbook, and are little understood even by expert programmers (such as VM implementers), but we find they are of surprisingly powerful design—albeit fragmented, and lacking a cornerstone which we might call an *object model*. Secondly, we describe such a model and its embodiment in liballocs. This means creating an efficient whole-process implementation of reflection, which must also adhere to requirements of *compatibility* and *retrofittability*. We achieve this by *modelling* in a pure

[1] For our purposes, Windows and Mac OS are essentially Unix-like; we discuss this further in §7.

Copyright © 2015 ACM [to be supplied]. . . $15.00
DOI: http://dx.doi.org/10.1145/(to come)

sense—not unilaterally defining how objects must be, but *accommodating* pre-existing reality: the patterns by which real programs' state is organised and transformed, whether "native" or "managed".

Concretely, our contributions are the following.

Rudiments We survey modern Unix interfaces for dynamic loading, introspection and debugging, relating them to the analogous abstract concerns of VMs, and identifying how they currently fall short.

A model of allocations Having identified the concern of *whole-program metadata* as a recurring lack, we introduce liballocs as a system designed to remedy this lack, by extending the Unix process to dynamically maintain this metadata by observing the *allocation* activity of running code.

Applications and retrofitting We show how liballocs enables a selection of VM-like services, including run-time type checking, object serialization, late-bound scripting and precise debugging, applying both to native code written in multiple languages and (we anticipate) to VM-hosted code, after modest retrofitting of existing VMs.

2. Rudiments

In this section we survey various VM-like rudiments in the Unix world which, we argue, may be extended to support familiar VM services. We will be asking the following question.

> What do we need to add to Unix-like programming interfaces in order to implement VM-like services which benefit *both* existing native code *and* existing managed code (after retrofitting existing VMs)?

Our focus is not on how to precisely emulate the semantics of any one existing virtual machine, but instead on the essential abstract *services* which distinguish VMs from "unmanaged" native environments. For our purposes, these are: *dynamic loading* of code in abstract representations; *dynamic compilation* to the host machine; *reflection* facilities that recover a high-level view of a program as it executes; *debugging* interfaces that extend reflection with control interfaces; and *garbage collectors* that work by enumerating all objects and stored references in the system. We consider each of these in turn.

2.1 Dynamic Loading and Linking

VMs in both the Lisp and Smalltalk traditions are designed to be programmed *from within*. They accept code at run time, and *load* it, i.e. make it available for binding ("linking" in Unix-speak). Unix processes were conceived as being programmed from the outside, by linking a fixed selection of compiled units. But they later acquired dynamic loading. This arose from the desire to distribute software in smaller components (libraries) shareable on disk and in memory, entailing that programs be linked together on startup rather than ahead of time. This loading service was designed so that it remains available throughout execution, in the form of the API illustrated in Fig. 1. This was developed in SunOS [Gingell et al. 1987] before becoming quasi-standardised in the late 1980s along with the ELF object file format [AT&T 1990].

```
// dynamically load some code
void *handle = dlopen("/path/to/lib.so", RTLD_NOW);
// bind to some definitions it contains
float (*f)(int) = dlsym(handle, "myfunc");
float *v        = dlsym(Handle, "myvar");
// use the bindings
*v = f(42); // call myfunc, store result in myvar

// "backwards lookup" on the same definitions
Dl_info info; dladdr(f, &info);
printf("Object called %s, in module %s\n",
   info.dli_sname, info.dli_fname);
//   "Object called myfunc, in module /path/to/lib.so"
```

Figure 1. Illustrative use of the Unix dynamic loading API

Dynamic loading necessarily requires dynamic binding. Two kinds of dynamic binding are evident within the libdl API. One, which we call "forwards", accepts a meta-level query such as a file or module name and returns a base-level object satisfying that query—as with dlopen() (loading modules by name) and dlsym() (retrieving code or data by name). Another form of late binding works in the "backwards" direction, interrogating a base-level object to recover meta-information. We see this in the dladdr() call, which maps from objects' memory addresses to their metadata. It is rudimentary because only "static" objects such as functions or global variables are queryable, and only limited metadata is available. (Of course, "static" is a misnomer, since we just established that these objects can be loaded and unloaded dynamically.) To become more VM-like, we would like all units of program state to be queryable, and to have more semantic metadata—perhaps some "type"-like notion.

2.2 Dynamic Compilation

VMs typically load code in a fairly abstract representation, such as a bytecode which lightly concretises some source language. By contrast, Unix binaries contain native instructions. These instructions are, nevertheless, themselves lightly abstracted: *memory* is abstracted from the machine's fixed address space to a collection of smaller spaces addressed symbolically (sections or segments); code and data representations are abstracted by *relocation records*, invoking a repertoire of patch-like recipes for forming address bindings at load time; and *metadata* about definitions and uses is included in the form of defined and undefined symbols. On modern Unices this metadata includes not only name information (required by the dynamic binding we saw in Fig. 1), but also namespacing and visibility information, dependencies, versioning, constructor and destructor logic, and so on. These have evolved in a bottom-up fashion characteristic of Unix: classic Unix binaries included symbol metadata only, but the arrival of shared libraries necessitated other metadata, such as version information. To make Unix processes subsume VMs, we must ask how to push these capabilities a level higher, without disrupting backwards compatibility (either binary or source).

What about accepting code in alternative and/or higher-level forms? There is no obligation on a Unix loader to support only binaries targeting the host machine. Dynamic translation between instruction sets is already done by at least one binary loader (qemu-user[2], pluggable into Linux or FreeBSD), while Nethercote and Seward [2007] noted that Valgrind's techniques are capable of the same. Although both of these work at whole-process granularity, applying similar translation at per-library or finer granularity is no less feasible. When done on a portable bytecode rather than a machine language, we call this a JIT compiler. We will describe a working system for integrating VM-style JIT compilers with a dynamic ELF loader in §6.1.

One gap stands out: Unix has not yet evolved anything strongly resembling "type information", either attaching to code (such as method signatures) or to data (such as object classes). This makes it unclear how to implement VM-style load-time checks such as Java bytecode verification [Lindholm and Yellin 1999], or many dynamic optimisations which consume type information, like inline caching [Chambers et al. 1989]. Moreover, many run-time services (e.g. garbage collection, serialization) rely on "backwards lookup" from objects to metadata, for which Unix has no general mechanism. As we noted, dladdr() does not handle stack or heap allocations; it is also typically too slow for critical-path use. As we will continue to see, this is a recurring gap, and one which we will later fill (§3).

2.3 Reflection

It's clearly possible to do *some* reflection on stack and heap data in Unix, because programs can be debugged. We split our consideration of debugging into firstly general *reflection* primitives, and secondly the remaining aspects concerning *control* (such as starting, stopping or modifying execution). We define reflection as *metaprogramming against a running program*, and "introspection" as *self-reflection*.[3]

As we'd expect, reflection in Unix evolved bottom-up. Machine-level debugging has been supported since the earliest versions of Unix. Source-level reflection was also an early addition and is now extensively supported, using a division of responsibilities which departs considerably from most VMs' reflection or debugging systems. Nevertheless, Unix-style reflection adheres remarkably tightly to the "mirrors" design principles of Bracha and Ungar [2004]—even though these were conceived with VMs in mind. In keeping with the latter, we will talk about "reflection" generically, even though the mechanisms we describe are associated primarily with debugging.

```
$ cc -g -o hello hello.c && readelf -wi hello | column
<b>:TAG_compile_unit          <be>:TAG_pointer_type
     AT_language  : 1 (ANSI C)      AT_byte_size: 8
     AT_name      : hello.c         AT_type     : <0x2af>
     AT_low_pc    : 0x4004f4  <dc>:TAG_subprogram
     AT_high_pc   : 0x400514        AT_name     : main
<85>:TAG_base_type                  AT_type     : <0xc5>
      AT_byte_size: 4               AT_low_pc   : 0x4004f4
      AT_encoding : 5 (signed)      AT_high_pc  : 0x400514
      AT_name     : int       <f0>: TAG_formal_parameter
<9f>:TAG_pointer_type                AT_name    : argc
      AT_byte_size: 8                AT_type    : <0xc5>
      AT_type     : <0x2b5>          AT_location: fbreg - 20
<a5>:TAG_base_type            <fe>: TAG_formal_parameter
      AT_byte_size: 1                AT_name    : argv
      AT_encoding : 6 (char)         AT_type    : <0x7ae>
      AT_name     : char             AT_location: fbreg - 32
```

Figure 2. DWARF information (.debug_info section) for a "Hello, world!" C program

The principles of Unix reflection are summarised as:

- require no cooperation from the reflectee, which might equally be a "live" process or a "dead" coredump;
- support *multiple reflected views*: at least source-level and assembly-level, and perhaps others;
- keep the compiler and reflecting client (≈ debugger) separate, communicating via well-defined interfaces.

