Demystifying Magic: High-level Low-level Programming *

Daniel Frampton
Australian National University
Daniel.Frampton@anu.edu.edu

Stephen M. Blackburn
Australian National University
Steve.Blackburn@anu.edu.edu

Perry Cheng
IBM Research
perryisreading@gmail.com

Robin J. Garner
Australian National University
Robin.Garner@anu.edu.edu

David Grove
IBM Research
groved@us.ibm.com

J. Eliot B. Moss
University of Massachusetts at Amherst
moss@cs.umass.edu

Sergey I. Salishev
St. Petersburg State University, Russia
Sergey.I.Salishev@gmail.com

Abstract

The power of high-level languages lies in their abstraction over hardware and software complexity, leading to greater security, better reliability, and lower development costs. However, opaque abstractions are often show-stoppers for systems programmers, forcing them to either break the abstraction, or more often, simply give up and use a different language. This paper addresses the challenge of opening up a high-level language to allow practical low-level programming without forsaking integrity or performance.

The contribution of this paper is three-fold: 1) we draw together common threads in a diverse literature, 2) we identify a framework for extending high-level languages for low-level programming, and 3) we show the power of this approach through concrete case studies. Our framework leverages just three core ideas: *extending semantics* via intrinsic methods, *extending types* via unboxing and architectural-width primitives, and *controlling semantics* via scoped semantic regimes. We develop these ideas through the context of a rich literature and substantial practical experience. We show that they provide the power necessary to implement substantial artifacts such as a high-performance virtual machine, while preserving the software engineering benefits of the host language.

The time has come for high-level low-level programming to be taken more seriously: 1) more projects now use high-level languages for systems programming, 2) increasing architectural heterogeneity and parallelism heighten the need for abstraction, and 3) a new generation of high-level languages are under development and ripe to be influenced.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation; Compilers; Memory management (garbage collection); Optimization; Run-time environments

General Terms Design, Experimentation, Languages, Performance, Reliability

* This work is supported by ARC DP0452011, ARC DP0666059, NSF ITR CCR-0085792, NSF CCR-0310988, NSF CNS-0615074, IBM, Intel, and Microsoft. Any opinions, findings, conclusions, or recommendations expressed herein are are those of the authors and do not necessarily reflect the views of the sponsors.

VEE'09, March 11–13, 2009, Washington, DC, USA.

1. Introduction

While on the one hand systems programmers strive for reliability, security, and maintainability, on the other hand they depend on performance and transparent access to low-level primitives. Abstraction is the key tool for enabling the former but it typically obstructs the latter. This conundrum is the focus of our paper. Our response is part survey, part experience report, and part manifesto. Our credo is Ken Kennedy's goal of *abstraction without guilt* [43].

Hardware and software complexity is making it harder and harder to reason about the environment in which code is written, frustrating the objective of reliable, secure, and maintainable software [2]. A standard strategy is to fortify C or C++ with a set of idioms, macros, tools, and conventions that step around the most conspicuous shortcomings of the language [31, 63, 66]. However, there are significant limits to this approach. Conventions and idioms are hard to enforce, rules such as not using threads [66] may become untenable, and some abstractions will demand non-existent native support from the base language [14]. Furthermore, while transparent access to low-level primitives is often touted as essential to performance, this argument depends on the user being able to effectively reason about hardware which is increasingly complex and subtle. Finally, we conjecture that most code written by systems programmers does not require transparent access to low-level primitives, which brings into question the use of C as a rule rather than as an exception.

An alternative and less common strategy is to write systems code in a high-level language. The remainder of this paper focuses on this objective. While this approach does not currently enjoy universal support, it is interesting to note striking similarities to the debate over moving away from assembly language programming in the mid 1970s [26, 27, 28, 38]. In that case increases in hardware and software complexity—alongside improvements in programming language technology—ultimately decided the issue, and we feel this will be the case again. For the purposes of this paper, we start by loosely defining a high-level language as one that is type safe, memory safe, and that provides strong abstractions over hardware. We then define low-level programming as that which requires transparent, efficient access to the underlying hardware and/or operating system, unimpeded by abstractions.

There are at least four distinct approaches to low-level programming in a high-level language. (a) The language could directly support low-level coding [16]. (b) The language could be extended piecemeal to support the necessary low-level features [5]. (c) Low-

level coding could be expressed in a different language, accessible from the high-level language via calls through a foreign function interface (FFI) such as Java's JNI [47]. (d) The language could be extended to support low-level coding via an extensible framework.

If, as we believe, low-level coding is the exception for systems programmers rather than the rule, then approaches such as (a) and (b) are to be avoided since they uniformly lower all coding contexts to one for low-level coding. Approach (a) suffers further because it depends on all necessary features being directly supported by the language specification, yet languages are notoriously slow to change and systems programmers must deal with hardware which can change abruptly. Approach (b) is only possible when the systems programmers have a means to extend the language or the language platform [5], which is not generally the case. The use of an FFI (approach (c)) is common, but must contend with the impedance mismatch between the two languages and the consequent performance impact. Furthermore, code written in the foreign language may in principle completely abandon the advantages of the high-level language; an all-or-nothing approach to security and reliability. We therefore argue that the high-level language be extended via an extensible framework. We also argue that the (exceptional) use of those extensions be well contained/scoped, leaving the majority of code to be written in the original, unadulterated, unextended high-level language.

We present an extensible framework for low-level programming in high-level languages. The framework is based on three fundamental requirements. 1) The ability to introduce unboxed types when the required types are not supported by the language (for example, *bit fields* or an *architectural word*). 2) The ability to introduce semantics that may not be supported by the language (for example, low-level *memory fences* [46]). 3) The ability to bypass built-in abstractions (for example, bypassing the scheduler when implementing an scheduler [5]). Furthermore, the framework leverages the idea of *containment*; minimizing the reach of unsafe operations and maximizing the scope of untainted high-level code. We will make the case that this approach maximizes the advantages of a high-level language; maximizing safety and reliability while also maximizing opportunities for optimization of the low-level code by the language runtime. Furthermore, we show that this approach allows the extensions to be *virtualized*. For example, an `Address` type (which provides operations over raw memory, similar to a **`void`**$\star$ in C) may be realized both natively, with support of compiler intrinsics from the underlying runtime, and virtualized in a pure high-level implementation that can run (inefficiently) without any special support from the runtime. Like other virtualization systems [51], this virtualized form can be a very powerful debugging tool, and—as we show in Section 5—is trivial to implement. Our framework has evolved from ten years of experience in using Java as a systems language [5] and is currently used in various forms by two major JVM implementations, one written in Java [5] and the other in C/C++ [7]. We describe elements of the framework which are currently unimplemented in either setting.

