The Moxie JVM Experience

Stephen M. Blackburn,[‡][*] Sergey I. Salishev,
Mikhail Danilov, Oleg A. Mokhovikov, Anton A. Nashatyrev, Peter A. Novodvorsky,
Vadim I. Bogdanov, Xiao Feng Li, Dennis Ushakov

‡ Department of Computer Science
Australian National University
Steve.Blackburn@anu.edu.au

Software Solutions Group
Intel Corporation
<Firstname>.<I>.<Lastname>@intel.com

Abstract

By January 1998, only two years after the launch of the first Java virtual machine, almost all JVMs in use today had been architected. In the nine years since, technology has advanced enormously, with respect to the underlying hardware, language implementation, and in the application domain. Although JVM technology has moved forward in leaps and bounds, basic design decisions made in the 90's has anchored JVM implementation.

The Moxie project set out to explore the question: 'How would we design a JVM from scratch knowing what we know today?' Amid the mass of design questions we faced, the tension between performance and flexibility was pervasive, persistent and problematic. In this experience paper we describe the Moxie project and its lessons, a process which began with consulting experts from industry and academia, and ended with a fully working prototype.

Categories and Subject Descriptors   D.2.10 [*Software Engineering*]: Design

General Terms   Java, Java in Java, Virtual Machine

Keywords   Moxie, JVM, Java, JiJ, Virtual Machine

1. Introduction

Most of the widely used Java virtual machines today rest on code bases which date to the first two years of the public life of the Java language. The core of IBM's very successful j9 JVM dates back to even earlier times, emerging from a high performance SmallTalk implementation. Almost a decade later, the technology within these JVMs have advanced enormously. On one hand, such success precisely vindicates the aging technical cores of these JVMs: *don't fix it if it ain't broke!* Alternatively, these technical advances highlight why we need to reevaluate the core of JVM design.

The Moxie project emerged in late 2005 as part of Intel's involvement in the Apache Harmony project, whose goal is to build a 'compatible, independent implementation of the Java SE 5 JDK under the Apache License v2' and a 'community-developed modular runtime (VM and class library) architecture' [7]. While the

nascent Harmony community bootstrapped itself with an established JVM [18], the Moxie project set out to explore the design of a next generation JVM. This, of course, first required us to figure out what exactly a 'next generation JVM' was!

On one hand, a JVM is a relatively simple system. Accomplished programmers have been known to build working JVMs single-handedly from scratch in a matter of months. On the other hand, the technology that underlies a modern high performance JVM is enormously complex, far beyond the grasp of any single person. As a small example, more than 160 research papers have been published using Jikes RVM alone. This and other work has in turn been built on the back of focused research efforts through the 80's [45] and 90's [23, 32]. Commercial JVMs build on this with patent portfolios. It would be extremely naïve to embark on a 'next generation' JVM design unless one could leverage this expertise.

Given the open setting of this project, we invited experts from industry and academia to brainstorm JVM design. We set up two meetings in late 2005 and early 2006 and used these to guide the development of a prototype. The goal of the prototype was to provide a testing ground for new ideas in JVM design that emerged from these meetings. The prototype[1] and notes from these meetings are being made available online [37] under the Apache License.

The project ran for a year and within a few months had a modest JVM working. By the end of the year, the prototype was capable of running complex workloads such as Eclipse, was ported to both IA32 and ARM architectures and could run on Linux, Windows and the L4 microkernel [34]. The prototype successfully demonstrates a number of new technologies and design ideas. In particular, it tests existing ideas for retargetable template compilers [24], includes an advanced Java-in-Java bootstrap model, implements a generalized code persistence mechanism, has pluggable object models, and includes a novel field packing algorithm for object layout.

This paper contributes to virtual machine design at three levels: 1. We examine and question the status quo, suggest the need for a second generation of JVM design, and discuss the *social factors* in the current state of affairs. 2. We describe a highly componentized *JVM design* with a publicly available prototype. 3. We have made a number of advances in *JVM implementation*, particularly with respect to techniques for Java-in-Java implementation.

The remainder of the paper is structured as follows. First we discuss related work. Then in Section 3 we discuss input from experts, perspectives on future JVM design, and the specific goals of the Moxie project. Section 4 discusses the design of the Moxie JVM. Section 5 discusses the key lessons of the Moxie project and Section 6 concludes.

---

[*] This work was conducted while the author was at Intel.

---

[1] The code is not available at the time of writing. We are committed to releasing the source under the Apache license, but the process is slow. Our hope is that the source will be available at `moxie.sf.net` by late 2007.

2. Background and Related Work

The literature on JVM design and implementation is substantial. We limit our review here to work related to three of Moxie's major themes: modular JVM design, JVM implementation in Java, and systems programming in Java.

2.1   Modular Design

Modular design increases flexibility, but often at the cost of reduced performance. Sun's EVM included a well-defined and comprehensive GC interface [29]. Performance critical aspects of the interface (such as read and write barriers) are expressed as C macros. EVM GC components were therefore only statically selectable.

The open runtime platform (ORP) [18] was designed to have pluggable compiler and garbage collector implementations. The GC and JIT interact with the VM through well defined interfaces which are implemented as C/C++ function tables. The function tables are established at boot time, which allows the GC and JIT to be boot-time configurable. In most cases, the JIT emits calls through the interface, which typically yields adequate performance. In performance critical settings, such as a check cast or write barrier, the VM or GC (respectively) must provide the JIT with code expressed in an architecture-neutral, low level intermediate language (LIL) [30]. The JIT then emits the code directly, avoiding a call through an interface. By contrast, Moxie exploits the fact that code such as a check cast or write barrier is written in Java, and is therefore automatically inlined and optimized in-situ by the optimizing compiler, following the example of Jikes RVM [3], Since Moxie depends on an ahead-of-time compiled boot-image (like Jikes RVM), component bindings are statically defined (for example, write barriers are inlined throughout the boot image, nailing down a specific choice of collector at build time).

Ovm is a framework for building language runtimes [38]. Ovm consists of statically configurable components, written in Java, which are stitched together at the time the JVM is built. The stitching process uses a custom-built component system based on a special intermediate form, OvmIR. While EVM and ORP used C macros and LIL, and Moxie and Jikes RVM depend on inlining, it is unclear how Ovm ensures performance in critical parts of the component interface. Ovm does not have an aggressive JIT, so for performance, application code can be compiled ahead of time using their Java-to-C++ compiler, for subsequent compilation by gcc/g++. C++ is used as the back-end language to leverage its built-in exception handling support. While inlining is an invaluable mechanism for the performance of Jikes RVM [5], it tended to lead to code bloat problems with the Ovm gcc/g++ back-end [14].

