Maxine: An Approachable Virtual Machine For, and In, Java

CHRISTIAN WIMMER, MICHAEL HAUPT, MICHAEL L. VAN DE VANTER,
MICK JORDAN, LAURENT DAYNÈS, and DOUGLAS SIMON, Oracle Labs

A highly productive platform accelerates the production of research results. The design of a Virtual Machine (VM) written in the Java™ programming language can be simplified through exploitation of interfaces, type and memory safety, automated memory management (garbage collection), exception handling, and reflection. Moreover, modern Java IDEs offer time-saving features such as refactoring, auto-completion, and code navigation. Finally, Java annotations enable compiler extensions for low-level "systems programming" while retaining IDE compatibility. These techniques collectively make complex system software more "approachable" than has been typical in the past.

The Maxine VM, a metacircular Java VM implementation, has aggressively used these features since its inception. A co-designed companion tool, the Maxine Inspector, offers integrated debugging and visualization of all aspects of the VM's runtime state. The Inspector's implementation exploits advanced Java language features, embodies intimate knowledge of the VM's design, and even reuses a significant amount of VM code directly. These characteristics make Maxine a highly approachable VM research platform and a productive basis for research and teaching.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Run-time environments, Debuggers, Memory management, Compilers, Optimization*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Java, Maxine VM, metacircular VM, tool support, inspector, debugging, VM design, VM interfaces

ACM Reference Format:
Wimmer, C., Haupt, M., Van De Vanter, M. L., Jordan, M., Daynes, L., and Simon, D. 2013. Maxine: An approachable virtual machine for, and in, Java. ACM Trans. Architec. Code Optim. 9, 4, Article 30 (January 2013), 24 pages.
DOI = 10.1145/2400682.2400689 http://doi.acm.org/10.1145/2400682.2400689

1. INTRODUCTION

Research in a high-level language Virtual Machine (VM) inevitably involves dealing with complex system software. Adding a feature often requires modifying and extending code in numerous locations, unveiling intricate dependencies among parts of the system that surface only through crashes with hard-to-identify causes. VM and compiler research typically addresses performance and neglects the productivity aspect. The goal of the Maxine project [Oracle 2012d] is to create a platform that supports effective and highly efficient research in VM technology with as few distractions as possible.

We believe that the Maxine VM is highly *approachable* in many senses: barriers to entry are reduced, it is coded entirely at a high level of abstraction in Java™, alternate implementations of significant subsystems can be "plugged in", and extensive

Authors' addresses: C. Wimmer (corresponding author), M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, Oracle Labs, 500 Oracle Parkway, Redwood Shores, CA 94065; email: christian.wimmer@oracle.com.

DOI 10.1145/2400682.2400689 http://doi.acm.org/10.1145/2400682.2400689

tool support is provided. The code base, including tools, is freely available as open source from Oracle [2012c], has a simple build process, and is supported on multiple platforms. Maxine's architecture is modular, in particular with respect to heap management, object layout, synchronization, and compilers, and it demonstrates "systems programming in Java".

Several strategies guide Maxine's design and implementation. The VM is written almost entirely in a single high-level language and aggressively leverages modern abstraction mechanisms. The VM's architecture is modular, with decoupled components (known as *schemes*) used both to implement significant subsystems and to specify target platform characteristics. The design is metacircular: runtime state is represented uniformly as objects, and certain code generation specifications are described using Java.

Compiler extensions enable low-level programming that is otherwise unsupported in Java. The VM is compatible with an otherwise unmodified JDK, with nearly complete coverage of the Java language and VM specification; support for the new Java 7 bytecodes is underway. The entire code base is compatible with standard Java IDEs for editing, building, and advanced services such as refactoring. Extensive, specialized tools (co-developed with the VM) leverage advanced features of Java and share code with the VM. A specialized visualizer/debugger, the Maxine Inspector, displays the entire runtime state of the VM at multiple levels of abstraction.