It is important to note that this work is *not* about language-in-language implementation in the tradition of LISP-in-LISP [36] and Java-in-Java [5]. While the techniques we describe may *enable* efficient language-in-language runtime systems to be developed—and this is the context in which many of our ideas have been developed—the resulting approach is more widely applicable. Moreover, language-in-language implementation involves other important challenges (such as bootstrapping and self hosting), which are orthogonal to the subject of this paper.

A handful of large projects now use high-level languages for low-level programming, including the Singularity operating system [22] and the Jikes RVM virtual machine [5]. More tellingly, when experts from each of the major JVM vendors were brought together to discuss the design and implementation of a next generation JVM, there was wide agreement that such a JVM should be implemented in Java [8]. The increasing complexity of hardware and software has led to the emergence of a new generation of programming languages [4, 17, 18], which are ripe to learn from the lessons provided from our experience with Java [5], C# [22], and Self [64]. Thus we think the time is right to re-examine high-level low-level programming.

The rest of this paper is structured as follows. We first discuss related work in Section 2 before describing our approach in Section 3. In Section 4 we present our concrete solution. Finally, in Section 5, we flesh out the approach with two concrete examples of its use.

2. Related Work

We start our discussion with the following observations:

Observation 1. *High-level languages provide abstractions that lead to software with greater security, better reliability, and lower development costs.*

Observation 2. *The goals of high-level languages tend to be at odds with low-level programming [57]. This is primarily because high-level languages gain power through abstracting over detail, while low-level programming may require transparent access to detail. There are other concerns related to performance, determinism, portability, and existing programmer skills—among others—but they lie largely outside the scope of this paper.*

2.1 Fortifying a low-level language

While individual motivations vary, it is clear that systems programming projects have found it desirable to reach for higher levels of abstraction akin to those found in high-level languages. This has a strong history stretching back to the 1970's [26, 27, 28, 38].

In today's context, memory safety is an area in which C is conspicuously lacking, and there have been countless idioms and techniques devised to improve the situation including conservative garbage collection [13, 15], smart pointers and reference counts [31], static and dynamic analysis tools such as Valgrind [51], as well as custom allocators such as `talloc` [63]. This fortification process also feeds into revisions of language specifications; future versions of the C++ specification are expected to include both high-level language features (e.g., garbage collection) as well as additional systems programming features (e.g., `constexpr` functions that are resolved at compile time).

The SAMBA Developers Guide [66] includes a set of *coding suggestions* which amount to a list of conventions and idioms designed to work around language shortcomings in areas such as safety and portability. Enforced by convention rather than by the semantics of the underlying language, these often exist to simply work around artificial limitations of the base language.

There is also a limit to the extent one can introduce high-level abstractions into an existing language. High-level abstractions such as threads have been shown to be problematic to implement [14], as correctness can not be ensured without the cooperation of the underlying language.

2.2 Systems programming languages

There has been a long history of language development targeting low-level or systems implementation, with differing degrees of innovation and success [16, 44, 53, 60, 68]. The Modula series of languages [16] was notable in several respects. It had a stronger notion of type safety, integrating garbage collection into the programming environment by providing two heaps for dynamic allocation: one garbage collected and the other explicitly managed. Modula-3 also

included a *module system*, now uniformly considered an important building block for developing large and complex systems. While there is criticism of the Modula implementation, the lack of such a system providing intermediate levels of visibility is noted as a shortcoming in C [55], and—prior to the introduction of namespaces in 1994—also in C++ [61].

There have also been attempts at creating safer derivatives of C, an example being Cyclone [42], which introduces stronger type and memory safety, as well as a safe approach to multithreading [34].

This type of approach—in addition to approaches that aim to provide richer static-analysis tools to prove 'unsafe' code correct [23]—focuses on proving low-level programming techniques correct, rather than allowing high-level low-level programming. We believe this neglects our second observation above: by maintaining a transparent view to low-level details *across the board*, much of the potential gains of high-level abstractions are lost.

2.3 Two-language approaches

The most common technique used to resolve the tension between high-level and low-level programming is to simply use different languages or dialects for each task. This general approach affords itself to several solutions, based primarily around the level of integration of the two languages.

Extreme: Don't use high-level languages for low-level tasks. This is perhaps the most extreme position but is also the status quo, demonstrated through the continued dominance of C and C++ for low-level programming while other languages continue to enjoy increased popularity for general programming tasks.

Intermediate: Call out using a foreign function interface (FFI). This technique provides an *escape hatch* where the programmer can call into a separate language to implement low-level features, and is available in almost all modern language environments, from C's ability to call into assembly, to Java [33] with the Java Native Interface [47] and C# [20] (in the Common Language Infrastructure [21]) with Platform Invoke. This allows some low-level programming, such as that in Java described by Ritchie [56], but it is a coarse-grained approach and the split between low-level and high-level code can compromise both software design and performance.

Minimal: Introduce dual-semantics to the language. A refinement of the above technique is to introduce regions within the high-level language that allow the use of low-level language features. This allows greater coherency between the high- and low-level aspects of the system being implemented. Modula-3 [16] achieves this through *unsafe modules* (where features such as the use of pointers or unchecked type casts are allowed) and *unsafe interfaces* (which can only be called from unsafe modules). Safe modules may interact with unsafe modules by exposing an unsafe module through a safe interface. C# [20] and the CLI [21] also use a similar concept of unsafe code, but control the the interaction of safe and unsafe code at a coarser granularity.

Techniques to reduce both the design [37] and performance [59] disadvantages of these approaches exist. There is however a more fundamental problem in that they treat the need to perform low-level operations as an all-or-nothing requirement. This does not resolve well with our above observations: it should be possible to leverage high-level abstractions for everything other than the *specific* low-level detail you are dealing with.

2.4 Foundations of our work

Another approach is to provide the tools with which one can *extend* a high-level language to perform low-level programming. This is the general approach taken throughout our work. Much of the progress in this area has been through projects where the focus was not on language itself. Two areas of research in particular—operating systems and virtual machines—have been largely responsible for the progress in this area, although contributions to language design have generally been driven by other pragmatic goals.

Operating systems have been developed using high-level languages including Modula-3 [9], Haskell [35], Java [52], and C# [40]. SPIN [9] is a research operating system focused on safety, flexibility, and extensibility [10], and is written in an extended version of Modula-3 [16]. Extensions include improvements to allow interoperability with externally defined interfaces (such as for accessing hardware devices), changes to improve error-handling behavior, and the ability to safely cast raw memory into typed data [24]. Yamauchi and Wolczko [69] embed a small virtual machine within a traditional operating system to allow safer drivers written in Java, while the Singularity project [40, 41] (written using Sing#, an extension of Spec#, itself an extension of C#) aims to discover how a system should be built from the ground up for high-level language execution, including models for inter-process communication [3, 22].