MMTk [15] is a self-contained component of Jikes RVM providing a framework for building garbage collectors. MMTk is shared by other JVMs, including Moxie and Ovm [26]. The performance of MMTk depends entirely on aggressive optimization of calls across the VM-memory manager interface. MMTk has been highly tuned to allow the flexibility of supporting multiple collectors within Jikes RVM without performance penalty [15]. The Moxie project raised the reciprocal concern of supporting multiple VMs from MMTk. As we discuss later, this was solved by using an abstract factory pattern [28], which the Jikes RVM optimizing compiler could optimize across with zero performance penalty. It is possible that MMTk could have utilized a component infrastructure such as Jiazzi [36], but the efficacy of the Jikes RVM optimizing compiler meant that the simple abstract factory pattern was adequate. The efficiency of the MMTk interface design, combined with the capacity to inline allocation sequences (which are all expressed in Java), means the MMTk allocation performance can match that of a highly optimized C implementation [15]. The lessons of MMTk were fundamental to, and were extended by, the Moxie project.

2.2   Java-in-Java JVMs

There is a long-standing tradition among language developers of 'eating one's own dog food' (implementing language L using the same language L) [8]. The sense of this approach depends greatly on the suitability of the language at hand to the task of language implementation. C compilers are typically written in C. The Standard ML of New Jersey compiler is written in ML. Andrew Appel has discussed in detail [8] the bootstrap problem that arises with a self-implementing language/runtime. Squeak [32] is a high performance SmallTalk VM implemented in SmallTalk.

JavaInJava [44] was a *proof-of-concept* experiment at Sun which implemented a JVM in Java. The VM was written in pure Java and executed over a host JVM, yielding slowdowns around two orders of magnitude. The author concedes that an entirely different approach would need to be taken to achieve performance comparable to a JVM written in C or C++.

Jikes RVM [4] (formerly known as Jalapeño) was developed at IBM at about the same time as JavaInJava, and serves as a *proof-of-performance* for the Java-in-Java JVM concept. Jikes RVM is almost entirely written in Java (it has a small bootstrapper and OS interface wrapper written in C). Jikes RVM includes an aggressive optimizing compiler which targets PowerPC and IA32. In February 2002, Jikes RVM achieved 95% the performance of the IBM 1.3.0 DK JVM on the SPECjvm98 benchmarks on Linux/IA32 [5], demonstrating that it is possible to build a high performance JVM in Java. While the Jikes RVM optimizing compiler makes good use of the Java language (following the dog food credo), parts of the VM core and the original garbage collectors made heavy use of static code—reflecting some of the authors' background as C coders and the relative novelty of Java at the time the code was written. The MMTk memory management subsystem [15] was written somewhat later by authors who had enormous faith in the optimizing compiler. Nonetheless, correctness concerns (such as not being able to call **new**() within the garbage collector) lead to rather stylized use of Java in such places. The Moxie project heavily leverages the Jikes RVM experience. Importantly, Moxie explicitly set out to address shortcomings in Jikes RVM as identified by longstanding members of the Jikes RVM team. These include: a much stronger focus on pluggable components; a more principled approach to extending the Java language with 'magic'; a more robust and general boot image construction method, and a generalization over code persistence required for boot image writing.

Squawk [43] is a JVM written in Java targeted at embedded devices. Squawk avoids most problems associated with Java-in-Java construction by using a limited subset of Java which maps to C. This approach allows Squawk to be compiled with a C compiler, but limits its appeal as a 'Java' implementation. Since Squawk avoids the Java runtime, it is probably not truly a Java-in-Java JVM. By contrast, the Moxie project wanted to exploit the strengths of the Java language as fully as possible and had a much broader focus as. A number of other JVMs have been written in Java, each with a different focus. These include Joeq [46] (compiler development), JNode [40] (operating systems), and Rivet [17] (testing). Ovm is also written in Java, and is discussed above.

2.3   Systems Programming In Java

The appeal of writing a JVM in Java is three-fold: a) the JVM implementation can exploit the software engineering benefits of a strongly typed, dynamically checked language, b) the impedance mismatch between JVM helper code and application code is removed, offering performance advantages [3, 15], and c) there is an aesthetic appeal and symmetry in eating one's own dog food. To successfully eat one's own dog food, the language at hand must be suitable to the task of constructing a runtime. In particular, it must be sufficiently expressive, and capable of efficiently rendering performance-critical code. This concept has been successfully

demonstrated in SmallTalk [32] and Java [3]. Nonetheless, there is a considerable cultural block to using a language other than C or C++ for systems work [42].

Unlike C#, Java does not have inbuilt mechanisms for performing tasks essential to systems programming, such as direct memory access. While the JavaInJava project [44] bypassed the problem by implementing *above* a host JVM in pure Java, Jikes RVM successfully tackled the problem head-on [3]. The original Jikes RVM approach was to add *magic* to the language in two forms: a) compiler intrinsics[2] which implemented semantics beyond the specification of the Java language (such as an address read or cache flush), and b) compiler pragmas expressed idiomatically which were used for correctness (e.g. scheduling control) and performance (e.g. inlining hints). Initially magic types such as addresses were untyped (**int** was used for addresses!). Subsequently this became more principled. In particular, the magic types became better typed with the introduction of magic types such as `Address` (akin to **void***) and `Word` (an `unsigned` **int** of the same size as the underlying architectural word). These are captured in the `org.vmmagic` package, which is used by a number of projects.

Ovm makes idiomatic use of standard Java [26] to express extensions to the standard Java semantics (such as compiler pragmas and direct memory accesses), which are necessary when implementing a JVM. The development of idioms for this purpose was largely due to Chapman Flack who drew from, extended and generalized similar ideas in Jikes RVM. Ovm idioms are iteratively evaluated during the build process and the resulting OvmIR is compiled into C++ for subsequent compilation to native code. Moxie depends on compiler intrinsics to extend Java semantics, rather than iterative translation of an IR. Moxie uses standard Java annotations rather than idiomatic use of the **throws** and **implements** keywords as Ovm does (Jikes RVM has recently adopted Moxie's approach). The Ovm approach means that all code using idiomatic Java must be compiled ahead of time, whereas with Moxie and Jikes RVM, such code can be JIT'ed, and may therefore be excluded from the initial boot image unless it is essential to the bootstrap process.

In a recent invited paper [42], Jonathan Shapiro discussed some of the reasons why the operating systems community is so attached to C, and points to concerns with managed languages and even C++. He does offer a glimmer of hope in his relatively enthusiastic view of the Singularity project [25]. Singularity has demonstrated that a strongly typed, managed language (C#) can be used in place of hardware protection to enforce interprocess isolation in an operating system, with considerable performance advantages. Nonetheless, contrary to the instinctive fears of many systems programmers (including many of our colleagues), there can be performance *advantages* associated with reducing impedance mismatch and strong typing. MMTk showed that performance-critical object allocation sequences in Java could outperform highly tuned C code [15]. The advantage came due to the inlining allowed by an absence impedance mismatch between application and allocator, and that in Java, for most allocation sites, the size of the allocated object is statically known at the call to **new**(). These factors combined to allow an aggressive compiler to highly optimize an otherwise complex free list allocation sequence. JNode [40] is an operating system written in Java. JNode uses MMTk and Jikes RVM's `org.vmmagic` package.

Moxie builds on the approach taken by Jikes RVM, and has refined some of the ideas in an effort to make *magic* more principled (some of these refinements have filtered back into the Jikes RVM code base).