The Maxine VM shares many ideas with other metacircular Java VMs, such as the *Jikes Research VM* [Rogers and Grove 2009], *OVM* [Palacz et al. 2005], or the *Moxie VM* [Blackburn et al. 2008]. Jikes is probably the best-known of these in the research community, with several hundred research papers based on it. The general structure of the Maxine VM and the Jikes RVM are comparable; both have major components such as the just-in-time compiler and garbage collector written in Java, use their own compiler to create a boot image, and provide a framework for low-level programming in a high-level language [Frampton et al. 2009]. We believe that Maxine improves on Jikes in terms of approachability and developer efficiency. The idea of metacircularity is not restricted to Java; examples for other languages are the Smalltalk VM *Squeak* [Ingalls et al. 1997], the *Klein VM* for Self [Ungar et al. 2005], and the Python VM *PyPy* [Rigo and Pedroni 2006].

In summary, this article contributes the following.

—This is the first comprehensive description of the architecture of the Maxine VM, a VM for Java written in Java.
—We present the design of a tightly integrated VM visualization/debugging tool, useful for both beginners and experts.
—We describe the annotations, interfaces, and other source-code techniques that make the Maxine VM approachable.
—We provide a performance evaluation showing that the Maxine VM is robust and fast enough to serve as a base for research projects.

2. SYSTEM STRUCTURE

The Maxine VM is a full-fledged Java VM [Lindholm et al. 2012] that is compatible with the JDK from Oracle and the OpenJDK [Oracle 2012e]. It acts as a replacement of the Java HotSpot VM [Oracle 2012a] shipped with these JDKs. Figure 1 presents an overview of the Maxine VM and the interactions among its main components. The Maxine VM is written almost entirely in Java, with a small part, called the *substrate*, written in C. The substrate implements the native launcher for the Maxine VM. It encapsulates in a platform-independent API the native services from the Operating System (OS), e.g., virtual memory operation, native thread support, and signal handling. The substrate also includes native services to support JNI and JVMTI (see Section 3.7).

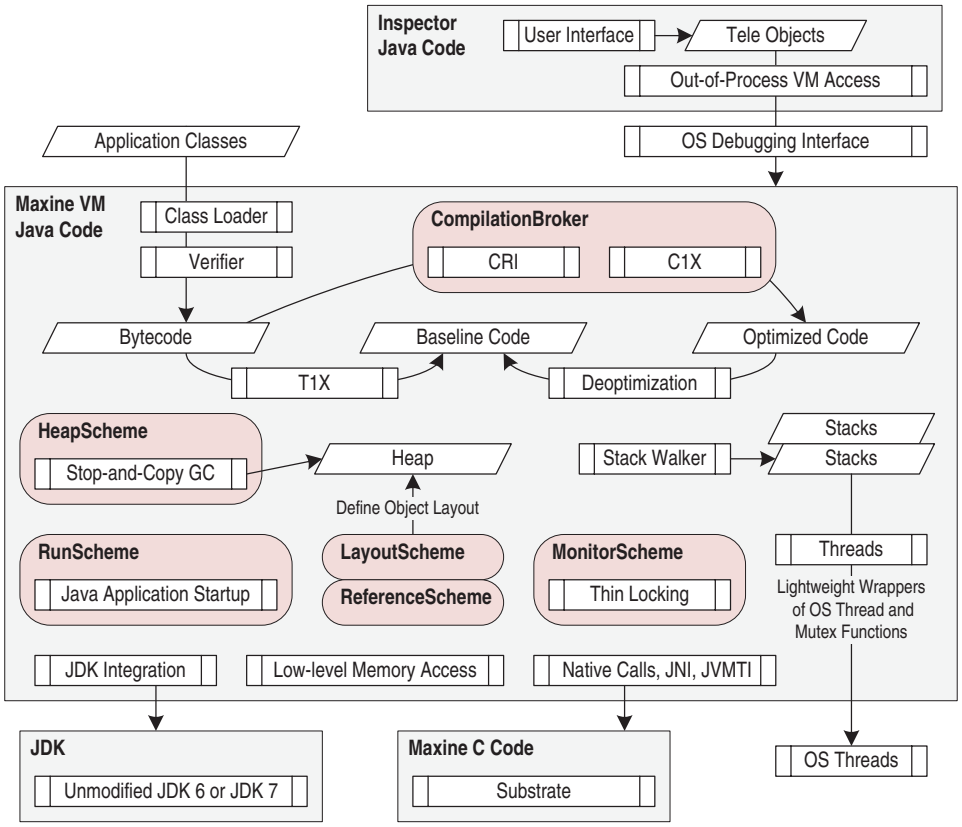


Fig. 1. Structure of the Maxine VM.