There have been many examples of virtual machines written using high-level languages [6, 12, 25, 54, 58, 49, 64, 67], most likely due to the combination of a systems programming task in concert with a deep understanding of a high-level language. Virtual machine development is the context within which much of our work has been undertaken. Jikes RVM, formally known as Jalapeño [6] is a high-performance Java-in-Java virtual machine, requiring extensions—known as *magic*—to support required low-level operations [5]. Maessen et al. [48] provided a deeper understanding of how magic operations interact with the compiler, and what steps must be taken to ensure correctness in the face of compiler optimizations. OVM, an ahead-of-time Java-in-Java virtual machine aimed at real-time applications, used similar magic idioms, but built more principled abstractions around them [25]. Moxie [12] is a clean-slate Java-in-Java virtual machine that was used to prototype some of the ideas that have helped to feed into our approach. The `sun.misc.Unsafe` API, implemented by current production virtual machines, and implemented in Jikes RVM through our more general magic framework, provides some similar functionality. Interestingly, it may be possible to use `sun.misc.Unsafe` as a pragmatic means to implement a limited subset of our framework on existing production virtual machines.

3. High-level Low-level Programming

We now describe our approach to low-level programming in a high-level language. Our premise is that *high-level programming is desirable whenever it is reasonably achievable*. High-level languages are designed to abstract over the specifics of the target environment, shielding the programmer from complexity and irrelevant detail so that they may focus on the task at hand. In a systems programming task, however, there is often a need for transparent access to the lowest levels of the target environment. The presence of high-level abstractions can obstruct the programmer in this objective.

3.1 Approach

Our approach is guided by a principle of containment whereby we minimize exposure to low-level coding both in *extent* (the number of lines of code) and *depth* (the degree to which semantics are changed). Our view is that to achieve this efficiently, effectively, and safely, adding low-level features to high-level languages requires: (1) extensibility, (2) encapsulation, and (3) fine grained lowering. We will now describe each of these attributes in more detail.

Extensibility. To reach beyond the semantics of a high-level language, systems programmers need to be able to either change the language (generally infeasible), use a different language (undesir-

able), or extend the language. Jikes RVM took the third approach. However, the original Jikes RVM approach had two notable shortcomings: a) the extensions were unstructured, comprising a potpourri of ad hoc extensions accreted over time; and b) the extensions required modification to the compiler(s) and runtime. An extensible framework for introducing and structuring low-level primitives is necessary. Such a framework will maximize reuse and not require modifying the source of the language runtime in order to provide new extensions. We discuss our extensible framework in Section 4.

Encapsulation. Thorough containment of low-level code is essential to minimize the erosion of any advantages of the high-level language setting. Two-level solutions, such as those provided by foreign function interfaces (FFIs) [47], unsafe sub-languages [16, 20], or other means [37], tend to polarize what is otherwise a rich spectrum of low-level requirements. Consider the implementation of a managed runtime, where on one hand the object model may internally require the use of pointer arithmetic, while the scheduler may instead require low-level locking and scheduling controls. Simply classifying both as 'unsafe' renders both contexts as equivalent, reducing them to the same rules and exposing them to the same pitfalls. By contrast, a general mechanism for *semantic regimes* may allow low-level code to be accurately scoped and encapsulated, avoiding under- or over-specification.

```
1 @AssertSafe  // Any code may call this method
2 @UncheckedMemoryAccess
3 public Word getHeader(ObjectReference ref) {
4   return ref.loadWord(HEADER_OFFSET);
5 }
```

Figure 1. Unsafe code encapsulated within a safe method.

We illustrate encapsulation in Figure 1, where a safe method, `getHeader()`, is implemented through safe use of an unsafe memory operation, `loadWord()`.[1] The `@UncheckedMemoryAccess` annotation is used to scope the method to explicitly permit its use of `loadWord()`, while the `@AssertSafe` annotation encapsulates the unsafe code by asserting that calls to `getHeader()` are 'safe'. This allows `getHeader()` to be called from any context. The result is a more general and extensible means of describing and encapsulating low-level behavior than the practice of simply declaring entire contexts to be either 'safe' or 'unsafe'. We discuss the implementation of semantic regimes in our concrete framework in Section 4.2.2.

Fine grained lowering. A key issue when lowering semantics is the granularity at which that lowering occurs with respect to program scope. Coarse grained approaches, such as the use of FFIs, suffer both in performance and semantics. Performance suffers because of the impedance mismatch between the two language domains. In some cases crossing this boundary requires heavy-weight calling conventions [47], and it is generally difficult or impossible for the high-level language's compiler to optimize across the boundary. (Aggressive compiler optimizations have recently been shown to reduce this source of overhead [59].) Similarly, the coarse grained interface can generate a semantic impedance mismatch, requiring programmers who work at the interface to grapple with two distinct languages. Instead, we argue for introducing semantic lowering at as fine a grain as possible. Thus in the example of the object model in Figure 1, the programmer implementing `getHeader()` must (of course) reason about the layout of objects and their headers in memory, but is not required to code in an entirely distinct language, with all the nuances and subtleties that entails. Further, an optimizing compiler can reason about `loadWord()`, and, if appropriate, inline the `getHeader()` method and further optimize within the calling context. In practice, the result yields performance similar to a macro in C, but retains all of the strengths of the high-level language except for the precise concern (memory safety) that the programmer is required to dispense with.

[1] The safety of `getHeader()` is due to the use of the strongly typed `ObjectReference`. The method would not have been safe had the weakly typed `Address` (i.e., **`void`**$*$) been used.

3.2 Requirements and Challenges

Having outlined our approach at a very high level, we now explore the primary concerns that face the construction of a framework for high-level low-level programming. The challenges of low-level programming in a high-level language fall broadly into two categories: 1) the high-level language does not allow data to be represented as required, and 2) the high-level language does not allow behavior that is required.

3.2.1 Representing Data

Low level programming may often require types that are not available in the high-level language. For example, high-level languages typically abstract over architecture, but low-level programming may require a type that reflects the underlying architectural word width. Additionally, an operating system or other interface may expect a particular type with a certain data layout which is unsupported by the high-level language.

Primitive types. It may be necessary to introduce new primitive types—types that could otherwise not be represented in the language—such as architecturally dependent values. In the original Jalapeño, a Java **`int`** was used to represent an architectural word. This suffered from a number of fairly obvious shortcomings: Java **`int`**s are signed, whereas addresses are unsigned; a Java **`int`** is 32-bits, making a 64-bit port difficult; and aliasing types is undesirable and undermines the type safety of the high-level language. (For the 64-bit port, it was necessary to disambiguate large numbers of **`int`**s throughout the code base, and determine whether they were really addresses or integers [65]). Ideally systems programmers would be able to introduce their own primitive types for such purposes. This objective might imply that operators over those types could be added too.

Compound types. Systems programmers must sometimes use compound types to efficiently reflect externally defined data, such as an IP address. Because these are externally defined, it is essential that the programmer have precise control over the layout of the fields within the type when required. Typically, a language runtime will by default do its best to pack the fields of a type to optimize for space, or to improve performance through better locality, etc. However, the need to interface with externally defined types means that the user must be able to optionally specify the field layout. Some languages (e.g., C# [20]) provide fine control over field layout, but others (e.g., Java [33]), provide none.