---

[2] Compiler *intrinsics* are methods whose semantics are beyond the scope of the implemented language, so are implemented directly by the compiler.

3. Scoping The Project

The Moxie project emerged as a modest, forward-looking companion to what are much larger and very pragmatic engineering efforts under the emerging Apache Harmony project. In particular, the Moxie project differentiated itself as being focused on *innovation*; by developing innovative ideas, and moreover, by developing a framework to *facilitate innovation.*

The mandate of the Moxie project was to serve both production *and* research objectives. This created some significant challenges. When engineering a JVM, arguably one may chose any two of *performance, robustness,* and *flexibility* as design goals. Experience suggests that given this choice, product will sacrifice internal flexibility and research will sacrifice robustness. Of course we'd rather have our cake and eat it. The foremost challenge for Moxie therefore became engineering a system capable of all three objectives. A corollary of the product-research tension is that while innovation demands *revolution*, robustness favors *evolution*. The Moxie project will ideally allow both development styles to coexist.

Consistent with our skepticism of monolithic JVM development, we favor an *agile* development style [11]. In particular our approach was: small (for the most part, just four engineers), fast, iterative, and adaptive (shifting focus as new challenges emerged). Our aim was to *prototype* ideas rather than kid ourselves that we could deliver a finished, complete, performance-tuned JVM in a one year project.

Finally, we needed to draw on a wealth of outside expertise in order to set meaningful priorities.

3.1 Expert Input

As soon as the project had approval, we got together a group of people to have our first brainstorming meeting in San Francisco [13] in December 2005. A month later, we held a second meeting in Albuquerque [14].

3.1.1 First Moxie Meeting

The objective of the first meeting was to use brainstorming to explore in the broadest terms possible the shape of future JVMs. Eleven individuals from a range of institutions participated in the two day gathering [13]. The meeting was structured around four sessions, each of which had a single theme couched as a question. The brainstorming was driven by a series of sub-questions under each theme. Each participant wrote responses on small pieces of paper. The responses were gathered and then grouped using 'mind maps'. Details are available on the Moxie web site [13]. Here we summarize the discussions surrounding each of the four themes. These high level thoughts were intended *not* to provide specific goals for Moxie, but to help define a broad context for the Moxie project, given its aspiration to being a 'next generation' JVM.

Where will (or should) Java be in 10 years? Unsurprisingly, it was hard to reach consensus. There was a strong sense that the Java Language Specification [31] was limiting Java, and that the speed of Java's evolution was its greatest future challenge (five members were concerned it was too slow, two were concerned it may be too fast). The performance of Java was seen as a paramount ongoing challenge, not a solved problem. Java's support for native code was seen as a particular weakness.

What should a next-generation JVM look like? Many saw implementation issues as the biggest limitation of the current generation of JVMs, specifically lack of internal modularity. Most saw performance as the most important lesson of the current generation of JVMs. This topic lead to a heated discussion as to who should design and implement the next generation of JVMs. In particular there was much discussion of the difficulty of the task, the need for expertise and how compatible the task was with various open source

development models. There was some consensus that developing a product quality JVM required a much greater depth of technical expertise than is seen in most open source development projects. One of the participants stated that *building a high quality VM requires people with highly specialized knowledge & serious system programming ability. There aren't that many people with both and many of them work on commercial VMs* [13]. This seemed to capture the thoughts of many—but not all—on the panel, and raises serious challenges for a project such as Moxie and Apache Harmony. The contrary viewpoint was that this perspective was elitist (we were saying *'Rocket scientists only please!'*), and would stand in the way of community efforts to build a next generation JVM.

What needs to be done to maximize innovation? Interestingly, panel members felt by a ratio of 2:1 that *industry* rather than *academia* had been the major source of innovation. There was also a strong view that complexity of the JVM Specification and existing implementations had been the major inhibitors of innovation [35]. The corollary is that to maximize innovation, a next generation implementation must eschew complexity. Lack of adequate/realistic workloads was cited as a problem—unrealistic workloads can lead to unproductive research since many optimizations which work well on small benchmarks do not scale out to large, realistic workloads [14, 16].

What are the highest priorities for new JVM design? There was a substantial discussion of the value of hardware support for JVM implementation. Hardware assistance for control-flow (generic lightweight traps), memory management [19], and synchronization (such as the full/empty bit [6, 2]) were popular suggestions. The utility of hardware support for barriers and lightweight traps [19] was hotly debated, but many were skeptical. The group viewed non-determinism, particularly with respect to dynamic optimization, as the greatest inhibitor of robustness in modern JVMs. Specifically, there was a strong consensus that concurrency (particularly correctness with respect to concurrency) must be a first-order concern from the outset. When asked how best to deal with the tension between performance and flexibility, there was some consensus that an important strategy was using Java as an implementation language combined with aggressive optimization such as specialization [22].

The four brainstorming sessions were followed by open discussion primed by the question *"What would your priorities be if you were tasked with building a new JVM from scratch (given appropriate resources)"?* These discussions provoked questions such as how a new JVM could be designed on an Apache mailing list, and what it was about Moxie that would differentiate it from other JVMs. The following sentence emerged as a succinct summary of the Moxie project goals:

> *The Moxie project will create an open source platform for developing product-quality JVMs and an environment for JVM innovation.*

We then discussed what the objective of innovation entailed, and concluded that in a nutshell, it implies efficient modularization—a big departure from the monolithic structure of most JVMs today.

3.1.2 Second Moxie Meeting

By contrast to the first meeting, the second meeting was focused on *experience* and concrete technical issues. The meeting was anchored around four experience talks, each of which are available online [14]. The first, by Cliff Click of Azul gave a retrospective on the HotSpot compiler [39]. Dave Grove of IBM then spoke about modular object models [10]. Jan Vitek of Purdue discussed lessons of the Ovm project [38]. Marcus Lagergren of BEA finished with a discussion on the importance of a solid QA infrastructure to the JVM development process. Together, these talks gave a rich backdrop of technical insight for the project. We summarize a few of the notable discussions here.

Java-in-Java There was resounding support for implementing the JVM in Java. Many of those in attendance had experience with Java-in-Java, and others who lead development on C/C++ based JVMs argued that Java had matured enormously since their JVMs were initiated, and sufficiently that they would use Java now, if implementing from scratch. Anecdotally there remain some very experienced JVM architects who oppose the idea, but we did not hear from them.

It was clear by this stage that Moxie would be implemented in Java, which colored the discussion that followed. Discussions among those with extensive Java-in-Java experienced identified the following five challenges which had not been adequately addressed in existing Java-in-Java JVMs:

1. **Debugging** the JVM in a Java-in-Java context has typically been a weak spot, mainly because one cannot leverage existing, underlying C/C++ runtime support for debugging.

2. **Portability** of the execution engine is a greater challenge because unlike C/C++-implemented JVMs, a Java-in-Java JVM cannot achieve trivial portability via an interpreter that leverages the existing portability of the C/C++ platform.