"No cooperation" means the client is given direct access to the reflectee's memory and registers. Metadata generated by the assembler affords a somewhat symbolic view of these. Metadata generated by the compiler affords a source-level view. Fig. 2 shows metadata in the DWARF format [Free Standards Group 2010], standardised beginning in 1992, describing the main compilation unit in a simple "hello world" C program.

This contrasts with VM approaches to reflection, in which the reflecting client consumes the services of an in-VM reflection API and/or debug server. This is expedient: the reflection system and debug server share code in the runtime, and need not describe the compiler's implementation decisions explicitly. A VM debug server need never disclose the kind of addressing, layout and location information shown in Fig. 2. But it cannot easily support the post-mortem case, and tightly couples run-time support with compiler: we cannot use one vendor's debugger to debug code from another vendor's (in-VM) compiler. It becomes hard to implement reflection features not anticipated in the design of the reflection API or debug server command language. By contrast, metadata is open-ended and naturally decoupling. We are not the first to note this: Cargill [1986], describing his Pi debugger, remarked that "Smalltalk's tools cooperate through shared data structures… [whereas] Pi is an isolated tool in a 'toolkit environment'… interacting through explicit interfaces." In other words, the Unix approach entails inter-tool

[2] http://wiki.qemu.org/Main_Page, as retrieved on 2015/8/14

[3] This is standard, but has the confusing consequence that "reflection" includes the *non-reflexive* case.

encapsulation, hence stronger public interfaces than a single integrated virtual machine. One such interface was the /proc filesystem [Killian 1984], co-developed with Pi; another is the exchange of standard debugging metadata.

Almost twenty years after Cargill's Pi, Bracha and Ungar [2004] outlined their "mirrors" design principles for metaprogramming facilities, embracing reflection, debugging and so on, summarised as follows:

- **encapsulation**: "the ability to write metaprogramming applications that are independent of a specific metaprogramming implementation"—meaning metaprogramming interfaces should not impose undue restrictions on clients, such as reflecting only on the host program (a weakness of Java core reflection);
- **stratification**: "making it easy to eliminate reflection when it is not needed", such as on embedded platforms or in applications which happen not to use it;
- **ontological correspondence**: that metaprogramming interfaces should retain user-meaningful concepts, both structural (e.g. preserving source code features in the meta-model) and temporal (e.g. the distinction between inactive "code" and active "computation").

The Unix approach to debug-time reflection satisfies all of these principles either fully or very nearly; we discuss each in turn.

Encapsulation The encapsulation of mirrors is motivated via a hypothetical class browser tool, noting that the Java core reflection APIs bring an unwanted restriction: reflecting only the host VM, not a remote instance. This is a failure of encapsulation, not because it doesn't hide the VM's internals (it does!), but on criteria of *plurality*: clients may reflect only on one specific machine's state (the host machine's), are provided with only a single, fixed view, and only one implementation of the interface may be present in any one program. Different mirrors offering distinct meta-level views are often desirable, as alluded to by Bracha's and Ungar's mention of "a functional decomposition rather than… leaving that decision to the implementation of the objects themselves". Coexistence of different implementations of the same abstraction is a key property of object-oriented encapsulation, as noted by Cook [2009] and Aldrich [2013].

Unix reflection is very strongly encapsulated. The same client can reflect on programs generated by diverse compilers; it is easily extended to remote processes and can reflect on coredumps similarly to "live" processes. The use of metadata as the "explicit interface" means there is no need to fix on a command language, and the client is free to consume the metadata in any way it sees fit. Unix debugging information has a history of being put to diverse and unanticipated uses, such as bounds checking [Avijit et al. 2004], link-time code generation [Kell 2010] or type checking [Banavar et al. 1994].

Stratification Unix reflection is strongly stratified. This follows from the decision to avoid run-time cooperation from the reflectee (which, indeed, might be dead), and from the decoupling of compiler and runtime. Programs that are not reflected on do not suffer any time or space overhead, yet debuggers can be attached "from the outside" at any point, loading metadata from external sources as necessary.[4] In-process reflection can also be added late, via dynamic loading if necessary. In-process stack walkers are found in backtrace routines or C++ runtimes; the latter's "zero cost" exception handling design [de Dinechin 2000] is stratified in that code throwing no exceptions pays no time or space overheads. It is no coincidence that this is usually implemented with metadata also used by debuggers. This ability to "add reflection" extends even to source languages such as C which don't specify any kind of introspection interface. We will see a richer example of reflection on C code later (§3.1).

Temporal correspondence This is illustrated by the hypothetical desire to "retarget the [class browser] application to browse classes described in a source database". It refers to a distinction between "mirroring code and mirroring computation"—where "code" means *code not yet activated* (such as method definitions in source code) while "computation" means *code in execution* (such as method activations in a running program). The authors remark that having attempted to do away with this distinction, they found themselves recreating it, in the Self project's "transporter" tool. (This tool could be described as Self's linker and loader.) Unix exhibits temporal correspondence in the sense that the metamodel of Unix loader and debugger inputs (shared objects, executables, and the functions and data types they define) is separate from run-time details (function activations, data type instances, etc.). In DWARF, we find the latter are described distinctly, in terms of an embedded stack machine language (encoding mappings from machine state, such as a register, to units of source program state, such as a local variable). Consumers of DWARF which care only for static structure can ignore these attributes, and DWARF data which omits them remains well-formed.

Structural correspondence This requires that all features of source code are representable at the meta-level. DWARF models a wealth of information from source code, including lexical block structure, namespacing features, data types, module imports, various attributes, and so on. However, it does not undertake to model every feature, so arguably falls short of structural correspondence. In fact DWARF actively *abstracts away* from source, in that its metamodel deduplicates certain language features. For example, a Pascal record and a C struct are both modelled as a DWARF structure_type. Although Bracha and Ungar envisaged that distinct source languages would offer "distinct APIs", this im-

[4] Helpfully, in recent years it has become increasingly common for open-source Unix distributors to package not only binaries and source code, but also debugging information relating the two.

plies a strong dependency between reflection system and source language which is not always desirable. The conceptual distinction between record and struct is negligible, so it is more often valuable to unify them than to distinguish them. DWARF's choice, of (conservatively) unifying where this is conceptually lossless, hints at two intriguing possibilities: firstly of "translating" or *reinterpreting* the state of a Pascal program *as if it were a C program*, and secondly of reflecting semantic intent that was not expressible in the original source language. For example, in C, a Pascal-like variant is often realised as a union within a struct whose first field acts as a discriminant. C does not let us describe the field as a discriminant, but since Pascal does, DWARF metadata has features for expressing the C object's layout more precisely than C itself can. In general, Unix's pluralist approach to reflection actively enables multiple source-level views of the same objects; we revisit this in §5.

What's missing We have established that Unix's introspection support has a remarkably strong design. However, it's not clear that this debugging infrastructure, designed to run at "human speed", affords the efficiency expected of a more general reflection mechanism. For example, debugging metadata is structured so that computing the location of a named local variable is fairly cheap, but to map a stack address "back" to the variable it holds requires expensive searching and/or stack-walking operations. Later, we describe our work addressing these problems (from §3.1).

2.4 Debugging

Building a working debugger on Unix requires not only reflection, but also control of the target. This is provided by the ptrace() call, which allows one "tracer" process to attach to any other process, suspend and resume it, inspect its memory, and intercept system calls or signals. All this occurs without the knowledge or cooperation of the tracee. (This is an innovation over classic Unix designs, as noted by Killian [1984].) As before, ptrace() offers only a machine-level view, but higher-level views may be recovered using debugging metadata.

2.5 Garbage Collection

A central part of most virtual machines is a tracing garbage collector, building on several of the VM facilities we have discussed. Enumerating roots means collecting knowledge of all the loaded code's global variables, then using pointer metadata to explore the stack and heap. The most well-known approach to collection in Unix is conservative collection [Boehm and Weiser 1988], which works around the absence of precise metadata. Unfortunately, its performance is uncompetitive with more precise collectors which demand precise metadata. Unix's lack of type metadata and "backwards lookup" (from object to metadata) stymie such designs. Conversely, the availability of these would allow us to rethink this. We return to this in §6.2.

```
struct uniqtype;                                         /* type descriptor    */
struct allocator ;                                 /* heap, stack, static , etc */
uniqtype *  alloc_get_type      (void *obj );  /* what type?          */
allocator * alloc_get_allocator (void *obj );  /* heap/stack? etc     */
void *      alloc_get_site      (void *obj );  /* where allocated?    */
void *      alloc_get_base      (void *obj );  /* base address?       */
void *      alloc_get_limit     (void *obj );  /* end address?        */
Dl_info     alloc_dladdr        (void *obj );  /* dladdr-like         */
```

Figure 3. A simplified liballocs process-wide metadata API

3. A Model of Allocations

We have seen that Unix has evolved considerable support for VM-like services, but with two significant gaps: the lack of type-like metadata, and lack of a mechanism for dynamic binding "backwards" from objects to metadata. In this section we introduce liballocs, a system which addresses these shortfalls by implementing something akin to an "object model' for Unix processes. Unlike a VM's object model, we conceive liballocs with compatibility and plurality in mind: it must *model* what a large variety of existing code does, not simply provide a structure for future code to be written against.