The Java part of the Maxine VM is structured around a set of components that collaborate via public interfaces. Each of these interfaces correspond to a *scheme*. Schemes formalize the functional interface between high-level abstractions of a VM implementation. The intent is to limit exposure of many low-level implementation-dependent details across these abstractions to ease the replacement of one implementation with another. The systematic use of interfaces shields schemes from their implementation details and provides this simple interface between different implementations. A VM binary contains a single concrete implementation of each scheme; this rule, enforced at VM construction, allows the optimizing compiler to completely erase otherwise expensive interface dispatch. The number of schemes in Maxine is not meant to be fixed, but rather reflects those aspects of VM design for which Maxine encourages experimentation.

—Object layout (`LayoutScheme`). This configures how objects are represented in memory, including header and fields. By default, an object is represented as a contiguous block of memory with a header stored in the first words of the block. The first header word holds a pointer to a *hub*; a second header word stores locking and Garbage Collection (GC) information. The hub carries the most frequently used metadata associated with a class, such as object size, virtual method tables, and reference maps.
—Object references (`ReferenceScheme`). This configures how objects are accessed for mutator use and how references are encoded, e.g., as direct pointers or handles. The

default is a direct reference scheme, i.e., a pointer points directly to the first word of the header of an object. By introducing a dedicated `Reference` type, this scheme relieves developers from using explicit handles in VM code.

—Heap allocation and GC (`HeapScheme`). This defines how automated memory management is implemented for objects and for dynamically compiled native code. Objects are allocated in a heap with its own GC methods, while code is allocated in a *code cache* whose implementation may support code eviction. The default heap scheme implements a straightforward stop-the-world flat semi-space collector for objects, and a form of semi-space collector for the code cache. Both application objects and internal VM objects (including class metadata objects) are allocated in the same heap and are processed by the same collector. More advanced GC algorithms and support for multiple generations are under development; this source code is available for exploration. For testing, implementations with limited or specialized functionality are helpful, e.g., an implementation that allocates but never collects, or one with extensive heap sanity checking.

—Thread synchronization (`MonitorScheme`). This represents an abstraction of monitors, including translation of synchronization bytecodes, as well as implementations of the `wait` and `notify` methods. The default monitor scheme implements a form of thin locking [Bacon et al. 1998]. It stores lock ownership information in the object header using nonblocking atomic instructions when satisfying uncontended lock request. On contention, the locking information embedded in the object header is converted into a pointer to an external heavyweight locking data structure that relies on OS mutex functions to handle contention. The lock is inflated as well when the heavyweight `wait` and `notify` object synchronization operations are used. Maxine also has an implementation of biased locking [Russell and Detlefs 2006], but it is not enabled by default. For testing, monitors can be disabled completely using the "ignore" monitor scheme.

—VM startup sequence (`RunScheme`). This is invoked by the VM after it has started basic services and is ready to set up and run a language environment. The default "Java" run scheme starts up normal JDK services and then loads and runs a user-specified Java class.

—Optimized compilation (`CompilationBroker`). This handles requests for compilation and adaptive re-compilation of methods. The Maxine VM follows a dynamic-compilation-only strategy: method bytecodes are always compiled to machine code on the first method invocation. These initial compilations are performed by *T1X*, a template-based baseline compiler that favors fast compilation over code quality (see Section 3.9). Frequently executed methods are then scheduled for recompilation using the optimizing compiler *C1X* (see Section 3.10).

The optimizing compiler is central to the Maxine VM. In addition to dynamic compilation, it compiles (ahead of time) the optimizing compiler itself, the templates used by T1X, and the methods needed to start up the VM. This centralizes in one place the mechanisms necessary for generating *safepoints*, *reference maps*, and *debugging information*, as well as other compiler-generated runtime support such as read and write barriers.