3. **Footprint** may be an issue if Moxie wishes to target embedded applications because existing Java-in-Java JVMs have not demonstrated small footprints (Squawk [43] is a notable exception, but is does not use the Java runtime, only its syntax).

4. **Systems programming** is not supported by Java. Existing extensions [3, 26] need to be further generalized, extended and standardized.

5. **Strong isolation** between application and VM code is essential for a production JVM. The majority were skeptical that existing solutions [20, 38, 9] (in particular, isolates [20]) were practical from a performance standpoint.

6. **Clean bootstrap** is lacking from both Jikes RVM [3], which is brittle with respect to host and target class libraries, and Ovm [38], which lacks generality due to its dependence on C/C++ and lack of an optimizing JIT.

Having elected to build Moxie in Java, proof-of-concept solutions to these shortcomings became a high priority.

Robustness Three themes emerged under robustness. The first was the objective of using generators wherever possible rather than hand-coding intricate and error-prone code, such as argument shuffling. The second was the importance of reducing the size of the trusted code base (TCB). The third was the importance of establishing a quality assurance infrastructure right from the outset.

Portability As an open source project, it was important that Moxie be portable. The strong advice from the meeting was that portability be addressed right from the beginning, preferably with two or more significantly different architectures. The meeting discussed three different approaches to achieving portability: a) use a retargetable compiler such as that proposed by Ertl and Gregg [24], b) consider a tool such as VPO [33, 12], or c) simply use carefully crafted assembler—anecdotally this worked well for HotSpot [14, 39].

Components A modular design is implicit in Moxie's first-order goal of flexibility. The group discussed two distinct manifestations of components within a JVM. In the first instance, a component is a relatively well-contained module with well defined services and/or requirements with respect to the JVM core. The JIT and memory

manager are clear examples. Modularizing such components has been studied extensively [18, 38, 15]. A more challenging problem lies in components such as an object model—the subject of Dave Grove's experience talk at the meeting [10, 14]. In this case the component is cross-cutting, and so an approach such as aspect oriented programming may be more appropriate.

3.2 Project Goals

The initial goals of the Moxie project were determined by our objective of creating an '*innovation friendly*' framework for JVM implementation and our decision to implement the system in Java. We therefore decided to focus on an aggressively modular JVM architecture, and to attack the six identified short-comings of existing Java-in-Java JVMs (enumerated in Section 3.1.2).

We identified these sub-goals with respect to modularity:

- Modularize the JVM at as fine a grain as reasonable.

- Implement multiple component instances where ever possible.

- Reuse components such as MMTk [15] and Jitrino [1, 7].

- Improve over previously used approaches [15, 38].

Given their scale and complexity, we did not want to attempt to implement an optimizing compiler or non-trivial memory management subsystem in our prototype. Instead we decided to use Jikes RVM's MMTk [15] and Harmony's Jitrino [7] (which is a second-generation derivative of StarJIT [1]) and implement our own trivial alternatives: a non-collecting memory manager and a simple template based JIT. Future implementations of either component from scratch would allow us to test our capacity to simultaneously support evolution (of the existing components) and revolution (in the introduction of entirely new ones).

With respect to Java-in-Java JVM implementation, we decided to make our major focus *portability*, *clean bootstrap*, *debugging*, and *systems programming* but footprint and vm-application isolation remained important considerations.

Since Jikes RVM had established proof of performance [5], and we would not have access to an optimizing compiler until later in the project when Jitrino was ported, performance was *not* an objective for our prototype. Our position that performance should not be a first-order concern for Moxie has been borne out in ongoing performance comparisons of Jikes RVM and DRLVM [21]. These results show that Jikes RVM, the java-in-java VM, consistently continues to outperform its cousin, DRLVM, written in C++. This is an interesting comparison since both projects have similar goals, similar roots in the late 1990's as research JVMs and both have advanced JIT compilers. Nonetheless, we did perform some targeted performance analysis. For example, we were able to evaluate the performance of our application of the Abstract Factory pattern by experimenting with MMTk in Jikes RVM.

A third, longer-term goal of the project was to explore the issue of redundancy between operating system and JVM services such as memory managers and schedulers. To that end, we decided to port the JVM to the L4 microkernel [34] at the outset. A port to a microkernel would allow us considerable flexibility with respect to the provision of such key OS services.

Thus toward the end of 2005, we embarked on a year long project to build a highly modular JVM in Java armed with a substantial list of goals based on input from experts and perceived shortcomings of existing work. To re-iterate, our notable *non-goals* were proof of performance [5] and the implementation of an optimizing compiler [1] or non-trivial management system [15]. Or notable *goals* were to implement a highly modular Java-in-Java JVM, ported to at least two architectures and two operating systems, with an improved bootstrap mechanism and that as far as possible attacked the above list of Java-in-Java shortcomings.

4. Moxie Design and Implementation

We now describe the design and implementation of the Moxie prototype. Complete source code is being made publicly available [37] under the Apache Public License v2.

4.1 Design

The design of Moxie is shaped by the decision to implement in Java and to achieve maximal modularity. We now describe the component model we used, and then briefly describe each of the major components.

4.1.1 Component Model

MMTk had previously demonstrated modularity of JVM components without sacrificing performance [15]. Rather than statically define a single, simple interface between all memory managers (MM) and virtual machines (VM), MMTk assumes each will define its own set of *services* and *requirements*. Then for any particular MM, VM pairing (mm_i, vm_j), glue is written in the form of two 'wedges', $mmvm_{ij}$ and $vmmm_{ji}$, where a 'wedge' maps requirements onto services as optimally as possible given the specifics of mm_i and vm_j. This approach has a number of properties. First, it avoids the lowest common denominator effect of reducing all MM, VM pairings to use a single, global, statically defined interface. This property is particularly important given our objective of flexibility. Second, in the Java-in-Java setting, if the binding is statically determined, it allows an optimizing compiler to optimize across the glue code. This property lead to the extremely efficient implementations reported for MMTk [15].

In MMTk, requirements were stated simply in the form of the static methods of a set of interface classes. MMTk provided 'stub' instances of these classes, and peer VMs provided concrete instances which replace the stubs. Although this approach worked reasonably well in the context of MMTk and Jikes RVM, it is brittle, as there is no way to ensure coherence between the distinct implementations of the static interfaces—they are simply expected to alias each other correctly. Moxie applied components very aggressively and soon this brittleness became unacceptable. So we explored alternative approaches.

```
1   public static final MMInterface mmInterface;
2
3   static {
4     try {
5       String name = System.getProperty("moxie.mm.factory");
6       Class cl = Class.forName(name);
7       MMFactory mmf = (MMFactory) cl.newInstance();
8       mmInterface = mmf.newMMInterface();
9     } catch (Exception e) {
10      throw new RuntimeException(e);
11    }
12  }
```

Figure 1. Using the Abstract Factory Pattern To Glue Components

Eventually we established that if carefully applied, the abstract factory pattern [28] could be used without any loss of performance. This design pattern uses an abstractly defined interface to avoid the problem of brittleness described above. For example, MMTk's requirements of the VM are defined in a series of abstract classes in the package `org.mmtk.vm`. Any VM peer must offer an interface that extends these classes. Singleton instances are used to avoid static methods, and the final types of these instances (which pins down the specific VM) is defined at build time. An example of this is illustrated in Figure 1, where the core component establishes a concrete instance of a memory manager at initialization time (binding a specific memory manager to the core). Since the singletons

are declared `final`, an aggressive optimizing compiler can resolve the final types and inline the calls at will. We carefully evaluated this approach in the context of Jikes RVM (which has an aggressive optimizing compiler and uses MMTk), and found that there was no loss of performance. We then used this pattern throughout Moxie. The abstract factory method was then adopted by Jikes RVM for its interface with MMTk (in about July of 2006). We see the example of the abstract factory pattern as important, because it is *so* far removed from orthodox systems programming. Yet it provides great flexibility, and with suitably powerful tools, can perform optimally.