Unboxed types. High-level languages allow users to define compound types. However, these types are often by default 'boxed'. Boxing is used to give an instance of a type its identity, typically via a header which describes the type and may include a virtual method table (thus identifying the instance's semantics). From a low-level programmer's point of view, boxing presents a number of problems, including that the box imposes a space overhead and that the box will generally prevent direct mapping of a type onto some externally provided data (thereby imposing a marshaling/-copying overhead at external interfaces). *Unboxed types*—types that are stripped of their 'box'—allow programmers to create compound types similar to C `struct`s. User-defined unboxed types

are not uniformly supported by high-level languages (for example, Java does not offer user-defined unboxed types). Integration of unboxed types into an environment implies a variety of restrictions. For example, subtyping is generally not possible, as there is no way of reestablishing the concrete subtype from the value due to the absence of a box that captures the instance's type. Furthermore, in some languages there is no way to refer to an instance of an unboxed type (if, for example, the language does not have pointers, only object references), which limits unboxed types to exist as fields in objects or as local variables. C# provides unboxed types, and supports interior pointers to unboxed types as fields of objects.

References and values. Conventionally, data may be referred to directly (by value) or indirectly (by reference). In many high-level languages, the language designers choose not to give the programmer complete freedom, preferring instead the simplicity of a programming model with fixed semantics. For example, in Java, primitive types are values and objects are references; the system does not allow an object to be viewed as a value. Thus Java has no concept of pointer, and no notion of type and pointer-to-type. Since pointers are a first order concern for systems programmers, a low-level extension to a high-level language should include pointers, and allow the value/reference distinction to be made transparent when necessary.

3.2.2 Extending the Semantics

In the limit, a systems programmer will need to access the underlying hardware directly, unimpeded by any language abstractions. This problem is typically solved by writing such code in assembler, following a two-language approach. Alternatively, *intrinsic functions* could be added to the language which directly reflect the required semantics, and *semantic regimes* could be defined within which certain language-imposed abstractions would be suspended.

Intrinsic functions. Intrinsic functions allow the addition of operations that are not expressible in the high-level language. An example of this is a systems programmer's need to control the hardware caches. For example, Jikes RVM (like most virtual machines) dynamically generates code and for correctness on the PowerPC platform, must flush the data cache and invalidate the instruction cache whenever new code is produced. However, a high-level language such as Java abstracts over all such concerns, so a programmer would typically resort to a two-language solution. Likewise, the implementation of memory fences [46] and cache prefetches [30] require semantics that are architecture-specific, and that a high-level language will abstract over. Intrinsic functions are widely used, and in the case where the systems programmer *happens* to be maintaining the very runtime on which they depend, they may readily implement intrinsic functions to bypass the language's restrictions. Ideally, a high-level language would provide some means for extensible, user-defined intrinsics. In that case, the user would need to provide a specification of the required semantics. In the limit such a specification may need to be expressed in terms of machine instructions, augmented with type information (to ensure memory safety) and semantic information (such as restricting code motion) essential to allowing safe optimization within the calling context.

Semantic regimes. In addition to *adding* new operations to the semantics of the high-level language, sometimes low-level coding will necessitate *suspending* or *modifying* some of the semantics of a high-level language. This scenario is particularly common when a virtual machine is implemented in its own language, as it must curtail certain semantics to avoid infinite regress; the virtual machine code that implements a language feature cannot itself use the language feature it implements. For example, the implementation of `new()` cannot itself contain a `new()`. For semantics that are directly expressed in the high-level source code (such as `new()`) this is achievable through careful coding. However, an explicit semantic regime can be a valuable aid in automatically enforcing these restrictions. In other cases, the semantics that need to be suspended are not controllable from the high-level language. For example, low-level code may need to suspend array bounds checks, avoid runtime-inserted scheduling yieldpoints, be compiled to use non-standard calling conventions, or be allowed to access heap objects without executing runtime-inserted garbage collector read/write barrier sequences. By defining orthogonal and composable semantic regimes for each of these semantic changes, the programmer can write each necessary low-level operation while preserving a maximal subset of the high-level language semantics. Thus ideally a runtime would provide a means of defining new semantic regimes and applying such regimes to various programming scopes.

4. A Concrete Framework

We now take the general approach outlined in the previous section and explore how it plays out in practice. Concretely, we introduce a framework for building language extensions that allows Java to support low-level programming features. This framework is the basis for the publicly available `org.vmmagic` package. We characterize the extensions in terms of the same categories we use in the preceding section: extending the type system and extending language semantics.

In addition to the requirements discussed in the previous section, for a concrete realization we add the pragmatic goal of minimizing or eliminating any changes to the high-level language syntax. This enables us to leverage existing tools and retain portability. Portability is important to the examples we discuss in Section 5, where we use virtualized implementations of magic to rehost code into different contexts, potentially running on different host runtimes. This powerful facility depends on portability.

To help ground our discussion, we will use a running example of the evolution of an `Address` type, as shown in Figure 2. This is an abstraction that provides functionality similar to that provided by an untyped pointer (`void*`) in C, an unsafe feature absent from many high-level languages but essential for many low-level tasks. For simplicity, we show only a very minimal subset of the `Address` type as it evolves. Although for concreteness our example is expressed in terms of Java syntax, the abstract approach from Section 3 and many aspects of this concrete framework are language-neutral, including applicability beyond Java-like languages to others including dynamic object-oriented languages like Python.

The `org.vmmagic` package has in various forms been both used by and shaped by use in three Java-in-Java JVMs (Jikes RVM [5], OVM [25], and Moxie [12]), one C/C++ JVM (DRLVM [7, 32]), and one operating system (JNode [52]). Much of what we describe here is publicly available in the 3.0.1 release of Jikes RVM; some aspects are currently under development, and a few other clearly identified aspects of the framework are more speculative.

```
1 class Address {
2   ...
3   byte loadByte();
4   void storeByte(byte value);
5   ...
6 }
```

Figure 2. First attempt at an `Address` type.

4.1 Type-System Extensions

In Section 3.2.1 we discussed the system programmer's requirement of being able to extend the type system. We address these requirements concretely through two mechanisms. The first, *raw*

storage, allows the introduction of types with explicit layouts that may depend on low-level characteristics of the target system. The second allows us to introduce *unboxed* types with control over field layout.