Safepoints and reference maps support *exact* GC, which enables the use of GC algorithms that relocate objects. Safepoints and debugging information support *deoptimization* [Hölzle et al. 1992], which is key for recovering from invalidated assumptions made for speculative optimizations and to support debugging while running with optimized code. Both exact GC and deoptimization require the thread stacks to be in a known state. Exact GC requires knowing the location of all references on the stack. These are recorded in reference maps by the optimizing compiler.

```
export JAVA_HOME=...    // Define the JDK 6 or JDK 7 to be used
hg clone https://kenai.com/hg/maxine~maxine maxine  // Download source code
cd maxine
mxtool/mx build         // Build the source code
mxtool/mx image         // Create the boot image
mxtool/mx vm ...        // Start the VM
mxtool/mx inspect ...   // Debug the VM and visualize it in the inspector
```

Fig. 2. Commands to get, build, and execute the Maxine VM.

Deoptimization requires information for mapping an optimized method stack frame state back to an equivalent baseline method stack frame state. This information is recorded in the debugging information by the optimizing compiler.

Similarly to objects, dynamically compiled code may be relocated by the code cache's implementation of code eviction. Code pointers are distinguished from object references and known to the optimizing compiler, which produces separate reference maps for pointers to compiled code on the stack.

Class file parsing, verification, class loading, and linking are not implemented as schemes at the moment. All variants of the Maxine VM use the same code for these. Each class loader is associated with a class registry that keeps track of the classes it has defined. Class metadata is represented as Java objects that are currently allocated in the application heap, not segregated from other standard objects.

The Maxine VM is designed to support multiple architectures and operating systems. Currently, we support Linux, Solaris, MacOS, and a Virtual Edition [Oracle 2012b] on the Xen hypervisor [Barham et al. 2003], all on the Intel x86 64-bit architecture. The source code is available as open source under the same license as the OpenJDK project [Oracle 2012c]. Documentation and support are available from a wiki [Oracle 2012d] and a public mailing list. Getting started requires only few steps (see Figure 2) with the *mx* command line tool for building and running the VM. Automatically generated project files for Eclipse and NetBeans allow developers to leverage features of modern IDEs, e.g., incremental compilation of modified source files. Building the boot image takes less than one minute on a modern laptop, allowing fast testing and debugging cycles.

3. INGREDIENTS FOR APPROACHABILITY

This section presents selected subsystems and interfaces of the Maxine VM. We focus on parts that differ from traditional VMs and, as we argue, make the Maxine VM more approachable.

3.1. Inspector

The *Maxine Inspector* is a specialized visualizer/debugger that promotes a rapid, highly productive edit-debug cycle for the Maxine VM implementation. It is closely coupled to the VM and co-evolves with it. The Inspector's multiview display (see Figure 3) reveals important aspects of the VM's runtime state at many levels of abstraction, layered upon low-level memory and register contents. Identifying problems in one of the Inspector views, correcting the VM sources (using a Java IDE), building a new boot image, and starting a new inspection session often takes just a few minutes. Browsing the VM's runtime state is also an effective way for newcomers to become familiar with the design of the Maxine VM, for example via details presented in popup "tooltips" that appear when the mouse rolls over a word of memory.

Most views shown in Figure 3 display some specialized regions of VM memory, interpreted using intimate knowledge of the VM and displayed in terms of design abstractions such as objects, references, threads, stacks, stack frames, machine code, bytecode,