4.1.2 Component Structure

Each of the Moxie components, except the core, are pluggable and in most cases may have multiple different pluggable implementations. Some components have sub-components. The Moxie JVM currently comprises the following five top-level components.

Core The core includes the class loader, runtime class representation, scheduler, JNI, JVMTI, object model and utilities. These sub-components should each be pluggable, but due to time pressure we were only able to make the object model a pluggable sub-component with distinct instances (see below). Arguably, the cross-cutting nature of the object model made it the most challenging, and therefore the best choice for proof-of-concept.

Execution Engine The execution engine abstracts over code execution, code generation, code storage, the activation stack, and exception throwing and handling. Thus an execution engine instance may simply correspond to a trivial JIT (or interpreter), or to an adaptive hot-spot compilation system with multiple JITs and optimization levels. Thus the core works as a container for code providers and JITs implementing these services. Currently Moxie has two executable code storage modes: in-memory and persistent via a code cache. Two JITs are currently supported, our retargetable template-based compiler and Harmony Jitrino [7, 1]. The implementation of the retargetable compiler and code cache are discussed below. An important property of the retargetable compiler is that it generates relocatable code, which is necessary for our code persistence mechanism. Since Harmony Jitrino cannot generate relocatable code, we are unable to use it for code caching.

Memory Manager The memory manager encapsulates allocation and garbage collection. Initially Moxie had a trivial bump-allocate, non-collecting memory manager. Subsequently MMTk [15] was added. Currently Moxie only utilizes the MarkSweep MMTk configuration due to Moxie's requirements for pinning.

OS Porting Layer The OS porting layer abstracts over OS system calls and primitives such as synchronization, thread creation, console IO, file IO, and dynamic library loading. This layer is written in Java and interacts directly with the OS APIs via our native interface (Section 4.2.3). We have two instances of this component; for Windows and POSIX. By contrast, Jikes RVM's OS adapting layer is the largest non-Java component of the runtime [3].

Bytecode Verifier The Moxie bytecode verifier is a pluggable component. It is designed to be fully independent of the VM implementation. We currently have one instance of this component, which is a complete Java port of the Harmony bytecode verifier, which was written in C. It took one engineer about one month to complete the port.

In addition to describing the five major components of the Moxie prototype, we will briefly describe the object model sub-component of the core component, since the object model is the pervasive and systemic [10].

Object Model Moxie currently supports two different object models: 'plain' and 'segregated'. The plain object model is similar to the Jikes RVM standard object model [3, 10]. The object header is at a negative offset from the object references and scalar fields and array elements are at positive offsets. Our field packing algorithm described in the Appendix guarantees no storage overhead for machine word-aligned objects. The 'segregated' object model lays out scalar objects bi-directionally. Reference fields are at negative offsets to the object header, and primitive fields are at positive offsets. Our field packing algorithm may be applied to the scalar part of the object (the reference part is always dense). This model was proposed by SableVM [27], with the rationale that it improved locality and simplified object scanning at GC time by avoiding the need for a type lookup when scanning an object. We chose to implement this model simply because it is so different to our 'plain' model, and therefore provides a good proof-of-concept.

4.1.3 Component Unit Testing

A fundamental aspect of the Moxie design strategy is systematic unit testing of components. Unit testing is essential to stability and agility [11]. While many components are normal Java classes, and are therefore easy to write tests for, others such as JIT compiled code, the scheduler, etc, require the target execution environment in order to perform the test. The bootstrap process (Section 4.2.1) was designed to achieve this and maintain testing modularity. We use automatic test control flow analysis to restrict compilation and data copying to those required for a particular test. Thus there is no additional burden on the developer. This represents a significant improvement over the previous state of the art in Java-in-Java JVM testing and debugging [3, 38].

4.2 Implementation

We now describe some features of the Moxie implementation.

4.2.1 Bootstrap Process

There is always a bootstrap problem when a language implements itself [8]. Java-in-Java JVMs typically ahead-of-time (AOT) compile a core 'boot image' into an executable binary image, and then load and execute that image at runtime [3, 38]. In the case of Jikes RVM and Moxie, the JIT compiler has the additional role of performing this AOT compilation. The image must contain both code and data for the core classes necessary for bootstrap. Image construction is performed by executing relevant parts of the JVM within a host and then using a reflective mechanism to move the relevant code and data into a persistent image. Moxie supports three modes of execution:

1. **Hosted Mode**. In this mode, bytecode is executed in a host JVM (not natively). The Moxie class loader, compiler, and some utility classes support execution in this mode. This mode is intended for testing and building the boot image.

2. **Hosted Target Mode**. This mode is accessible through hosted mode. It is primarily intended for testing. To prepare the code for running in this mode, the hosting application allocates (native) memory for a target JVM heap and compiled code storage. Then it executes the Moxie compiler within the host to compile necessary code and it copies objects to the target heap. After this step, the allocated memory contains a binary image of the code. The hosting application then can save the memory as the bootimage or directly transfer the control to the target's entry point. This allows building a test image, running it in target environment, and then analyzing the results of execution, all from within a host JVM. This has a number of debugging advantages, including isolating the possibility that the correctness of Moxie compiler's execution might be recursively affected by its own bugs—when running on a host JVM this is ruled out. Neither Jikes RVM or Ovm support such a mode [3, 38].

3. **Target Mode** (Self-Hosted). In this mode Moxie is restored from a bootimage by a simple native image loader and run as native application.

It is possible to include arbitrary classes in the boot image as long as the necessary classes are loaded and requisite instances created at the time boot image creation starts. To create an image, two distinct procedures are used:

1. **Instance Migration**. This process is similar to serialization. We start with a set of seed objects and use reflection to copy the transitive closure of these seeds into the target heap.

2. **Kernel Compiling**. This process builds a conservative set of instantiated classes and called methods. We start with the set of classes whose instances are copied during instance migration and a predefined list of entry point methods. The following rules are then applied iteratively to create a transitive closure:

   (a) Each new class is scanned for explicitly annotated entry point methods;

   (b) Each new virtual method is checked for overrides against the existing list of potentially instantiated classes;

   (c) Each new method is conservatively analyzed for method calls and object instantiation;

   (d) Each newly identified potentially instantiated class is checked for overrides against the existing set of potentially called virtual methods.