4.1.1 Raw Storage

Raw storage allows the user to associate an otherwise empty type with a raw chunk of backing data of a specified size. The size may be specified in bytes, or more abstractly in terms of architectural width words (whose actual size will be platform dependent). Raw storage is a contract between the writer of the type and the runtime system which must allocate and manage the data. Raw storage is not visible to the high-level language, and can only be accessed through the use of intrinsic functions. In Figure 3, the `@RawStorage` annotation[2] is used to associate a single architectural word with our `Address` type.

```
1  @RawStorage(lengthInWords=true, length=1)
2  class Address {
3    ...
4    byte loadByte();
5    void storeByte(byte value);
6    ...
7  }
```

Figure 3. Associating a word-width payload with `Address`.

This example shows how the raw storage mechanism allows systems programmers to fabricate basic (non-compound, unboxed) types. In Section 4.2.1 we will discuss how the programmer can define operations over such types.

At present we have limited our framework to byte-granularity storage. However, as future work we intend to explore sub-word granularity storage and layout. Bit-grained types are important to projects such as Liquid Metal [39], which may use this framework, or similar. Prior work in the OVM project tentatively explored this issue [25]. The SPIN project described an example of packet filtering in Modula-3 which used bit masks and bit fields, however the example they gave was at a 16 bit (2 byte) granularity [24].

4.1.2 Unboxed Types

We allow programmers to define unboxed types by marking class declarations with an `@Unboxed` annotation. Since an unboxed type is only syntactically distinguished from an object, we rely on the runtime compiler ensuring that unboxed types are never used as objects. Our current implementation in Jikes RVM is limited to supporting single field types (such as `Address`), which are treated like Java's primitives and are thus passed by value and allocated only on the stack.

Control of field layout. When specifying an unboxed type, our framework allows the programmer to specify that field order should be respected by setting the layout parameter to `Sequential`, and requires the user pad the type with dummy fields as necessary (as is commonly done in C). This allows the programmer to precisely match externally defined types.

Support for compound unboxed types and pointers to unboxed types are not available in Jikes RVM 3.0.1, but will be released in a future version.

4.2 Semantic Extension

Our framework follows the discussion in Section 3.2.2, providing two basic mechanisms for extending the semantics of the language:

[2] Here we use the Java annotation syntax to annotate the type. As demonstrated in [25] and [5], other mechanisms such as marker interfaces can be used to similar effect when the language does not explicitly support an annotation syntax.

```
1  @Unboxed(layout=Sequential)
2  class UContext {
3    UInt64 uc_flags;
4    UContextPtr uc_link;
5    StackT uc_stack;
6    ...
7  }
```

Figure 4. Unboxing with controlled field layout.

1) *intrinsic functions*, which allow the expression of semantics which are not directly expressible in the high-level language, and 2) *semantic regimes*, which allow certain static scopes to operate under a regime of altered semantics, according to some contract between the programmer and the language implementation.

4.2.1 Intrinsic Functions

Intrinsic functions amount to a contract between the programmer and the compiler, whereby the compiler materializes the function to reflect some agreed-on semantics, inexpressible in the high-level language. In early implementations of magic in Jikes RVM [5], the contract was implemented by compiler writers intercepting method calls to magic methods in the Java bytecode (identified by the class and method being called) and then realizing the required semantics in each of the three runtime compilers instead of inserting a method call.

```
1  @RawStorage(lengthInWords=true, length=1)
2  @Unboxed
3  class Address {
4    ...
5    @Intrinsic("org.vmmagic.unboxed.loadByte")
6    native byte loadByte();
7    ...
8    @Intrinsic("org.vmmagic.unboxed.storeByte")
9    native void storeByte(byte value);
10   ...
11   @Intrinsic("org.vmmagic.unboxed.wordLLT")
12   native boolean LT(Address value);
13   ...
14 }
```

Figure 5. Use of intrinsics for `Address`.

Moxie [12] developed the idea further by canonicalizing semantics, separating the usage of an intrinsic operation from the semantics of the operation itself. Figure 5 shows how intrinsic function declarations can then reference the desired semantics, with the intrinsic function declarations of `loadByte` and `storeByte` referring to canonical `loadByte` and `storeByte` semantics—both of which may be (re)used by corresponding intrinsics within other types (e.g., an `ObjectReference` type). The benefit of this approach becomes clear as we extend `Address` to include more intrinsic operations, such as the less-than ($<$) intrinsic in Figure 5, which is defined in terms of canonical `wordLLT` semantics, and could again be reused by a number of word-sized types. In the Moxie solution, individual compilers thus only needed to understand how to provide each of the full set of intrinsic semantics once, no matter how many times they were used.

The conspicuous limitation of all the described approaches to intrinsic functions is that they require the co-operation of those maintaining the host runtime. This is convenient when the runtime itself is the coding context, but is not a general solution. A more general approach—and the one that we have taken—is to associate the semantics of individual intrinsic operations with `IntrinsicGenerator` instances. These instances understand how to generate the appropriate code for an intrinsic operation. In our current implementation `IntrinsicGenerator` in-

stances are stored in a table indexed by the unique string provided at the intrinsic function declaration (e.g., "org.vmmagic.unboxed.loadByte" in Figure 5). Currently, our IntrinsicGenerator instances must be implemented with knowledge of the compiler internals (to allow the intrinsics to code their own semantics), but in the future we intend to allow intrinsics to be constructed more generally, through either providing a set of compiler-neutral building blocks, or the use of a specialized language such as CISL [50].

4.2.2 Semantic Regimes

Recall that in Section 3.2.2 we introduced the idea of statically scoped semantic regimes which change the default language semantics. When the compiler encounters code that is marked with a semantic regime it treats it specially. Currently, support for individual semantic regimes must be hard-coded into the compiler. This includes turning on and off language features such as bounds-checks, the use of locking primitives, and the presence of yield points. It also includes allowing (or disallowing) calls to certain language features, such as calls to **new**(), or the use of unchecked memory operations (e.g., @UncheckedMemoryAccess in Figure 1). This mechanism is essential to our objective of containment, allowing the finely specified, well scoped declaration of a region with changed semantics.

5. Real World Experience

The framework we describe has emerged as the pragmatic consequence of a decade of experience with systems programming in the context of high performance JVMs. The framework is thus solidly grounded in real-world experience. In this section we discuss this experience and provide two case studies that illustrate a little of the power of our approach. The first describes Jikes RVM's memory management toolkit, MMTk [11], which is written in Java and makes extensive use of our framework. The second describes how strong abstraction facilitates *virtualization*, allowing us to create a powerful synthetic debugging harness for MMTk at no cost to production performance. We conclude the section by discussing other interesting applications of this virtualization capability to various runtime services.

5.1 Deployment

Our ideas have evolved through real-world experience in the implementation of three Java-in-Java virtual machines [5, 12, 25], a Java operating system [52], and a C/C++ JVM [7].