3.1 Overview

Fig. 3 shows a simplified version of liballocs's API, in C. It maps from a relatively basic, general, machine-level view up to various kinds of metadata, and subsumes the dynamic loader's dladdr() call (which we noted offered a basic object metadata lookup function). Type metadata is modelled by reified data types, instances of struct uniqtype. We can think of this API as a protocol which different allocators implement differently, but is collectively implemented *somehow* for every allocation in the process.

Can it be implemented usefully against existing Unix code, much of which is written in permissive languages like C and C++? Can it be implemented efficiently? Can there be an adequately general model underlying struct uniqtype, liballocs's notion of "type"? Can we accommodate a full complement of VM services within this model, such that it is possible to *retrofit* existing VMs onto it?

In the remainder of this paper we present arguments and evidence answering these four questions in the affirmative. This section covers the design of liballocs; it pays careful attention to compatibility with C and C++ (§3.7), has a working prototype (§4) and we have implemented several services on top of it (§5). This answers the first pair of questions reasonably strongly. The latter pair are somewhat more speculative, but we will present various arguments and evidence already amassed in their favour (§6).

3.2 What Are Allocations?

Allocations are coherent subdivisions of the state of a running program. They are contiguous in memory and in lifetime, so constitute units of data meaningful (at some level)

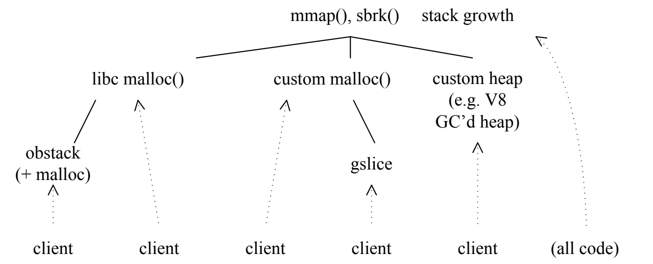


Figure 4. Allocator tree in a large C/C++/JavaScript program

in the program. Allocations have metadata, including a notion of "type", a base address and end address in memory, and the site (instruction address) where allocation occurred. This embraces and extends the dladdr() API we saw earlier: each symbol definition in a loaded object file, including functions and global variables, is a distinct allocation, but so are heap objects and stack frames. Allocations are a "common denominator" across all languages, runtimes and virtual machines. They can be identified at the machine level, as memory regions, yet we can ascribe abstract meanings or roles to them, in terms relating to the source program.

Allocations are slightly lower-level than most senses of "object", since allocations deliberately lack a notion of behaviour, such as a system of messaging or method dispatch.[5] A great variety of dispatch behaviours can be implemented on top of our liballocs interfaces, but it is left to each *client* of an allocation, i.e. each caller, to implement the dispatch semantics that they locally require; none is enforced. This separation of mechanism from policy keeps many language-specific details out of allocations; it is essential to our goals of compatibility and plurality. (That is not to say that liballocs is not useful for dynamic binding—far from it. In §5 we see how liballocs enables late binding in places where it wasn't previously available. We consider options for clients performing dynamic dispatch in §6.2.)

3.3 The Allocator Hierarchy

In a modern Unix, the state of a process consists of a memory space structured as a flat collection of *mappings*. (It also includes a register file per thread, and some kernel-side state such as the file handle table.) These top-level mappings are parcelled out to user code via some arrangement of intermediate allocations: memory pools, slabs, stacks, and so on. These are often nested, such as allocators that are themselves clients of malloc(). The leaves of this tree are the units of state specified by the end programmer: local variables ("stack"), global variables ("static") and heap objects. To maintain per-allocation metadata detailed enough to implement VM-like services, we require a general model of these nested allocations.

We can model this structure as a tree of allocators. Allocator B is a child of A if B calls A to obtain the memory that it parcels out to clients. All allocated memory in a process—whether heap, static or stack—can be attributed to some such allocator. Fig. 4 shows the tree of heap allocators that might be present in a large C/C++/Java program. At the top of the tree are the operating system mechanisms—on Unix these are mmap(), sbrk(), and the kernel-supplied stack allocator.[6] Client code can request allocations at various levels in the tree: from malloc(), or from an allocator layered under malloc(), or direct from mmap(), and so on.

3.4 Allocator Diversity

The role of liballocs is twofold: firstly, to define a standard *meta-protocol* for allocators; secondly, to separately maintain additional metadata in the cases of existing allocators which do not keep adequate metadata by themselves, such as malloc() implementations or the kernel's stack allocator. We deliberately avoid constraining allocators: they may expose whatever base-level interfaces they choose, backed by whatever implementation.

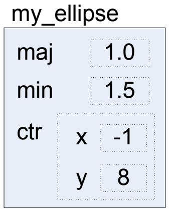
Since liballocs accepts meta-level queries about arbitrary memory addresses, it must dynamically keep track of how the address space is parcelled out to allocators, including nesting relationships, so it can delegate queries appropriately. Allocators must therefore announce themselves to liballocs, and must *either* communicate allocation events as they occur *or* implement calls by which existing allocations can be identified post-hoc, at the point of a metadata query. This communication can be achieved either by explicit code changes or by instrumentation. Helpfully, the latter is particularly feasible for allocators used by native code.

We can divide allocators into three categories: system, user-explicit and user-implicit; these correspond roughly, but not exactly, to static, heap and stack respectively. As we saw earlier, "static" allocations, such as of global variables, actually occur at load events during execution, within the dynamic loader provided by the operating system. This makes them easy to observe, by instrumenting load and unload functions. *User-explicit* allocations include those created by *procedurally abstracted* allocators in user code, such as malloc() in C or new in C++.[7] It is similarly possible to hook these allocations at run time, using link-time interposition, breakpoint-like trap instructions or lightweight binary instrumentation [Kessler 1990; Miller et al. 1995]. The remaining *user-implicit* allocations are *not* procedurally abstracted; examples include stack allocations (made simply by adjusting the stack pointer) or "new space" allocations in generational garbage collectors. We cannot efficiently trap every update to the stack pointer or bump pointer, so we require that these

[5] ... except for the "call" or "jump" primitive of the underlying machine/ABI. Code necessarily resides in allocations, just like data.

[6] The latter is instantiated by the MAP_GROWSDOWN flag to mmap(), and invoked by subsequent page faults.

[7] Note that malloc() is user code, since a program may supply its own implementation.



```
struct ellipse {
  double maj;
  double min;
  struct point {
    double x, y;
  } ctr;
}
```

```
2d: DW_TAG_structure_type
  DW_AT_name        : point
39: DW_TAG_member
  DW_AT_name        : x
  DW_AT_type        : <0x52>
  DW_AT_location: (DW_OP_plus_uconst: 0)
45: DW_TAG_member
  DW_AT_name        : y
  DW_AT_type        : <0x52>
  DW_AT_location: (DW_OP_plus_uconst: 8)
52: DW_TAG_base_type
  DW_AT_byte_size   : 8
  DW_AT_encoding    : 4 (float)
  DW_AT_name        : double
59: DW_TAG_structure_type
  DW_AT_name        : ellipse
  DW_AT_byte_size   : 32
61: DW_TAG_member
  DW_AT_name        : maj
  DW_AT_type        : <0x52>
  DW_AT_location: (DW_OP_plus_uconst: 0)
6f: DW_TAG_member
  DW_AT_name        : min
  DW_AT_type        : <0x52>
  DW_AT_location: (DW_OP_plus_uconst: 8)
7d: DW_TAG_member
  DW_AT_name        : ctr
  DW_AT_type        : <0x2d>
  DW_AT_location: (DW_OP_plus_uconst: 16)
```

Figure 5. A simple ellipse data type in C, DWARF and diagrammatically

allocators allow post-hoc identification of individual allocated regions. We describe an efficient implementation of this for stack regions in §4.3, and consider GCs in §6.2.

3.5 Type Metadata for Allocations

Most allocation metadata, such as start and end address, is semantically straightforward. Type metadata is trickier. What *are* types? In Unix's familiar bottom-up fashion, liballocs borrows an existing notion of "type" found in DWARF debugging information [Free Standards Group 2010], which is more than an adequate starting point.[8] Fig. 5 shows a simple ellipse data type in C, graphically, and in dumped DWARF. Note how it spans from machine level up to source level: it describes the size and encoding of all primitive data (here float means the machine-native floating-point encoding), while a stack machine language encodes the offset of fields from their object base. Although not shown, DWARF also records source-level structure, such as (in C++ or Java code, say) which functions are members of a class.