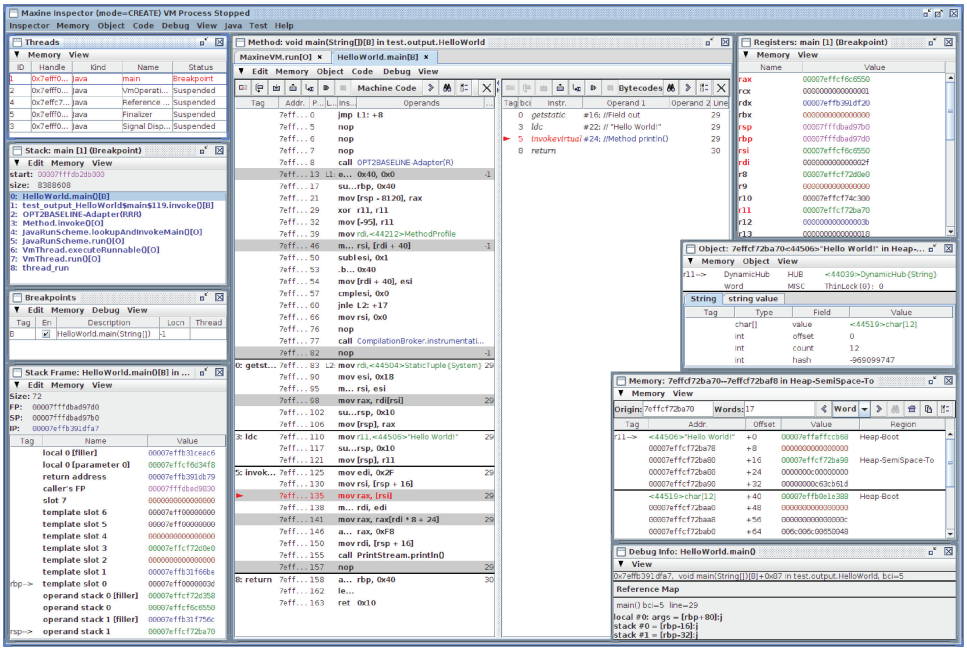


Fig. 3. Screenshot of the Maxine VM Inspector.

and thread-local state. Development bugs can disrupt these high-level interpretations, in which case low-level memory views support diagnosis. The Inspector can visualize either the state of a static boot image, a halted process during a debugging session, or a core dump.

The Inspector operates out-of-process, requires no active support from the VM, and incurs very little runtime overhead. Using the OS debugging interface, the Inspector reads memory (from which it "disassembles" runtime state), sets breakpoints (either concretely at machine code locations or abstractly on bytecodes, possibly on methods not yet loaded), and sets watchpoints (either concretely at specific memory locations or abstractly on specific object fields, tracking those fields across object relocations). Predefined breakpoints are available for interesting VM events, such as the beginning of a method compilation or the end of a GC.

The Inspector's architecture broadly comprises two parts: one supporting visual presentation and user interaction, and another (informally called *tele*) that mediates all interaction with VM state, including process management. The central function of *tele* is to read VM memory using low-level data access and to build the best possible *model* of Maxine's runtime state, expressed in terms of intelligible (to humans) design abstractions. For performance reasons, this model is incomplete (confined to those objects and other state that matter in the current session) and must be updated *incrementally*. Significant complexity in this effort derives from the *circularity* of Maxine's design. For example, the Inspector provides access to objects in terms of their *types*, which requires access to the VM's metadata, which is also represented in the VM as objects. Such circularities require a phased approach to construct and maintain *tele*'s model of runtime state, mirroring in some respects the phased construction needed for the Maxine boot image generation. In fact, the Inspector uses some of the same techniques as the boot image generator (see Section 3.2).

The Inspector's implementation, especially *tele*, is closely coupled to the VM, using many of the same language features, design techniques, and even VM code itself. This is inspired by the reflective debugging approach of the *Klein VM* [Ungar et al. 2005], a metacircular Self VM. For example:

—The Inspector loads every class loaded by the VM and uses Java reflection to extract static properties. Classes are loaded from the shared class path if possible, but the Inspector can also load directly from class information copied out of VM memory.
—At startup, the Inspector examines the configuration of the build. It then loads and sometimes reuses the specific implementation included for each VM scheme (see Section 3.3), for example simplifying the reading of object data from memory by reusing methods defined in the VM's `LayoutScheme`.
—The Inspector implements a subtype (`RemoteReference`) of the VM's `Reference` type, allowing reuse of VM code for data layout and access. It also implements its stack walker by subclassing the one used in the VM.
—The Inspector can access any remote class member (both fields and methods) using a combination of Java reflection, local class loading, reuse of the VM's object layout scheme, and reuse of platform-specific data i/o. More convenient access to members of significance (especially critical metadata and other runtime state) is supported by automatic code generation. An offline tool searches the VM code for the `@INSPECTED` annotation and generates specialized Inspector code for direct, named access to those members in VM memory.
—The Inspector uses a disassembler to display machine code encountered in VM memory. The results of code disassembly are usefully combined with the analysis of heap objects, e.g., identifying instruction arguments that point to objects.
—When the Inspector must traverse a particularly complex data structure in VM memory, for example the map from addresses to method compilations, it can use a special *remote interpreter* to execute the VM's lookup method. The code runs in the Inspector, but with data access redirected to VM memory.