The use of a Java instantiation of our framework, org.vmmagic, in DRLVM [7]—a C/C++ JVM based on the ORP [19] and StarJIT [1] code bases—is particularly interesting. DRLVM uses the framework to express runtime services such as write barriers and allocation sequences in Java. Our Java-based framework made the code easier to express, removed the impedance mismatch between the service code and the user context in which it is called, and allowed the service code to be trivially inlined and optimized into application code. Previously, DRLVM had used ORP's LIL [32] to express service code. Aside from providing a more natural medium to express the service code, the use of our framework was motivated by performance [45]. At the time of writing, our framework is used by DRLVM to implement actions including object model operations, class registry access, lock reservation (lock biasing), accessing the current Thread object, the object allocation fast path, and garbage collection write barriers.

Jikes RVM makes extensive use of our framework and is the primary environment from which org.vmmagic emerged. The memory management subsystem makes particularly heavy use of org.vmmagic, principally because it is concerned with accessing raw memory, which is not supported by regular Java semantics. As the single largest user of the framework, we have focused much of the discussion in this section on the memory manager. However, org.vmmagic is used throughout Jikes RVM in a variety of capacities. A few examples of the wide variety of semantic regimes used by Jikes RVM include: stipulating that an object may not move (NonMovingAllocation); defining the special semantics of trampoline code, which by definition never returns (DynamicBridge); preventing optimization (NoOptCompile); asserting callee save semantics for volatiles (SaveVolatile); and eliding null checks (NoNullCheck). Jikes RVM also makes use of a wide variety of compiler intrinsics, including: atomic operations used to implement locks; memory barrier and cache flushing operations (required when compiling code and initializing classes on architectures with weak memory models); stack introspection (for exception delivery and debugging); and persisting, modifying, and restoring thread state (to support exact garbage collection, green thread scheduling, and exception delivery). The unboxed magic types used by the memory manager (Word, Address, ObjectReference, Offset, and Extent) are used throughout the JVM.

5.2 MMTk

MMTk [11] is a high performance Memory Management Toolkit written in Java that leverages org.vmmagic for low-level access to the underlying hardware. MMTk was initially developed as the memory manager for Jikes RVM, but has been successfully ported to other Java [12] and non-Java [7, 29, 62] runtimes, and has thus evolved a clean interface with respect to the underlying runtime environment.

Why Java? MMTk uses Java for two distinct reasons. First, MMTk derives significant software engineering benefits from being implemented in a high-level, strongly typed language [11]. Second, MMTk is written in the same language that it was originally designed to support. This avoids an 'impedance mismatch' between the supported language and the language in which the support is written, which provides significant performance advantages, as noted by the Jalapeño [6] and DRLVM [45] experiences.

The performance advantage of implementing in Java is borne out by both a) our own empirical inspection of compiled code fragments for performance critical sections of MMTk, and b) through a direct performance comparison with a high quality C implementation of a memory manager [30]. When the language impedance mismatch is removed, performance critical code (object allocation and write barriers) can be inlined into user code, and the optimizing compiler can produce code as good as or better than hand-selected machine code.

The traditional language for implementing memory managers is C, but it would be very difficult to build a toolkit as flexible as MMTk in C. Using C++ may make it possible to achieve an equally flexible structure, but high performance allocation and barriers would require a complex and fragile solution, such as providing hand-crafted IR fragments to the compiler [32], or taking DRLVM's approach and using org.vmmagic for the helper code and C/C++ for the remainder of the memory manager implementation.

Low-level coding. To illustrate why MMTk requires low-level access to hardware resources, consider the operation of tracing an object in a parallel copying garbage collector. Given an object the collector must:

1. Determine where references to other objects are located in the object, typically by consulting a reference map linked from the object's header; then
2. Take each reference location, load the reference and:
   (a) Determine that the reference is non-null (if not then we move on to consider the next reference location);

(b) Determine that the reference does not point to an already copied object (if it does then we update the reference location to point to the new copy and move on to consider the next reference location),
(c) Atomically mark the object to ensure only one copy is made (forcing other threads to spin and wait for us to finish if they are also considering it);
(d) Allocate an area in the target space and copy the contents of the object to it;
(e) Store a forwarding pointer from the old version of the object to the new copy (also unmarking the object to allow any threads waiting in Step 2c to continue); and
(f) Update the reference location to point to the new copy, and then move on to consider the next reference location.

Two things are clear from the above operations. First, many of the operations required—such as determining pointer locations, updating them, and accessing synchronization information in object headers—deal with information that is generally not accessible in a high-level language. Second, given that the number of objects in the heap generally runs into in the millions—and that garbage collection can take a significant fraction of execution time (a common rule of thumb is 10%)—any unnecessary overhead (such as that from a foreign function interface) would have an intolerable performance impact. Fine-grained intrinsics (see Section 4.2.1) which the optimizing compiler can reason about are vital to achieving performance. We were successful [11, 30] in meeting our goal that MMTk perform *at least* as well as a very well tuned, optimized C/C++ memory manager implementation.

5.3 The MMTk Debugging Harness

Garbage collectors are notoriously difficult to debug. (Even when written in a type safe high-level language!) They are very tightly bound to the environment in which they execute: an error in write barrier code could cause pointers to become corrupted, manifesting as errors in user code with no apparent link to the code that produced the error. Modern garbage collectors also tend to be parallel (where multiple collector threads perform collection work at the same time) as well as concurrent (where collection work executes in parallel with user code). Modern programming styles also dictate that memory managers support parallel allocation and write barriers, due to the prevalence of multithreaded user code. These characteristics tend to conspire to make debugging a garbage collector within a production JVM a challenge.

The MMTk harness seamlessly rehosts MMTk from the complex, natively executing environment of a high performance JVM to an environment that is synthetic, controlled, and yet rich. MMTk can be debugged in this controlled environment using scripts expressed in a trivial domain-specific language executed by an interpreter written in Java. Rehosting MMTk onto this interpretive debugging engine requires a) a simple virtual machine interface layer targeted at the harness, and b) a *virtualized* implementation of `org.vmmagic` written in pure Java. The virtual `org.vmmagic` classes replace raw memory accesses with a virtualized view of memory, simulated within the harness as a hash table of memory pages (each in turn implemented as an array of integers).

The MMTk harness provides many debugging options that are unavailable in existing virtual machine implementations. We can put arbitrary watchpoints on words of memory, or on particular objects or sets of objects. We can also use graphical debuggers such as the Eclipse debugger. Most importantly we can write unit tests to exercise specific aspects of individual collectors, including during development where collectors are incomplete, something that is extremely problematic in virtual machine implementations where the memory manager must also manage other memory allocated by the virtual machine itself (in addition to the executing application).

```
1  class Address {
2    int address;
3
4    byte loadByte() {
5      return SimMemory.loadByte(address);
6    }
7
8    void storeByte(byte value) {
9      SimMemory.storeByte(address, value);
10   }
11 }
```

Figure 6. Virtualized version of `Address`.

The actual implementation of the virtualized `org.vmmagic` is quite straightforward, as the required operations—rather than being executed natively as intrinsics by the host runtime—fall through to the pure Java implementation working on top of our virtualized memory environment. Figure 6 shows an example of how `Address` can be implemented on top of the virtualized memory.