When an allocation occurs, how do we know what data type is being instantiated? We view this logically as part of the allocating code's debugging information. For stack and static allocations, DWARF already includes adequate information. For heap allocations, most languages offer *typed* allocation operations, such as new, so the compiler *could* output records describing what DWARF type is being allocated and at which instruction address. We call these addresses *allocation sites*. Even in "dynamically typed" languages, each source-level allocation site creates allocations of dynamically known types. (This does not exclude polymorphism, where the type is not *statically* known.) To work around the current lack of allocation site information in DWARF, we maintain our metadata separately (as described in §4.3). To deal with languages offering untyped allocation primitives, such as malloc() in C, we require source-level analysis (see §4.2). Some allocations *within allocators*, at the branches of our tree in Fig. 4, are genuinely untyped—these allocate memory held in a pool for later. Therefore, alloc_get_type() is permitted to return null.

3.6 Dynamism in Type Metadata

The role of types in liballocs is to capture semantic properties of allocations, including their substructure, what they model, and so on. Types in liballocs are not "static": they are reified at run time as uniqtype objects, much like class objects in a Smalltalk VM, and exhibit several kinds of dynamism. Most obviously, as allocations come and go, observed by liballocs, the memory at a given location changes type. Most allocators allow explicitly changing the type associated with a given allocation.

Sometimes details *within* an allocation change dynamically, such as gaining or losing fields. To this end, a single uniqtype can encode a bounded degree of internal per-allocation variability, perhaps in length (an array of varying size, say) or substructure (a mutable variant record, a union in C, or a hash-table layout in JavaScript). To encode this, a uniqtype may refer to a make_concrete() function which takes the object base address (and, optionally, some additional program context) and returns a *dynamically precise* snapshot of itself. This allows details to be decided at run time yet still be queried precisely, without spawning a distinct uniqtype for every possible case.

Over an allocation's lifetime, its type may change; this might spawn a fresh uniqtype in the case of variation not anticipated by the original.[9] Mutating a uniqtype in-place might make sense, but raises semantic questions (whether or how existing instances should be updated) and implementation questions (how to *reallocate* both the uniqtype and its instances). Currently it is not supported. A process-wide garbage collector would allow us to rethink this; we revisit this possibility later (§6.2).

3.7 Accommodating C and C++

C and C++ are archetypal languages of the Unix world, and supporting them seamlessly and compatibly is of great importance to liballocs. Does it even make sense to attach type metadata to allocations in languages such as C? Semantically, the C standard holds that only local and global variables have a fixed ("declared") type, and heap storage acquires its "effective" type from writes. However, the overwhelmingly common case is for the type to be fixed at allocation time. Therefore, for efficiency, we assign types on al-

[8] It is important—as this author has argued previously [Kell 2014]—to distinguish models of data abstraction from the logics that underlie type checkers. Both are often called "type systems", but only the former concern us here. We return to the issue of safety in §7.

[9] This occurs most commonly in dynamic languages, but can even happen in C, in the case of memcpy()ing only part of a structured object.

```
export LIBALLOCS_ALLOC_FNS="__ckd_malloc__(Zpi)p \
   __ckd_calloc__(zZpi)p __ckd_realloc__(pZpi)p __mymalloc__(Ipi)p"
export LIBALLOCS_ALLOCSZ_FNS="__ckd_calloc_2d__(iiIpi)p \
   __ckd_calloc_3d__(iiiIpi )p fe_create_2d( iiI )p"
export LIBALLOCS_FREE_FNS="__myfree__(P) ckd_free_2d(P) \
ckd_free_3d(P)"
```

Figure 6. Describing custom allocators in a large C codebase (sphinx3 from SPEC CPU2006)

location, and avoid trapping writes. We can, however, model subsequent changes of a heap block's uniqtype. This includes the case of receiving data via memcpy(); since this is procedurally abstracted it is easy to trap. The C standard also allows an effective type to be propagated either by raw characterwise copying or by memcpy(). Since real code tends to use the latter, and some compilers even turn the former into the latter, we choose not to support this.

4. Implementation

We have a prototype implementation of liballocs, currently only for x86-64 GNU/Linux machines but without particular obstacle to ports to other platforms. Work has so far focused on the core system and on accommodating native code compiled ahead of time from C and C++. The implementation consists of several parts: generic compile- and link-time tools to support allocation and data-type metadata; supplementary tools specific to C and C++; and the back-end liballocs runtime. We cover each in turn.

4.1 Tools and Compiler Wrapper

Our implementation includes various tools necessary to generate liballocs's run-time metadata. Most tools consume DWARF, but some consume program source, program binaries and/or a small amount of programmer-supplied guidance.

User allocators User-explicit allocators (§3.4) have their signatures declared to liballocs, currently using environment variables and an ad-hoc signature language. Fig. 6 shows a relatively complex example. Allocator nesting is inferred dynamically by liballocs, but users must declare *allocator wrappers* because they affect the logical *allocation site*. A common example is xmalloc(), defined to delegate to malloc() but terminating the program if it fails. The allocation site is at the caller of the wrapper, not the delegating call within the wrapper.

Type metadata DWARF is unsuitable as an in-process metadata representation. It is tightly packed on disk, but also highly redundant: each compilation unit includes its own copy of any recurring data types. We postprocess it into uniqtypes, so named because they are deduplicated. Those generated for our ellipse are shown in Fig. 7. Each uniqtype is reified at the linker level as a symbol definition, named using a simple human-readable convention. Their simple, fast in-memory representation encodes their size, substructure, and links to other *related* types, such as the types of fields, function arguments, and so on. Note how ellipse refers to point, and both refer to double. Types which are nominally and structurally identical are merged. Primitives are canonicalised according to their size and encoding (e.g. merging int with signed int). Synonyms (e.g. via typedef in C) are rendered as alias symbols. Each symbol name includes a "digest code" computed from the type definition, disambiguating distinct like-named types without requiring per-codebase namespaces. Standard Unix linker features (ELF section groups [SCO 2012] and global symbols) are used to ensure a unique definition of each data type is used at run time, even those shared between libraries and executable. This ensures that simple semantic operations, such as tests for type identity, are implemented efficiently as pointer comparisons.

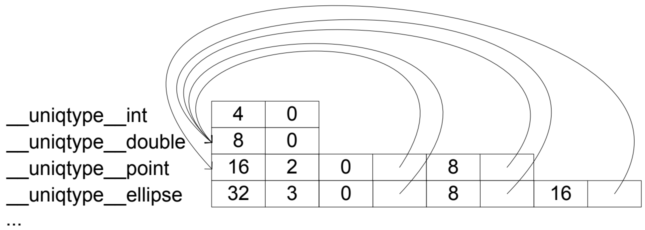


Figure 7. Our uniqtype representation (slightly simplified) of the ellipse and types it uses: pointer arcs represent containment, and the fields preceding pointers are the offsets of the contained subobject; other variants (not shown) encode arrays, functions, pointers, etc.

Allocation site metadata Since DWARF does not yet include allocation site information, we gather this separately. A binary-level analysis searches for calls to the named allocator functions (and all indirect calls); per-language source-level analyses (we outline one for C shortly) output allocation sites' source coordinates and type information. The two are combined with the help of debugging information. This allocation site metadata, along with all generated uniqtype instances, is "compiled" via C to binary metadata (per library or executable) loadable directly by liballocs.

Computing reliable stack layouts Stack frames each have a uniqtype too: although DWARF models these separately from data types, we unify them in postprocessing, yielding at least one uniqtype per function. We found that the way some on-stack allocations are recorded in DWARF information was problematic when computing stack frame layouts. Frame layouts encode the address of each on-stack argument or local variable as an offset from a logical *frame base* address. DWARF describes these offsets as generic stack-machine expressions, whose inputs may be the frame base address *or* register values. We found that the compiler occasionally describes the location of an on-stack argument or local not in terms of the frame base, but in terms of some other register which happened to be holding a stack address (say "rbp + 8"). This is fine for a debugger during execution (which can read the actual register values), but not for our ahead-of-time preprocessing. We fixed this by combining the stack machine expressions with additional stack-walking information from the .debug_frame section. The latter describes how one register's value can be reconstructed in terms of other registers' values. We use this to build a graph whose vertices are registers holding stack addresses and whose edges are fixed-offset relationships between these addresses. Deducing transitive fixed-offset relationships then becomes a graph reachability problem. This allows us to reliably identify every local and actual stored at a fixed offset from the frame base.