To aid the understanding of VM behavior, Maxine maintains a circular buffer of events that are optionally logged by VM components. The log buffer has several implementations, chosen during boot image generation. The default is a tightly encoded, thread-local buffer. The events can be viewed in the Inspector, which recreates a single time-ordered view of the log history. The Inspector accesses the log using the `@INSPECTED` annotation and preserves events that have been overwritten in the VM circular buffer. By accessing the metadata associated with a logger class, the Inspector is able to deduce the types of the event values and display them appropriately, with all the standard mechanisms for navigating to access further information.

3.2. Boot Image Generation

The vast majority of the Maxine source code is written in Java and translated to Java bytecodes by the standard *javac* compiler. Before it can be executed by a specific processor, the bytecodes must be translated to machine code. This requires similar concepts to what Maxine needs to execute a Java application at runtime. However, compilation needs to be done ahead of time. This process is called *boot image generation*. It produces a boot image comprising two contiguous regions: a heap populated with class metadata and objects implementing the VM itself, as well as a code cache populated with code produced by the optimizing compiler. A small C program from the substrate maps the boot image into memory before calling the VM entry point via an indirect call. This effectively hands control over to Java code that implements the Maxine VM.

The boot image generator is a Java application that runs on a preexisting "host" VM, for which we use the Java HotSpot VM. There is no conceptional problem of using the

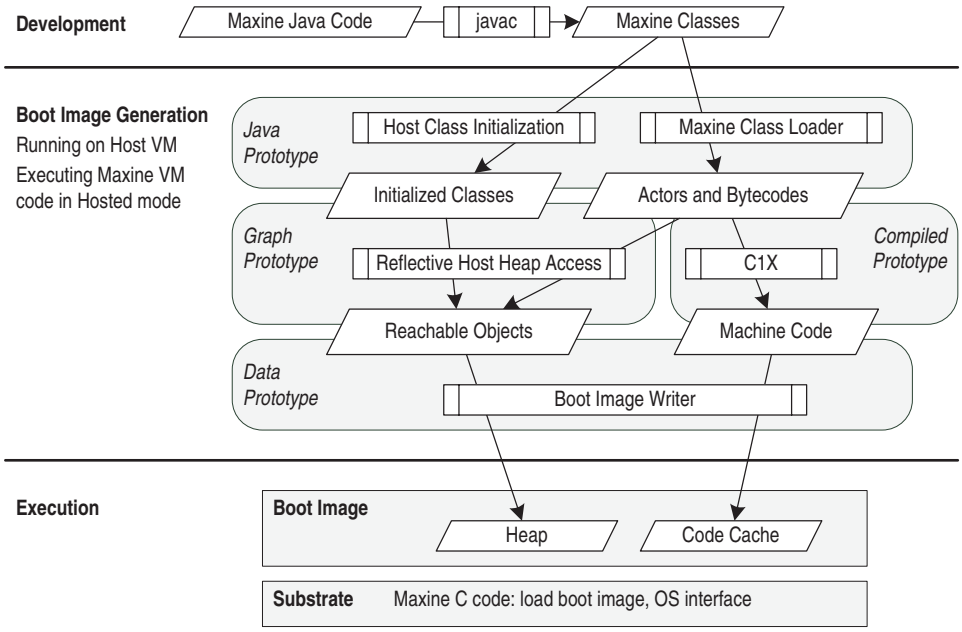


Fig. 4. Maxine VM boot image generation process.

Maxine VM itself as the host VM; however, this is currently not supported because the boot image generator includes some HotSpot-specific code. Figure 4 illustrates the boot image generation process. The boot image generator reuses large parts of the Maxine code: The Maxine class loader loads the bytecodes and generates the same internal class representation that is later also used for application classes. The C1X optimizing compiler translates bytecodes to machine code. Since these subsystems run on the host VM, they produce objects on the heap of the host VM. Using reflection, we collect these objects and write them into the boot image file.