Together, these guarantee that the bootimage will contain all code necessary for execution. Note that unlike Jikes RVM [3], entry points are identified in place (via annotations) rather than via an explicitly enumerated list. This approach is more transparent and therefore less error prone.

4.2.2 Systems Programming

Moxie began with the `org.vmmagic` package of unboxed types and compiler pragmas used by MMTk and Jikes RVM. Two notable improvements were made. First, Java 1.5 annotations were used to express pragmas more naturally than idiomatic use of **throws** and **implements** [3, 26]. Second, compiler intrinsics were made less opaque by using annotations to a) identify them as intrinsic, and b) specify the intended semantics via a handle. In the example in Fig-

```
1  package org.vmmagic.unboxed;
2
3  public abstract class Address {
4    ...
5    @MagicMethod(ADDRESS_STORE_CHAR_OFFSET)
6    public void store(char value, Offset offset) {
7        ...
8    }
9    ...
10 }
```

Figure 2. Making Intrinsic Methods Less 'Magic'

ure 2, line 5 annotates the `store()` method of `Address` to indicate that it is intrinsic and to specify that its semantics are defined by the handle `ADDRESS_STORE_CHAR_OFFSET`. The rest of the code in the example appears just as it would in Jikes RVM or MMTk. In addition to increasing transparency for the user of `Address`, this approach means that the compiler implements exactly the set of defined intrinsic semantics, allowing unimplemented intrinsic methods to be discovered by the compiler rather than via spurious behavior. Jikes RVM has since adopted both of these innovations in its `org.vmmagic` package.

4.2.3 The Native Interface

Java-in-Java JVMs must be able to make low-cost calls to certain native code, so that they can access OS services, for example. The

problem is even more acute in Moxie, given our decision to use the Jitrino optimizing compiler, which is written in C/C++. Our dependence on efficient native calls was consequently significantly greater than, for example, in Jikes RVM. In such settings, JNI is infeasible as it is too heavyweight, both in performance and in requiring native wrappers. Jikes RVM solves the problem with its own native call mechanism. The system is semi-automated, but limited, not fully general and it requires stubs to be implemented in C code (which then call through to the intended library call).

```
1  /*
2   * JITEXPORT void JIT_init(JIT_Handle jit,
3   *                         const char* name);
4   */
5  @ExternC
6  static native void JIT_init(Compiler jit, Address name);
```

Figure 3. Native Interfaces Using Annotations

Given the use of Jitrino, the decision to write the OS portability layer in Java, and the unsuitability of JNI, it was essential that Moxie have generic support for efficient native calls. There are three major concerns with native calls: linking, marshaling and pinning. Moxie addresses these as follows:

- **Linking**. Moxie uses annotations on Java **native** methods to describe the calling convention (see line 5 of Figure 3). Moxie lazily resolves each method at runtime, performing its own dynamic linking via the OS layer. Declarations such as the one in Figure 3 can be automatically generated by mapping `.h` files Java classes populated with such declarations.

- **Marshaling**. The `org.vmmagic` unboxed types are used for parameter types. So any pointer (**void** `*`) is described as an `Address`, a platform-sized **int** is a `Word` or `Offset` depending on whether signed. Fixed size numeric types are cast to corresponding Java primitive types.

- **Pinning**. Unless JNI (or some equivalent form of pointer indirection) is used, any object passed to a native method must first be pinned. Otherwise the object may be moved by a copying garbage collector while the native code is referencing it. Moxie uses MMTk to implement a trivial form of pinning. Currently this is done by forcing such objects to be allocated into a non-moving object space in MMTk. Moxie requires that the caller of the native method retain a reference to any passed object to ensure that that object is not collected during the call. Memory required for native data structures is allocated as Java arrays.

This approach does not address the problem of type safety of native calls. Since all pointers are effectively cast to **void**`*` and there are no composite native data types on Java side, programming to this interface requires a discipline which is hard to achieve when the native interface is 100 or more functions wide. The resulting bugs can be hideous, manifesting as hard to pinpoint crashes. Consequently, interacting with external native components was the most difficult and time-consuming part of working with Moxie. General support for unboxing of compound types in `org.vmmagic` might alleviate this problem.

4.2.4 A Retargetable Template Based JIT

Recall that portability was a high priority for Moxie. So although we could have quickly built a trivial non-optimizing JIT, we decided to evaluate a promising idea for retargetable JITs which was recently published by Ertl and Gregg [24]. Following their approach, our compiler consists of two parts: a template-based, platform-independent compiler framework written in Java, and a generator capable of generating platform-specific templates. Templates consist of: a) opaque code (arrays of bytes), and b) patching

procedures for adjusting addresses and offsets in the code. We evaluated this system by porting to IA32 and ARM at the outset.

The compiler framework was very portable. Porting to new ISAs only required the generation of new templates. Unfortunately we found that template generation was not as easy as we had hoped. The basic idea is that a pre-existing port to the target ISA (such as gcc) could be leveraged to automate the generation of templates, exploiting the 'labels as pointers' language extension supported by gcc [24]. Suitably written templates would allow a parser to extract sufficient information from the compiled code to produce stitchable code snippets and patching procedures.

The performance of this approach depended on the C compiler performing some optimizations. However, the patching process was brittle to many optimizations. For example, an optimizing compiler might substitute an addition of a constant with a subtraction of the compliment of the constant. The result is hard if not impossible for the template generator's parser to identify. While it is possible to modify the template generator to understand such optimizations, there are many possible optimizations to consider (and the compiler in question is likely to be a moving target).

The problem arose for both IA32 and ARM. On the ARM, the addition of a 32-bit constant results in four add immediate operations if the constant occupies all 32 bits (one add immediate operation can operate with a register and an 8-bit integer). However, when the implementation of the template for the `addi` bytecode adds a 32-bit integer to the register variable, the compiler cleverly generates a minimal number of ARM `addi` instructions, optimized with shifts when possible. However, our compiler should be able to compile `addi` with an arbitrary 32-bit integer, so we need precisely four ARM `addi` instructions. The consequence was that we ended up hand writing every addition within templates in assembly, losing all opportunity for optimization.

We also had problems using the C `goto` statement as a template separator as suggested [24]. The extensive padding required in branching templates on the ARM meant that the jump had to be very long. Padding was implemented as an assembly instruction and the compiler couldn't use that information, resulting in longer jumps than the ARM ISA supports. We ended up using `nop` instructions as template separators. We also note that the template approach breaks down in cases where the C compiler simply calls to intrinsic functions in the C runtime. This was a problem for a number bytecodes such as `f2i`, `ldiv`, `lrem`, etc. We had to conclude that templates need to be substantially tuned to the specific platform, compiler and compiler version and in our experience, this work is non-trivial.

We conducted some rudimentary performance testing of the compiler. The compiled code ran at about half the speed of that produced by Apache Harmony's 'JET' non-optimizing JIT compiler [7], but much faster than interpretation in a production JVM.