4.2 Supporting C and C++

To assign types to heap allocations made using malloc() and other untyped interfaces, our insight is that for allocations occurring at leaf level (§3.3), the type is invariably deducible from how the allocation is *sized*: the programmer chooses how much to allocate based on the size of the intended data type. We must therefore parse C source code and analyse its use of sizeof around calls to the all known allocation functions. For example xmalloc(n *sizeof(long)) tells us an array of long is being allocated. Indirect calls are analysed as if they might be calling *any* signature-compatible allocation function. A somewhat non-trivial analysis is necessary to handle cases where a size is computed in stages, perhaps some distance away from the malloc() call. Details of this analysis are left to a future publication, but overall it evolves the approach taken by Magpie [Rafkind et al. 2009]. We have found this approach to be remarkably robust in practice. Another useful property is that it can be implemented without recompilation of the source code.

We aspire to *compatibility*, meaning requiring no source-level changes, and no recompilation in the common case. Some awkward cases stymie the latter; one is memcpy(), mentioned earlier (§4.2); another is dynamically-sized stack allocation using alloca(), where we must intervene to explicitly notify liballocs of the allocation. This turns alloca() into a "user-explicit" on-stack allocator. C99 variable-length local arrays are also handled this way. Our C source-level passes are written using CIL [Necula et al. 2002], although an experimental Clang/LLVM front-end has also been developed.

Although our techniques do not strictly require recompilation in the common case, our allocscc compiler wrapper is a convenient way to package the various tools. Recompiling with the wrapper is often easier than going through the separate steps of acquiring source code and debugging information, picking through the source to check for absence of awkward features, and invoking the tools separately. (This would ideally be remedied by integrating liballocs more closely into package build systems, like that of Debian.) The wrapper also allows us to tweak compiler options to improve liballocs's performance, notably turning on frame pointers, and allows us to interpose on allocation calls purely at link time rather than via binary instrumentation.

Some allocators make link-time interposition difficult, even though they are procedurally abstracted. In one C codebase (bzip2) we found an allocator that is declared static and used only within the defining file, meaning there is no undefined symbol to hook. To handle awkward cases like these, we patched GNU objcopy to allow splitting a defined symbol into separate entries for the definition and use, hence allowing interposition even on these local symbols, and also made our compiler wrapper add a command-line option (-ffunction-sections) to ensure local symbols are never omitted. This could otherwise sometimes happen, if an allocator definition and sole reference were to occur in the same text section. Dynamic binary instrumentation techniques consuming DWARF would avoid these difficulties entirely, since they arise only because linker-level ELF symbols are omitted; the relevant definitions are still described in DWARF.

4.3 The liballocs Runtime

Recall (Fig. 3) that the run-time interface to liballocs concerns answering queries. Given a query such as alloc_get_type(p), some quick tests on p check it against the bounds of the current stack, the executable's mapped segments and the sbrk()-bounded heap. These are often sufficient to identify the originating allocator (static, stack or malloc()-heap). For pointers falling in none of these regions, we must resort to a more general traversal of the allocator tree, rooted at a specially crafted structure called the level-0 index. This tracks the virtual address space (VAS) at page granularity, associating page numbers (i.e. virtual addresses appropriately right-shifted) with an allocator identity and some general information. It roughly mirrors Linux's /proc/<*pid*>/maps file, with the addition of allocator information. We maintain it by interposing on memory mapping and dynamic loading calls (creating the first-level children in the tree) and also on "sub-allocations" made by user-identified allocator functions (creating deeper branch nodes in the tree, e.g. a heap backed by a malloc()-supplied chunk). All memory mappings and other non-leaf allocations are recorded in a *mapping table*, containing information about the allocator owning that area of memory. Their "mapping number" is their offset into this table.

The level-0 index's job is to make lookups into the mapping table efficient. In general, "indexes" in liballocs are *sparse*, *address-keyed* maps, and their implementations exploit *virtual memory*—virtual address space is a plentiful resource, over which Unix gives us considerable control. The level-0 index consists of a huge array, having one integer for every page in the user-accessible address space (or 2^{35} pages on current x86-64 platforms), such that if mapping number *i* includes page number *p*, l0index[*p*] stores *i*. We allocate the array with a request to the operating system to commit its memory lazily at page granularity.[10] In this way, the level-0 index provides a fast lookup from addresses to allocator-specific information, where the first stage of this associative lookup (down to a page-sized division of metadata) is encoded into the CPU's page tables, acting as a "hardware assist". Some existing tools make similar use of unreserved virtual memory [Serebryany et al. 2012; Akritidis et al. 2009; Nagarakatte et al. 2009], and the idea is closely related to "linear" page tables (first appearing in VAX/VMS [Levy and Lipman 1982] and later incorporated into more sophisticated designs [Talluri et al. 1995]).

For allocators which do not maintain adequate metadata by themselves, such as malloc() or similar untyped allocators, liballocs provides some ready-made index implementations. As a general-purpose index for untyped heap allocators, we can use a simple variation on the level-0 index. Again, we use a large linear array of metadata records, again allocated in unreserved virtual memory, but this time large enough to hold one metadata record for every byte in the suballocated region (the densest possible use). To preserve sparseness, the array is striped by the expected object size p (estimated e.g. from the first allocation). The first stripe includes enough records for one allocation per p bytes; the rare cases of denser allocation spill over into the next stripe's worth, later in memory, and so on. To support lookups on interior pointers, we remember the biggest allocation so far, and perform a linear backward search bounded by this size. This scheme supports even very tightly packed allocations.

A more specialised index is useful for malloc(), often critical for performance. We exploit its variable-size API and word-aligned chunks by storing metadata *within* the heap chunk, after incrementing the size requested, then keeping a large linear index of short pointers to the in-chunk metadata. For huge objects, a malloc() usually delegates to mmap(); we detect this and push the metadata into the mapping table. Other indexes are kept for the static allocator (a straightforward map from address to uniqtype) and for the stack (we walk the stack using libunwind[11], then use an index mapping *program counter* to frame uniqtype).

Many heaps already maintain adequate metadata, perhaps as type words in object headers. For these, the mapping table points to the relevant allocator-specific metadata retrieval routines. Sometimes, hybrid schemes are desirable. For example, headers alone cannot support queries on interior pointers. For this, liballocs can maintain an *object starts bitmap* mapping interior pointers to base pointers. Maintaining this bitmap is an issue of retrofitting, which we return to in §6.2.

4.4 Status and Performance Results

For an early indication of performance, we ran those SPEC CPU2006 benchmarks written entirely in C and not using complex numbers (currently unsupported by CIL), collecting basic data using SPEC's smaller test workloads. All eleven of these compiled under our wrapper, and ran and passed output validation with liballocs running. Table 1 shows execution times as the median of three runs on a developer-class machine (Lenovo Thinkpad T420s, Intel i7-2640M quad-core, 4GB memory) running an Ubuntu GNU/Linux 12.10 operating system. The heapmeta slowdown is observed when liballocs is loaded; it is possible to run without loading liballocs (no-load). Run-to-run variance was under 2% of the median in all cases except milc, which is very sensitive to memory placement. Small speedups were seen in some cases, in the range 0–2%; we suspect these are explained by the combined effects of varying memory placement and of the small changes our compiler wrapper makes to the compilation environment (§4.2). We anticipated the slower cases, gcc and perlbench, since they make heavy use of the "general-purpose" (non-malloc()) index (§4.3). A much simpler and faster "homogeneous" index implementation for gcc would be feasible, since its most heavily used nested allocator only allocates objects of a single type.

bench	normal/s	liballocs/s	liballocs %	no-load
bzip2	4.91	5.05	+2.9%	+1.6%
gcc	0.985	1.85	+88 %	– %
gobmk	14.2	14.6	+2.8%	+0.7%
h264ref	10.1	10.6	+5.0%	+5.0%
hmmer	2.09	2.27	+8.6%	+6.7%
lbm	2.10	2.12	+0.9%	(−0.5%)
mcf	2.36	2.35	(−0.4%)	(−1.7%)
milc	8.54	8.29	(−3.0%)	+0.4%
perlbench	3.57	4.39	+23 %	+1.6%
sjeng	3.22	3.24	+0.6%	(−0.7%)
sphinx3	1.54	1.66	+7.7%	(−1.3%)

Table 1. Basic performance results: "normal" execution time in seconds, the same under liballocs, and as a percentage. no-load is the slowdown of an allocscc-built binary when liballocs is *not* loaded. There is no data for gcc in the no-load case, owing to a bug in the GNU C library's handling of weak thread-local symbols on the x86-64 platform.

Source code to liballocs is available online.[12] The runtime is thread-safe and fairly well tested.

5. Applications

We have already implemented several interesting VM-like services on top of liballocs. Unlike conventional VMs' services, they range across the whole of a Unix process. We outline them here.