Note that in `org.vmmagic` magic types such as `Address` and `ObjectReference` are implicitly value types, so need to be passed by value. However, in a pure Java implementation the magic types will be realized as regular Java objects and thus be subject to Java's pass by reference semantics. Fortunately, in our current implementation of `org.vmmagic` the magic types are immutable,[3] so pass by value and pass by reference are semantically equivalent and our pure Java virtualization is trivial. Had this not been the case, we could have achieved pass by value semantics by copying the values as they were passed. This could be done transparently via code rewriting or bytecode rewriting.

5.4 Further Opportunities for Virtualization

The strongly abstracted view of low-level features is what allowed the virtualization leveraged by the MMTk debugging harness. We foresee other interesting applications of such virtualization. For example, an interesting twist on the debugging harness would be to use virtualization to drive MMTk via a dislocated JVM rather than via the harness. As with the debugging harness, MMTk would run in a pure Java environment on a host JVM. However, unlike the debugging harness, the heap would exist not as a hash table of memory pages, but as a real heap in the host JVM, accessed via a network socket. Such a scheme could be used, for example, to debug when the JVM is executing on a very constrained environment, such as an embedded processor, unsuitable for the rich debugging environments available on mainstream development platforms.

6. Future Work

While the approach outlined in this paper has evolved from a decade of real-world experience, and has been proven in practice through several projects, there is significant promise for future work in a number of directions.

Firstly, there is a clear benefit to undertaking the refinements of the type system extensions alluded to in earlier sections. These include allowing more general unboxed compound types—and consequently pointers to these types—as well as the bit-level specification of types such as those required by Lime [39]. While evidently not required for our existing applications, the addition of this functionality should add broader appeal to our approach.

To further extend the appeal of our approach, it would be interesting to implement it within the context of other language runtimes (both Java and non-Java). This would provide additional incentive to develop the compiler-independent representation of the

[3] For example, a field `ptr` of type `Address` cannot be incremented; a reassignment idiom must be used: `ptr = ptr.add(1);`.

`IntrinsicGenerator` objects (see Section 4.2.1) to reduce the amount of porting work demanded of users of the framework.

Also, while a large degree of the power of our approach lies with its extensible nature, it would be worthwhile developing a basic set of semantic regimes, unboxed types, and intrinsics. This would both avoid individual users reinventing the wheel, as well as providing a basic set of functionality to make the framework more approachable.

Finally, we hope that this paper will help to drive the adoption of high-level low-level programming as a technique within systems programming more broadly. By drawing together our approach into a coherent framework, we anticipate that it will open up the power of our approach to other developers who do not have the freedom to change the underlying runtime system.

7. Conclusion

Hardware and software complexity is making it harder and harder to reason about the environment in which code is written, frustrating the objective of reliable, secure, and maintainable software. Now more than ever, systems programmers need to be embracing high-level languages. However, although abstraction is the key to many of the benefits of high-level languages, it typically obstructs low-level programming. We have explored this conundrum in a pragmatic, experience-oriented setting.

Our contributions include: 1) characterizing the problem and identifying a solution, 2) illustrating our solution through two novel and concrete case studies, 3) outlining ambitious future directions for the work, and 4) drawing together a large and fragmented body of related work. Our solution is a publicly available work-in-progress, and is used in various forms in a number of JVMs (both Java and C), and one operating system. The primary advantages of our approach are: 1) the capacity to minimize the intrusion of low-level code into the high-level setting, both in extent, and in character; 2) the ability to trivially virtualize systems built upon this framework; and 3) outstanding performance (that has led to its use in a C/C++ JVM [7], on the grounds of improved performance [45]).

As new languages emerge [4, 17, 18, 39], we hope the designers will carefully consider the possibility of supporting low-level programming, and that they might find our work useful.

8. Acknowledgments

We thank our anonymous reviewers for helping us to improve the paper. We would also like to thank David Bacon and Doug Lea for their feedback and insights, as well as the developers of Jalapeño/Jikes RVM, Moxie, and OVM, all of whom contributed directly or indirectly to the work we report here.

References

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: a dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1):19–31, Feb. 2003.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, New York, NY, USA, 2000. ACM.

[3] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *MSPC '06: Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 1–10, New York, NY, USA, 2006. ACM.

[4] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maesse, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, Mar. 2008.

[5] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 314–324, New York, NY, USA, 1999. ACM.

[6] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), Feb. 2000.

[7] Apache. DRLVM – Dynamic Runtime Layer Virtual Machine. `http://harmony.apache.org/subcomponents/drlvm/`.

[8] B. Bershad, S. M. Blackburn, H. Boehm, M. Cierniak, C. Click, D. Frampton, D. Gregg, D. Grove, X. Li, B. Mathiske, and G. Skinner. First Moxie brainstorming meeting, Dec. 2005. `http://moxie.sf.net/`.

[9] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN: an extensible microkernel for application-specific operating system services. In *EW 6: Proceedings of the 6th ACM SIGOPS European Workshop*, pages 68–71, New York, NY, USA, 1994. ACM.

[10] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *SOSP '95: Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–283, New York, NY, USA, 1995. ACM.

[11] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[12] S. M. Blackburn, S. I. Salishev, M. Danilov, O. A. Mokhovikov, A. A. Nashatyrev, P. A. Novodvorsky, V. I. Bogdanov, X. F. Li, and D. Ushakov. The Moxie JVM experience. Technical Report TR-CS-08-01, Australian National University, Department of Computer Science, Jan. 2008.

[13] H.-J. Boehm. Space efficient conservative garbage collection. In *PLDI '93: Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, New York, NY, USA, 1993. ACM.

[14] H.-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–268, New York, NY, USA, 2005. ACM.

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

[16] L. Cardelli, J. Donahue, L. Glassman, I. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). Technical Report 52, DEC SRC, Nov. 1989.

[17] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, Aug. 2007.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[19] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The Open Runtime Platform: a flexible high-performance managed runtime environment. *Concurrency and Computation: Practice and Experience*, 17(5-6):617–637, 2005.

[20] ECMA. C# Language Specification, ECMA-334. `http://www.ecma-international.org/publications/standards/Ecma-334.htm`, June 2006. (ISO/IEC 23270:2006).

[21] ECMA. Common Language Infrastructure (CLI), ECMA-335. `http://www.ecma-international.org/publications/standards/Ecma-335.htm`, June 2006. (ISO/IEC 23271:2006).

[22] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 177–190, New York, NY, USA, 2006. ACM.

[23] P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code for .NET. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages,*

and Applications, pages 329–346, New York, NY, USA, 2008. ACM.

[24] M. E. Fiuczynski, W. C. Hsieh, E. G. Sirer, P. Pardyak, and B. N. Bershad. Low-level systems programming with Modula-3. In *Threads: A Modula-3 Newsletter*, number 3, Fall 1997.