4.2.5 Code Cache

We implemented a code cache for three reasons: a) as a generalization over the ahead-of-time compilation which must occur during boot image writing for a java-in-java JVM, b) to improve start-up time for cached applications, and c) to support code preemption, which may be useful in a memory-constrained context. A general implementation of code caching is non-trivial [41]. Our prototype implementation is persistent, performs relocation, supports preemption, and conducts primitive validity checks via checksums.

Lack of support for code relocation in the Jitrino optimizing compiler [7, 1] meant that the code cache could only be applied to baseline compiled code. The code cache will be most effective when the quality of the retrieved code is high and the cost of regenerating that code is high. Without an optimizing compiler, neither are likely to be true. Nonetheless, we found that the code cache could produce code on average around seven times faster than the baseline compiler. However, this only lead to a 10% improvement in total startup time. We found that this was because the cost of class loading and other overheads dominate method compilation when using the baseline compiler.

5. Status and Lessons

With a dedicated team of four to five engineers,[3] we were able to rapidly build a functional prototype from scratch. It took about three months to get a basic JVM running, although in retrospect (too) much of this time was devoted to getting the retargetable compiler working (without which we could execute nothing). Within a year we had integrated MMTk and Jitrino, had OS ports to Windows, Linux and L4 and a baseline compiler port to IA32 and ARM. The prototype JVM was capable of running substantial, complex applications such as the Eclipse IDE. We had practically demonstrated aggressive modularity, and improved on previous techniques. We had a complete bytecode verifier, a novel boot image building mechanism, useful extensions to the systems programming model, and an elegant native interface. We felt that we had advanced the state of the art and created a good basis for next generation JVM development.

Agile Development The principles of agile development [11] are very well suited to this technically complex task. We had a small, tightly integrated, dedicated team of highly experienced engineers following the agile style. Although our high-level goals were clear, our specific objectives were very fluid and subject to feedback as the project progressed. Our focus was on rapid, iterative prototype development and putting specific ideas to the test. We *felt* very productive.

Pluggable Components Our focus on modularization is consistent with agile development and with our desire to promote innovation and both evolutionary and revolutionary development styles. We found that the 'wedge' approach used by MMTk [15] did not scale well due to the brittleness of statically defined, unchecked interfaces. The abstract factory pattern solved the problem for us and we were delighted that an aggressive compiler [5] was able to successfully devirtualize calls through the interface, yielding us safe modularity at no penalty. Thus we considered our experiment with extremely modular JVM design very successful.

Unit Tests Rigorous unit testing of components is fundamental to the Moxie design and implementation strategy. We found that it was invaluable to stability and agility, and therefore crucial to our rapid development style. The difficulty of testing elements of the runtime such as the JIT and scheduler led us to develop the *hosted target* execution mode for harnessing components within a host while executing natively. We believe the effort and discipline associated with systemic unit testing paid off many times over.

Java in Java Construction Given prior positive experience with Jikes RVM and MMTk, and strong endorsement from the experts we consulted, we were determined to push Java-in-Java implementation as hard as we could. Of the six Java-in-Java shortcomings we identify in Section 3.1.2, Moxie made major progress on four, and generated insight into a fifth. Given the various priorities we were juggling, we did not find sufficient time to seriously explore the JVM footprint (item 3). We felt that we made substantial progress with respect to the challenges of debugging, systems programming and clean bootstrap (items 1, 4, and 6). In each case we developed novel techniques and applied them extensively. We explored architectural portability (item 2) at length although the finding was neutral. A future generalization of our hosted-target execution mode described below would lead to OS-like VM/application isolation (item 5).

---

[3] All engineers were expert Java programmers, but none had prior JVM implementation experience due to our desire for a clean-room implementation.

Native Interfaces　The implementation of the OS portability layer in Java and our decision to integrate a significant native component (the Jitrino JIT) meant we had to take seriously the implementation of native calls. We are pleased with the resulting mechanism, which is general, transparent, and can be automated. The weak link in the chain is that we have no way to type compound types passed across the interface, but must throw away typing and resort to a '`void *`'. Extending `org.vmmagic`'s unboxing to support compound types would complete an otherwise compelling picture.

Bootstrap Process　The Moxie JVM improves over previous Java-in-Java bootstrap techniques in three ways. First, it offers a 'hosted-target' mode which is invaluable to testing and debugging. Second, Moxie uses a more transparent image creation algorithm. Finally, it generalizes over boot-time compilation with a generic code persistence mechanism which also supports code preemption. As future work, it would be very interesting to see if the hosted-target mode could be further generalized to the extent that *all* JVM services could run hosted, while the application ran entirely natively. This would provide a very rich JVM debugging environment and a limit-study in VM/application isolation. Under such a model, all transitions from application to JVM services would become explicit at some point (like an operating systems trap). In the extreme, the application could run on a host remote to the VM services. The hosted VM services would forgo the use of compiler intrinsics (since the host JVM is not obliged to support them), so services such as garbage collection would be significantly slowed down. Nonetheless, it would provide a debugging environment which for current Java-in-Java developers exists only in their dreams.

Retargetable Compiler　We took our goal of portability seriously, so we persisted with the retargetable template-based compiler despite frustrations. Nonetheless, we were successful in implementing an effective compiler with ports to IA32 and ARM. In retrospect our experience was that the holy grail of retargetable compilation still has a way to go. Given the anecdotal evidence of achieving portability through careful (orthodox) design [14, 39], the more orthodox route to portability would have been more pragmatic. Nonetheless, we feel that our experience was useful and provocative—after all, the goal of the Moxie project was to put new ideas to the test.

6.　Conclusion

The Moxie project set out to pragmatically address the question 'How would we design a JVM from scratch knowing what we know today?' We started by drawing on expertise from industrial and academic practitioners through two meetings [13, 14], and used that input to drive a year-long prototype development project. By the end of the project, we had a working prototype JVM capable of running substantial workloads, and which concretely demonstrated most of our goals including flexibility, portability and internal modularity.

The project demonstrated that a highly modular, flexible JVM was attainable using technology which did not sacrifice performance. In doing so, we validated the agile programming paradigm, making great use of modularity, unit tests, a small team, and fluid, feedback-directed goals. We advanced the state of the art in Java-in-Java JVM construction, addressing longstanding shortcomings of previous approaches and using the Java language more naturally and fearlessly than before. Many of our insights and ideas have already made their way into other JVMs including Harmony [7] and Jikes RVM [3].

We thoroughly enjoyed building Moxie and look forward to making the source code public. We hope that the JVM development community will benefit from our experience.

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] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: architecture and performance. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 1995. ACM Press.

[3] 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.

[4] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, Denver, CO, Nov. 1999.

[5] B. Alpern, M. Butrico, A. Cocchi, J. Dolby, S. J. Fink, D. Grove, and T. Ngo. Experiences porting the Jikes RVM to Linux/IA32. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 51–64, Berkeley, CA, USA, 2002. USENIX Association.

[6] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *ICS '90: Proceedings of the 4th International Conference on Supercomputing*, pages 1–6, New York, NY, USA, 1990. ACM Press.

[7] Apache. Apache Harmony, 2006. http://harmony.apache.org/.