5.1 Run-time Type Checking

C and C++ are "unsafe", meaning buggy code can corrupt the program rather than raising a clean error. Debugging aids which cleanly report *memory bugs* have

[10] On Linux, this is done using mmap()'s MAP_NORESERVE flag.

[11] http://www.nongnu.org/libunwind/

[12] http://github.com/stephenrkell

emerged [Seward and Nethercote 2005; Nagarakatte et al. 2009; Serebryany et al. 2012]. However, an in-bounds, temporally valid access might still use the wrong *type*, largely because of *unchecked pointer casts*. We have implemented a run-time checker which instruments C code to check these using liballocs. It generates errors analogous to ClassCastException in Java (currently as warning messages, rather than exceptions). The implementation is simple: we use the liballocs API to define some check predicates, such as __is_a() which implements a containment-based subtyping check. We then extend our compiler wrapper to instrument casts with these checks.

A future publication will cover this system in depth, but as evidence of liballocs's value, we report some preliminary experiences here. As with liballocs, we require no source-level changes, only allocator identification (§4.1) and signposting of loose C idioms like structure prefixing (this requires a distinct check, __like_a()). These idioms present a tension between false positives and negatives. Currently, one of the benchmarks, perlbench, still shows unusably many false positives, since it relies on subtle structural polymorphism relations between its core data types. This kind of code is not even standards-compliant C, and is rare, but we are working on slightly refined checks which can accommodate it anyway.

Overhead is already low in most cases, as measured on the same eleven benchmarks seen in §4.4: the median overhead is currently 10%. As before, gcc provides the toughest test, since it has not only a very high allocation rate but also an abnormally high cast rate. Currently it suffers 160% overhead, although the same "homogeneous index" optimisation (§4.4) mooted for liballocs stands to improve this. Reporting of false positives currently dominates perlbench's execution time. Of the remaining benchmarks, the worst overhead is 40% and others are much lower still. Meanwhile, metadata coverage is proving extremely good in practice: only a handful of allocation sites (out of thousands) were not automatically typeable. Unlike C-focused systems, we can dispatch checks on allocations originating outside C code—perhaps from Fortran, or perhaps even (potentially) from a VM that can generate uniqtype instances (as we consider in §6).

5.2 Scripting without FFI

Access to native libraries from dynamic languages is typically achieved by *extending* or *embedding* a VM using "foreign function interface" (FFI) APIs at the implementation level. These are onerous to use, often change-prone and/or implementation-specific. Some tools abstract from them by generating code, either at run time[13] or beforehand [Beazley 1996], when supplied with a description of the library's interface using some ad-hoc interface model or C header files. Using liballocs, we avoid any need for the user to describe the native interface to the VM: native libraries are *already* described by their uniqtype instances, generated automatically from pre-existing DWARF metadata. We can therefore use them with the same immediacy as, say, a JRuby user picking up Java libraries.

```
require('−lXt');                          // calls dlopen...
var toplvl = process.lm. XtInitialize (    // all global syms
    process.argv[0], "simple", null , 0,  // appear in process.lm.*
    [process.argv.length], process.argv);
var cmd = process.lm.XtCreateManagedWidget( // create a button
    " exit ", commandWidgetClass, toplvl, null, 0);
process.lm.XtAddCallback(                  // add exit () as callback
    cmd, XtNcallback, process.lm.exit, null );
process.lm.XtRealizeWidget(toplvl);               // set it going...
process.lm.XtMainLoop();
```

Figure 8. Scripting liballocs-visible code in JavaScript using a lightly modified V8

We prototype this as a partial *retrofitting* of V8 onto liballocs, implementing V8's object protocol (get/set and enumeration) in terms of the uniqtype metamodel. Native objects are proxied in the V8 heap; objects passed from V8 to native code are *externalised*, i.e. moved out of V8's heap into malloc()'d storage. This generalises mainline V8's support for externalising strings and arrays. Fig. 8 shows an example session. The process's symbol bindings are exposed through the process.lm ("link map") object. When externalising [process.argv.length], the recipient XtInitialize's uniqtype is used to select the int element representation required.

Some limitations remain. Externalised objects are collected once unreachable from the V8 heap, rather than unreachable *process-wide*. Code cannot yet be externalised (e.g. to pass a JavaScript function to XtAddCallback). A debugger observing our process cannot yet read V8 frames or call V8 code. Proxy objects add unwanted indirection; we would rather V8 directly understood that some objects live outside its own heap. This could avoid large object graphs being externalised into the slower malloc() heap. All these limitations arise because we have not yet done the complementary retrofitting: implementing liballocs's meta-level protocol over V8's allocator and metadata, and pushing knowledge of this protocol into V8's compiler. We consider this tighter retrofitting in §6. As it stands, the system already improves on FFI-based approaches by avoiding their interface description step, hence providing much greater immediacy and lower maintenance effort.

5.3 Dynamically Precise Debugging

Debuggers like gdb have a major weakness: given a pointer, they cannot tell precisely what kind of object it points to. A void*, for example, cannot be followed, even though it dynamically points to some object. With liballocs, this limitation is removed. Simply calling alloc_get_type() from

[13] Examples include Java Native Access (https://github.com/twall/jna), node-ffi (https://github.com/node-ffi/node-ffi), ocaml-ctypes (https://github.com/ocamllabs/ocaml-ctypes), and others.

the debugger has already proved robust and useful. Tighter debugger integration would provide this as a debugger command, and allow *out-of-process* operation.

The same idea would allow precise monitoring and tracing, in tools such as strace or ltrace[14] which currently do not introspect on user-defined data. For example, although strace includes hand-rolled printing code for kernel data structures, it cannot provide user-side context, such as what kind of object is the target of a read() or the source for a write()—whereas liballocs can readily do so.

5.4 In-progress Applications: Serialization

Generic serialization routines for in-memory data become feasible with liballocs, since a uniqtype tells us where to find pointers in each allocation. By transitively closing over pointers, we can "deep-copy" and serialize to a file. This could even be an ELF file, since serialization is the converse of the *relocation* performed by linkers, in which pointers are marked by *relocation records* and fixed up to reflect the address bound on loading. The process's dynamic loader is therefore a candidate deserializer, using the load-time "static" allocator. One a drawback is that the resulting objects cannot be individually freed or resized later.

On-disk data is also in scope. Unix's bytewise file I/O, mimicked by all languages' standard libraries, forces even high-level programmers to write low-level code. Unix's memory-mapped files suggest an obvious extension subsuming the foregoing "deserialize in the loader" approach: a memory-mapped file can be treated like a live "allocated" region consisting of typed allocations. We are working on just such a system, focusing initially on simple binary file formats describable in terms of the uniqtype metamodel. The ELF format itself is one example: we can dynamically explore an ELF memory mapping almost exactly as if it were in-memory data, except that we must recognise additional *pointer-like* "relational" fields such as file offsets. By recording these relations in metadata structures much like our existing indexes (§4.3), we can subsequently reallocate or resize elements of the file, moving them out of the mmap()'d region into fresh memory, while remembering their logical containment and adjacency relations. This will, we hope, allow re-serializing them back into the file mapping, achieving file I/O without programming against bytes and buffers at any stage.

6. Retrofitting

How can we retrofit our existing, isolated VMs onto liballocs, so that they can cooperatively exploit our process-wide VM-like environment? This involves two principal steps: adding *meta-level visibility* by implementing the liballocs API over the VM's existing allocator, and adding *whole-process binding* by generalising the VMs' mechanisms for binding code and data together.

6.1 Meta-level Visibility

Basic plumbing Most VMs' heap allocators can largely implement the liballocs API using their existing object headers. At allocator initialization, an allocator can supply callbacks to the level-0 index (§4.3) associating it with the heap region it is to use, typically obtained from mmap(). Since headers cannot support querying interior pointers, we may additionally maintain an *object starts bitmap* (§4.3), and extend the allocation fast path to set this bit. For stack storage, VMs could generate each frame uniqtype directly, or (better) could generate DWARF, rendering them usable by unmodified debuggers.