[25] C. Flack, T. Hosking, and J. Vitek. Idioms in OVM. Technical Report CSD-TR-03-017, Purdue University, 2003.

[26] J. G. Fletcher. No! High level languages should not be used to write systems software. In *ACM 75: Proceedings of the 1975 Annual Conference*, pages 209–211, New York, NY, USA, 1975. ACM.

[27] J. G. Fletcher, C. S. Badger, G. L. Boer, and G. G. Marshall. On the appropriate language for system programming. *SIGPLAN Notices*, 7(7):28–30, 1972.

[28] D. J. Frailey. Should high level languages be used to write systems software? In *ACM 75: Proceedings of the 1975 Annual Conference*, page 205, New York, NY, USA, 1975. ACM.

[29] R. Garner. JMTK: a portable memory management toolkit. Honours thesis, Australian National University, Dec. 2003.

[30] R. Garner, S. M. Blackburn, and D. Frampton. Effective prefetch for mark-sweep garbage collection. In *ISMM '07: Proceedings of the 6th International Symposium on Memory Management*, pages 43–54, New York, NY, USA, 2007. ACM.

[31] D. Gay, R. Ennals, and E. Brewer. Safe manual memory management. In *ISMM '07: Proceedings of the 6th International Symposium on Memory Management*, pages 2–14, New York, NY, USA, 2007. ACM.

[32] N. Glew, S. Triantafyllis, M. Clerniak, M. Eng, B. Lewis, and J. Stichnoth. LIL: an architecture-neutral language for virtual-machine stubs. In *VM '04: Proceedings of the 3rd Virtual Machine Research & Technology Symposium*, page 9, Berkeley, CA, USA, 2004. USENIX Association.

[33] J. Gosling, B. Joy, G. Steel, and G. Bracha. *The Java Language Specification, Third Edition*. Prentice Hall, 3rd edition, June 2005.

[34] D. Grossman. Type-safe multithreading in Cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 13–25, New York, NY, USA, 2003. ACM.

[35] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in Haskell. In *ICFP '05: Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, pages 116–128, New York, NY, USA, 2005. ACM.

[36] T. Hart and M. Levin. The new compiler. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1962.

[37] M. Hirzel and R. Grimm. Jeannie: granting Java Native Interface developers their wishes. In *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 19–38, New York, NY, USA, 2007. ACM.

[38] J. J. Horning. Yes! High level languages should be used to write systems software. In *ACM 75: Proceedings of the 1975 Annual Conference*, pages 206–208, New York, NY, USA, 1975. ACM.

[39] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid Metal: object-oriented programming across the hardware/software boundary. In *ECOOP '08: Proceedings of the 22nd European Conference on Object-Oriented Programming*, pages 76–103, Berlin, Heidelberg, 2008. Springer-Verlag.

[40] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[41] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[42] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: a safe dialect of C. In *ATEC '02: Proceedings of the General Track of the USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[43] K. Kennedy. Generation of high performance domain-specific languages from component libraries. `http://www.cs.rice.edu/~ken/Presentations/TelescopeOSU.pdf`.

[44] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Upper Saddle River, NJ, USA, 1988.

[45] S. Kuksenko. Suggestion: Let's write some small and hot native(kernel) methods on vmmagics. `http://www.mail-archive.com/dev@harmony.apache.org/msg07606.html`, May 2007.

[46] D. Lea. Low-level memory fences. `http://gee.cs.oswego.edu/dl/concurrent/dist/docs/java/util/concurrent/atomic/Fences.html`.

[47] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. The Java Series. Prentice Hall, June 1999.

[48] J.-W. Maessen, V. Sarkar, and D. Grove. Program analysis for safety guarantees in a Java virtual machine written in Java. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 62–65, New York, NY, USA, 2001. ACM.

[49] S. Microsystems. Maxine Research Project. `http://research.sun.com/projects/maxine`.

[50] J. E. B. Moss, T. Palmer, T. Richards, I. Edward K. Walters, and C. C. Weems. CISL: a class-based machine description language for co-generation of compilers and simulators. *International Journal of Parallel Programming*, 33(2):231–246, 2005.

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

[52] E. Prangsma. Why Java is practical for modern operating systems. In *Libre Software Meeting*, 2005. Presentation only. See www.jnode.org.

[53] M. Richards. BCPL: a tool for compiler writing and system programming. In *AFIPS '69 (Spring): Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, pages 557–566, New York, NY, USA, 1969. ACM.

[54] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA Companion '06: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 944–953, New York, NY, USA, 2006. ACM.

[55] D. M. Ritchie. The development of the C language. In *HOPL-II: The second ACM SIGPLAN Conference on the History of Programming Languages*, pages 201–208, New York, NY, USA, 1993. ACM.

[56] S. Ritchie. Systems programming in Java. *IEEE Micro*, 17(3):30–35, 1997.

[57] J. Shapiro. Programming language challenges in systems codes: Why systems programmers still use C, and what to do about it. In *PLOS '06: Proceedings of the 3rd Workshop on Programming Languages and Operating Systems*, page 9, New York, NY, USA, 2006. ACM.

[58] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the Squawk Java virtual machine. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 78–88, New York, NY, USA, 2006. ACM.

[59] L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, and K. Stoodley. Inlining Java native calls at runtime. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 121–131, New York, NY, USA, 2005. ACM.

[60] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[61] B. Stroustrup. A history of C++: 1979–1991. In *HOPL-II: The second ACM SIGPLAN Conference on the History of Programming Languages*, pages 271–297, New York, NY, USA, 1993. ACM.

[62] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly, 2003.

[63] A. Tridgell. Using talloc in Samba4. Technical report, Samba Team, 2004. `http://samba.org/ftp/unpacked/talloc/talloc_guide.txt`.

[64] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA Companion '05: Companion to the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 11–20, New York, NY, USA, 2005. ACM.

[65] K. Venstermans, L. Eeckhout, and K. De Bosschere. 64-bit versus 32-bit virtual machines for Java. *Software: Practice and Experience*, 36(1):1–26, 1 2006.

[66] J. R. Vernooij. SAMBA developers guide. `http://www.samba.org/samba/docs/Samba-Developers-Guide.pdf`, Apr. 2008.

[67] J. Whaley. Joeq: a virtual machine and compiler infrastructure. In *IVME '03: Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pages 58–66, New York, NY, USA, 2003. ACM.

[68] W. A. Wulf, D. B. Russell, and A. N. Habermann. BLISS: a language for systems programming. *Communcations of the ACM*, 14(12):780–790, 1971.

[69] H. Yamauchi and M. Wolczko. Writing Solaris device drivers in Java. In *PLOS '06: Proceedings of the 3rd Workshop on Programming Languages and Operating Systems*, page 3, New York, NY, USA, 2006. ACM.