[8] A. W. Appel. Axiomatic bootstrapping: a guide for compiler hackers. *ACM Trans. Program. Lang. Syst.*, 16(6):1699–1718, 1994.

[9] G. Back and W. C. Hsieh. The KaffeOS Java runtime system. *ACM Trans. Program. Lang. Syst.*, 27(4):583–630, 2005.

[10] D. F. Bacon, S. J. Fink, and D. Grove. Space- and time-efficient implementation of the Java object model. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 111–132, London, UK, 2002. Springer-Verlag.

[11] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development, 2001. http://agilemanifesto.org/.

[12] M. E. Benitez and J. W. Davidson. Target-specific global code improvement: Principles and applications. Technical report, Charlottesville, VA, USA, 1994.

[13] 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/.

[14] S. M. Blackburn, H. Boehm, M. Cierniak, C. Click, D. Grove, M. Hermenegildo, T. Hosking, K. S. McKinley, J. E. B. Moss, M. Lagergren, S. Srinivas, D. Stefaovic, J. Vitek, G. Wright, and M. Wolczko. Second Moxie brainstorming meeting, Jan. 2006. http://moxie.sf.net/.

[15] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Scotland, UK, May 2004.

[16] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.

[17] D. Bruening and J. Chapin. Systematic testing of multithreaded programs. Technical Report MIT-LCS-TM-607, MIT, May 2000.

[18] M. Cierniak, B. T. Lewis, and J. M. Stichnoth. Open runtime platform: flexibility with performance using interfaces. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 156–164, New York, NY, USA, 2002. ACM Press.

[19] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 46–56, New York, NY, USA, 2005. ACM Press.

[20] G. Czajkowski, L. Daynès, and B. Titzer. A multi-user virtual machine. In *USENIX 2003 Annual Technical Conference, San Antonio, TX*, pages 85–98, Berkeley, CA, 2003. USENIX Association.

[21] DaCapo Consortium. DaCapo benchmark performance comparison, 2007. http://cs.anu.edu.au/people/Robin.Garner/dacapo/regression/.

[22] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 93–102, New York, NY, USA, 1995. ACM Press.

[23] J. Dean, G. DeFouw, D. Grove, V. Litinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 83–100, San Jose, CA, Oct. 1996.

[24] M. A. Ertl and D. Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 41–50, Washington, DC, USA, 2004. IEEE Computer Society.

[25] 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 *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 177–190, 2006.

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

[27] E. M. Gagnon and L. J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Java Virtual Machine Research and Technology Symposium*, pages 27–40. USENIX, 2001.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[29] A. Garthwaite and D. White. The GC interface in the EVM. Technical report, Mountain View, CA, USA, 1998.

[30] N. Glew, S. Triantafyllis, M. Cierniak, M. Eng, B. T. Lewis, and J. M. Stichnoth. LIL: An architecture-neutral language for virtual-machine stubs. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium, May 6-7, 2004, San Jose, CA, USA*, pages 111–125, 2004.

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

[32] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of squeak, a practical smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326, New York, NY, USA, 1997. ACM Press.

[33] P. Kulkarni, W. Zhao, S. Hines, D. Whalley, X. Yuan, R. van Engelen, K. Gallivan, J. Hiser, J. Davidson, B. Cai, M. Bailey, H. Moon, K. Cho, and Y. Paek. VISTA: VPO interactive system for tuning applications. *Trans. on Embedded Computing Sys.*, 5(4):819–863, 2006.

[34] B. Leslie and G. Heiser. Iguana/L4, 2006. http://ertos.nicta.com.au/research/l4/.

[35] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Prentice Hall, 2rd edition, 1999.

[36] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: new-age components for old-fasioned Java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 211–222, New York, NY, USA, 2001. ACM Press.

[37] Moxie. The Moxie Project, 2006. http://moxie.sf.org/.

[38] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a common intermediate representation for the Ovm framework. *Science of Computer Programming*, 57(3):357–378, 2005.

[39] M. Paleczny, C. A. Vick, and C. Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium*. USENIX, 2001.

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

[41] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver: a quasi-static compiler for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 66–82, New York, NY, USA, 2000. ACM Press.

[42] 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 Press.

[43] N. Shaylor, D. N. Simon, and W. R. Bush. A Java Virtual Machine architecture for very small devices. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 34–41, New York, NY, USA, 2003. ACM Press.

[44] A. Taivalsaari. Implementing a Java virtual machine in the Java programming language. Technical Report SMLI TR-98-64, Sun Microsystems, Mountain View, CA, USA, 1998.

[45] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[46] 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 Press.

Appendix: The Moxie Field Packing Algorithm

The Moxie field packing algorithm lays out object fields of size 2^n, aligned on either the field size or machine word boundary, whichever is smaller. The fields are allocated in order with gaps immediately filled if a suitable field is encountered later. For each potential field alignment the algorithm tracks an insertion point. The insertion point is the smallest offset in the object layout a field of that alignment can be inserted. By definition there is exactly one insertion point for each field alignment. Each new field is allocated at the insertion point corresponding to the field's alignment, then all insertion points are updated accordingly.

```
1   /**
2    * Allocates a new field with machine word alignment
3    * or size alignment whichever is smaller. Offsets
4    * array represent the insertion points for elements
5    * of each alignment. The index in array is the log2
6    * of the alignment size.
7    * @param size of new field
8    * @param offsets insertion points for each alignment
9    * @return the offset of new field
10   */
11  public static int allocateField(int size,
12                                  int[] offsets) {
13    final int log2size = log2(size);
14    final int base = min(offsets.length - 1, log2size);
15    int res = offsets[base];
16    for (int i = base; i < offsets.length - 1; i++) {
17      if (offsets[i] == res) {
18        offsets[i] = max(offsets[i] +
19          (1 << i), offsets[i + 1]) ;
20      }
21    }
22    if (offsets[offsets.length - 1] == res) {
23      offsets[offsets.length - 1] +=
24        max(1 << (offsets.length - 1), size);
25    }
26    for (int i = 0; i < base; i++) {
27      if (offsets[i] == res) {
28        offsets[i] = offsets[base];
29      }
30    }
31    return res;
32  }
```

Figure 4. The Moxie Field Packing Algorithm

A gap is continuous unused space in the layout. By definition, there is always a gap after the last allocated field. A new gap is created only if there is no existing gap with the same start alignment as the field about to be allocated. If there were, the insertion point for corresponding alignment would point to the start of that gap. After the allocation the insertion point will point to the start of the new gap. So the number of gaps is smaller than or equal to the number of insertion points. The size of each gap except the last is less than machine word size. Each insertion point has an offset less than or equal to the offset for any insertion point of larger alignment. So the total size of all gaps except the last one is less than machine word size. We can assume that the layout ends on the machine word bound, in the other case we can add and remove a dummy field to the end of layout. So for machine word aligned objects the algorithm creates no storage overhead.

The information about insertion points is maintained across the class inheritance. So a subclass can fill the gaps in its super-class object layout. The algorithm allows allocating new fields without a hierarchical lookup to the previously allocated fields. Figure 4 shows an implementation of the field packing algorithm.