JITting as loading Current VMs' JIT-compiled code is opaque to the wider system: it cannot be queried by dladdr(), by liballocs clients, by Unix debuggers like gdb or by existing backtrace library routines like libunwind. To fix this, we simply recognise that JIT-compiled code is an allocation like any other. Since we already view the dynamic loader as an allocator, an easy fix is to inform the dynamic loader about JITted code. To do this we must extend its interface slightly. In our prototype extension, libdlbind, we have added calls dlcreate(*name*) (create a new, temporary ELF file freshly loaded in memory), dlalloc(*size*) (reserve a chunk of space within the file) and dlbind(*name*, *addr*) (define a symbol in an existing ELF file). Collectively, these constitute an allocator for JIT compilers: they allow dynamically allocating memory backed by a temporary in-memory ELF file and setting up appropriate ELF metadata (symbol names). Type information may be attached using the liballocs API. The temporary ELF file is laid out with a large (but fixed) amount of space, allocated lazily (as in §4.3). As new code is bound into the file, the file is *re-loaded*, notifying any debugger attached via the existing r_debug protocol [AT&T 1990] and so allowing it to refresh its symbols.[15]

6.2 Cross-Process Binding

Generalising from our V8 prototype (§5.2), we want any allocation in our process to be able to bind to any other. We consider this in two parts: binding from code (accessing fields, or calling from one method to another) and binding from data (storing pointers between objects). Along the way, we will visit some issues saved from earlier: process-wide garbage collection (§2.5), and dispatch structures (§3.2).

Code–data binding, naïvely A simple interpreter might use liballocs's API directly to query target objects, obtain each's uniqtype, dynamically enumerate its fields and resolve them to memory addresses. This works even if their layout is not statically fixed (§3.6). Even though liballocs is reasonably fast, full-scale queries on each access would be slower than ideal.

[14] http://ltrace.alioth.debian.org/

[15] This "reload" is very nearly expressible using current GNU/Linux and FreeBSD dynamic linkers. Supporting it completely compatibly would require adding a "reloadable" flag to the dlopen() interface, with semantics almost (but not quite) identical to the existing RTLD_NODELETE.

Optimising code–data binding High-performance VMs use *inline caching*, i.e. caching within the instruction stream. The following snippet, adapted from the V8 developer documentation[16] shows a simple inline cache in x86 pseudo-assembly accessing a field x on some arbitrary object whose address is in register ebx. The cache speculates that the object will actually be an instance of the cached class.

```
cmp [ebx,<class offset>],<cached class>; test
jne <inline cache miss>                ; miss? bail
mov eax,[ebx, <cached x offset>]      ; hit; load field x
```

A similar inline cache retrofitted onto liballocs would speculate not only on the class of the allocation, but also *on the allocator*. Doing this adds a few instructions, but no additional memory references.

```
xor ebx,<allocator mask>               ; get allocator
cmp ebx,<cached allocator prefix>      ; test
jne <allocator miss>                   ; miss? bail
cmp [ebx,<class offset>],<cached class>; test class
jne <cached cache miss>                ; miss? bail
mov eax,[ebx, <cached x offset>]      ; hit; load field x
```

By speculating on an allocator for which the dynamic compiler has *affinity*, the fast path short-circuits away the liballocs query in favour of something allocator-specific (here class pointers). In this way, existing VMs like V8 can ship their own allocators, and generate code which specialises for them, without excluding the rest of the process: out-of-heap objects can still be accessed via a slower path.

Code–code: dispatch structures Our uniqtype deliberately omits any vtable or other dispatch structures (§3.2) because dispatch semantics are a property of the client's language, not the target allocation. Fortunately, in our evolved Unix, clients are in a position to dynamically construct their dispatch structures by querying the system for relevant code. For example, given a uniqtype such as that of Ellipse, a language implementation could query the loader for loaded functions to be considered as late-bound methods of Ellipse. This query might vary per-language: in Java, we might say "all functions declared lexically within Ellipse, left-merged with like-signature methods in inherited classes, transitively, excluding any marked final". In C++, by contrast, only methods decorated with virtual would be considered. These queries may be answered by the dynamic loader, if it retains appropriate metadata. Currently, source-level relations like "declared lexically within" are not modelled in liballocs; we are considering how to fix this by relating each uniqtype more strongly to relevant ELF symbol definitions. Once generated, dispatch structures can be attached to allocations either via on-demand reallocation (dynamically making space for the dispatch structure), via associative mappings like our index structures (§4.3), and/or via another kind of *affinity*: a compiler (say for Java) is permitted to reserve space for dispatch structures in the layouts it selects. Although allocations *need not* embed dispatch structures, they may speculatively do so, biasing them towards one implementation, but remaining bindable (albeit more slowly) from elsewhere.

Data–data binding Typically, garbage-collected heaps (GCs) are implemented in an aggressively self-contained fashion. A single heap is the unique domain of tracing, reclamation and object motion, and fixes strong invariants on addressing (typically that no interior pointer is ever stored to memory) and liveness (typically that any live object is reachable by a chain of stored pointers). They are usually also coupled to a compiler, to ensure metadata completeness (well-defined "safe points" where the compiler attests that pointers may be precisely identified) and software barriers for internal bookkeeping (most often write barriers, to remember the set of intergenerational pointers).

By contrast, liballocs is founded on embracing plurality, with a process containing many heaps and many allocators. Fortunately, the same techniques that GCs use internally can be harnessed to enable this coexistence. We have already seen a basic approach, in the externalisation of our modified V8 (§5.2). Logically this combines an *escape barrier*, to catching pointers flowing outside a given heap, with a *common heap* into which shared objects can be moved. Currently our escape barrier is the out-of-line path in V8 code for handling native accesses. Feasibly, a variation of the in-compiler write barrier could catch escaping references in-line, just as we caught incoming references in our earlier inline cache example. Since high-performance GCs are invariably capable of moving objects, moving "out" (one-way) to a common heap is an easy addition.

The problem of collection *within the common heap* remains. Per-heap *reference bits* in allocation metadata are one approach; once all are cleared, the object is free process-wide. This works when the inter-heap reference relation is acyclic—a consequence of the transitive externalisation approach we currently take with V8. A complementary approach is a *barely conservative collector*. enabled by liballocs's pervasive metadata. Although literature often talks about conservative or precise collectors, in reality any collector is only precise up to some assumptions, such as "reachability implies liveness"—itself a conservative approximation [Hirzel et al. 2002; Khedker et al. 2012]. Unlike in earlier work [Boehm and Weiser 1988], liballocs makes pointer metadata generally available, allowing precision to become the default rather than the exception. We can require users to explicitly identify the C or C++ code (say) known to exhibit "loose" behaviour with pointers (such as storing them in integer-typed storage), using instrumentation to observe writes of pointer-derived integers, much like our run-time type checker already instruments casts (§5.1). In this way, conservatism is enabled selectively at run time, rather than globally. Recent work has already found precise collection feasible for much C code [Rafkind et al. 2009] and classified real C codebases according to the loose pointer behaviours they exhibit [Chisnall et al. 2015].

[16] https://developers.google.com/v8/design as retrieved on 2015/3/30

7. Other Issues

Safety Most VMs aim to be "type-safe" in the sense of Ungar et al. [2005]: "the behavior of any program, correct or not, can be easily understood in terms of the source-level language". This implies an absence of corrupting failures. In reality, safety properties are inevitably *modulo trust* in some artifact (such as the VM implementation, a proof, a proof checker, etc.). Reducing the volume of blindly trusted code is actively being addressed by current work on verified compilation [Leroy 2009; Kumar et al. 2014] and on the large body of work which instruments or sandboxes code to provide clean failures and/or secure compartmentalisation. By extending the dynamic loader and managing metadata, liballocs is a good place to enforce policies on how loaded code must be vetted or sandboxed. Techniques typically implemented in compilers, such as control flow integrity, may be more appropriately implemented in a loader [Niu and Tan 2014]. Currently liballocs abstains from defining or enforcing such policies, but we hope to explore this in later work.

Insufficiency of "type" A uniqtype does not capture all useful semantic properties of the allocation it describes. Two obvious gaps are *initializedness* and *immutability*: respectively, whether reads or writes to a field (say) are valid. These properties are important to many reflective clients. Although we could follow the approach of C-like languages by adding read-only const and (the hypothetical) write-only invalid into the uniqtype metamodel, it seems preferable instead to extend memory-level metadata already capturing these concepts, namely read and write permissions. Proposed future machine architectures include memory protection at word granularity instead of page granularity [lowRISC Foundation 2014; Dhawan et al. 2015] which could be reflected as an extension of liballocs's metamodel.

What is "Unix"? We have been talking about Unix, but everything we have covered applies equally well to non-Unix native code environments. The other major 1970s-era design persisting into modern systems is VMS, via its successors in the Windows NT family. Like Unix, its interfaces and abstractions are light on metadata and heavy on early binding.

Functional runtimes as VMs Implementations of functional languages, such as OCaml or Haskell, exhibit only a subset of VM-like characteristics: they typically involve garbage collection but not dynamic compilation. We expect liballocs to apply usefully to such runtimes. Indeed, both OCaml and GHC runtimes are gradually improving their observability to debuggers like gdb. Haskell's laziness makes its runtime substantially quirkier, and therefore more challenging. It is worth remarking that one key application of liballocs, language interoperability, raises specific challenges here which are not revealed by the "native-to-dynamic" V8/JavaScript case we considered in §5. Consider, for example, a C library expecting to mutate a caller-supplied buffer. A Haskell client simply cannot supply a mutable object. This means that a *Haskell view* of the C code's interface lies some computational distance away: it must appear to return a fresh value, allocated by the callee, instead of doing mutation. This is a problem of *interface adaptation*, and bears a similarity to some adaptations expressible in the Cake language [Kell 2010]. After incorporating mutability into our metamodel, we hope to develop techniques by which the relevant adaptation can be automatically inserted—here, pre-allocating a fresh buffer which is passed to the C code for initialization.

8. Related Work

The Portable Common Runtime [Weiser et al. 1989] developed at Xerox PARC unified various languages, including C, Common Lisp and Modula-3, at the implementation level. Its most enduring artifact has been the Boehm collector. Although its goals are similar to ours, it explicitly avoids modelling data representations—liballocs's central feature.

Microsoft's Common Language Runtime (CLR) and, to a lesser extent, the repurposing of the JVM platform as a multi-language infrastructure [Rose 2009], seek to host ever larger fractions of a process's code on a common VM implementation. This theme continues with systems such as TruffleC [Grimmer et al. 2014]. A recurring weakness of these approaches is an inability to eliminate "awkward" yet valuable code which targets the baseline Unix interfaces and is not easily retargetable to the VM interfaces. This includes binary-only code, source code in unsupported languages such as C++ or Fortran, relatively unportable C or C++ code, and so on. This leaves the process still bifurcated into "managed" and "unmanaged" sectors, with all its familiar problems. We avoid this by several means: doggedly maintaining compatibility with Unix ways; eliciting extra semantic information only via minimally invasive techniques (such as link-time interposition or binary instrumentation); avoiding any requirement for commonality among language implementations beyond a very small shared core (the shared dynamic loader, and liballocs itself); proactively *accommodating* plurality of implementation by accepting per-allocator bindings onto a simple allocation meta-protocol. This binding is usually implemented using instrumentation (as in §3.4), and generally impinges little on the allocator implementation.

Specific implementations of individual pairs of languages have succeeded at rendering the two languages mutually interoperable [Rose and Muller 1992; Bothner 2003]. We lift this to the meta-level: rather than fixing specific implementation details, we fix only a meta-level protocol allowing diverse implementations to achieve this feat.

Object-like abstractions can be built in any language, even C, as witnessed by systems like GObject [Krause 2007]. Like conventional VMs, these are retrofittable onto liballocs. Introspection systems such as GIR and SMOKE[17] achieve

[17] https://techbase.kde.org/Development/Languages/Smoke and http://wiki.gnome.org/GObjectIntrospection/, retrieved on 2015/8/21

a subset of liballocs's benefits, but only for code that has already targeted these specific systems. Similarly, CORBA and other systems based on interface definition languages require the up-front commitment of writing user code against tool-generated stubs. By contrast, there is no need for user code to "target" liballocs; it accommodates virtually any existing code targeting the host platform.

At least one credible proposal for a "standard" memory management interfaces has appeared [Barnes et al. 1997], although without the process-wide approach of liballocs.

The author articulated a few ideas common to liballocs in earlier work [Kell and Irwin 2011], but with the emphasis on implementing VMs differently, rather than retrofittability.

9. Conclusions

We have surveyed the various rudiments of virtual machine services in Unix processes, and described liballocs, a system which expands on these by adding pervasive type metadata and a query-based metadata interface. We also identified many applications, some of them already implemented. These initial experiences have been positive, and we hope retrofitting existing VMs onto liballocs will soon enable a large reduction in complexity of common programming tasks which would otherwise involve onerous bridging of "native" and "managed" worlds.

Acknowledgments

This work was supported by the EPSRC grant EP/K008528/1, "Rigorous Engineering for Mainstream Systems". I thank Doug Lea for encouragement concerning the virtual memory techniques, and Stephen Dolan for many helpful discussions. This version has benefited from comments by Peter Sewell, Mark Florisson, Laurence Tratt and the anonymous reviewers.

References

P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

J. Aldrich. The power of interoperability: Why objects are inevitable. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 101–116, New York, NY, USA, 2013. ACM.

AT&T. *UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools*. AT&T, Upper Saddle River, NJ, USA, 1990.

K. Avijit, P. Gupta, and D. Gupta. TIED, LibsafePlus: tools for runtime buffer overflow protection. In *SSYM'04: Proceedings of the 13th USENIX Security Symposium*, Berkeley, CA, USA, 2004. USENIX Association.

G. Banavar, G. Lindstrom, and D. Orr. Type-safe composition of object modules. Technical Report UUCS-94-001, University of Utah, Salt Lake City, Utah, USA, 1994.

N. Barnes, R. Brooksby, D. Jones, G. Matthews, P. P. Pirinen, N. Dalton, and P. T. Withington. A proposal for a standard memory management interface. Presentation slides, retrieved on 2015/3/28, 1997. ftp://ftp.cs.utexas.edu/pub/garbage/GC97/withingt.ps. From the OOPSLA '97 Workshop on Garbage Collection and Memory Management [Evans and Dickman 1997].

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.

H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

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

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, New York, NY, USA, 2004. ACM.

T. A. Cargill. Pi: A case study in object-oriented programming. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 350–360, New York, NY, USA, 1986. ACM.

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

D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 117–130, New York, NY, USA, 2015. ACM.

W. R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 557–572, New York, NY, USA, 2009. ACM.

C. de Dinechin. C++ exception handling. *IEEE Concurrency*, 8(4):72–79, 2000.

U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 487–502, New York, NY, USA, 2015. ACM.

H. Evans and P. Dickman. Garbage collection and memory management. In *Addendum to the 1997 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 138–143, New York, NY, USA, 1997. ACM.

Free Standards Group. *DWARF Debugging Information Format version 4*. Free Standards Group, June 2010.

R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. Shared libraries in SunOS. In *Proceedings of the USENIX Summer Conference*, pages 375–390, 1987.

M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. TruffleC: Dynamic execution of C on a Java Virtual Machine. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform*, PPPJ '14, pages 17–26, New York, NY, USA, 2014. ACM.

M. Hirzel, A. Diwan, and J. Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Trans. Program. Lang. Syst.*, 24(6):593–624, Nov. 2002.

S. Kell. Component adaptation and assembly using interface relations. In *Proceedings of 25th ACM International Conference on Systems, Programming Languages, Applications: Software for Humanity*, OOPSLA '10. ACM, 2010.

S. Kell. In search of types. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 227–241, New York, NY, USA, 2014. ACM.

S. Kell and C. Irwin. Virtual machines should be invisible. In *Proceedings of the compilation of the co-located workshops*, SPLASH '11 Workshops, pages 289–296, New York, NY, USA, 2011. ACM.

P. B. Kessler. Fast breakpoints: design and implementation. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming Language Design and Implementation*, PLDI '90, pages 78–84, New York, NY, USA, 1990. ACM.

U. P. Khedker, A. Mycroft, and P. S. Rawat. Liveness-based pointer analysis. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, pages 265–282, Berlin, Heidelberg, 2012. Springer-Verlag.

T. J. Killian. Processes as files. In *USENIX Summer Conference Proceedings*. USENIX Association, 1984.

A. Krause. *Foundations of GTK+ development*. Springer, 2007.

R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM.

X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

H. M. Levy and P. H. Lipman. Virtual memory management in the VAX/VMS operating system. *Computer*, 15(3):35–41, Mar. 1982.

T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman, Boston, MA, USA, 1999.

lowRISC Foundation. Tagged memory and minion cores on the lowRISC SoC. lowRISC Foundation memo, December 2014. http://www.lowrisc.org/downloads/memo-2014-001.pdf. Retrieved on 2015/3/29.

B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, Nov. 1995.

S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer Berlin Heidelberg, 2002.

N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 577–587, New York, NY, USA, 2014. ACM.

J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise garbage collection for C. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 39–48, New York, NY, USA, 2009. ACM.

J. Rose. Bytecodes meet combinators: Invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11, New York, NY, USA, 2009. ACM.

J. Rose and H. Muller. Integrating the Scheme and C languages. In *Proceedings of the 1992 ACM conference on Lisp and functional programming*, pages 247–259. ACM, 1992.

SCO. System V ABI specification, 2012 snapshot. Working specification issued by The Santa Cruz Operation, December 2012. http://www.sco.com/developers/gabi/2012-12-31. Retrieved on 2015/8/21.

K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005. USENIX Association.

M. Talluri, M. D. Hill, and Y. A. Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 184–200, New York, NY, USA, 1995. ACM.

D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 11–20, New York, NY, USA, 2005. ACM.

M. Weiser, A. Demers, and C. Hauser. The Portable Common Runtime approach to interoperability. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 114–122, New York, NY, USA, 1989. ACM.