A small part of the code needs to distinguish between boot image generation and normal VM execution mode. This can be done by calling `MaxineVM.isHosted()`, as well as by annotating a method or field with `@HOSTED_ONLY`. Hosted-only code is not part of the generated boot image, and therefore imposes no overhead when the Maxine VM is running.

The boot image generator follows a specification that identifies which implementation of each particular scheme should be included and on which platform the image is intended to run. Any class that might be written into the boot image must first be loaded by the boot image generator, including the JDK classes needed during VM startup. In the following, the term *Maxine Classes* describes the bootstrap JDK classes and the VM classes. At runtime, the VM classes are isolated in their own class loader `VMClassLoader`, the parent of which is the standard boot class loader. This makes VM classes inaccessible to application classes. The structure is mirrored at boot image generation by `HostedBootClassLoader` and `HostedVMClassLoader`.

In Figure 4, two arrows leave the box *Maxine Classes*. This illustrates that the Maxine classes are processed twice, at different levels, during boot image generation: First, the boot image generator loads the Maxine classes using the `HostedVMClassLoader`. This initializes the classes in the host VM, i.e., it runs the static initializers. Static fields can point to larger object graphs. These host VM objects are accessed using reflection,

converted to the Maxine object layout, and written into the boot image heap. The loaded classes include the Maxine class loading code and the optimizing compiler.

Second, the `HostedVMClassLoader` invokes the Maxine class loading code to build class metadata (called *actors*) as well as bytecodes (as a `byte[]` array). These objects are the input for the optimizing compiler to generate the boot image code. Additionally, the actors are written into the boot image heap. The boot image generator constructs a series of four *prototypes* that gradually define the boot image.

(1) *Java prototype.* Load all the packages that participate in the selected VM configuration. This loads and initializes classes, creates certain objects, and arranges certain settings, resulting in Java objects on the host VM heap.
(2) *Compiled prototype.* Compile to machine code all methods that are reachable from VM entry points using C1X.
(3) *Graph prototype.* Starting from the class registry, gather all transitively reachable objects.
(4) *Data prototype.* Iterate over all objects in the Graph Prototype, determine their size, and assign them to compactly packed positions in a linear address space. Create a big byte array that can hold all the objects packed together, and then transcode the objects into the byte array, translating them from host VM format to the Maxine format for the target platform. Finally, write the byte array to the boot image file.

The smallest boot image would in theory contain the minimal collection of classes and methods needed to bootstrap the VM to the point where it can dynamically load further classes from the file system and compile methods for execution. In practice this set is not easy to define precisely and depends, amongst other things, on details of the JDK implementation. In particular, it can vary from one JDK release to another. The boot image generator determines the image content by starting from so-called entry points plus a subset of the standard platform classes that are known to be necessary for the bootstrap. Entry points are methods that can be called from the external environment, e.g., are annotated with `@VM_ENTRY_POINT`; methods that are called by generated code such as stubs; or methods that are manually selected, either to satisfy bootstrap requirements (and avoid infinite recursion at runtime), or to improve startup time.

These entry points form the basis of the analysis used to select methods for the boot image. An example is the `run` method in the class `MaxineVM`, which is the entry point from the native C code. Methods (and the classes that defined them) that are transitively called from this entry point are included in the boot image. This also includes JDK classes, e.g., many of the standard collection classes. Unfortunately, the amount of JDK code pulled into the boot image using just this algorithm is too large. Therefore, certain packages are manually black-listed and excluded. For example, security and logging packages are excluded since they are not needed for bootstrapping the VM.

3.3. Scheme Abstractions

The Maxine VM is structured as major components that interact with other components via standard published interfaces called *schemes*. This facilitates composition and replacement of scheme implementations. Many performance-critical implementation aspects, such as object layout, object references, and monitors are implemented as schemes. As described earlier, a Maxine VM image only accepts one concrete implementation per scheme, a rule enforced at boot image generation.

Using class hierarchy analysis [Dean et al. 1995] and constant folding of `final` fields, the optimizing compiler can eliminate some of the interface overhead. Still, a couple of additional annotations are needed in order to eliminate the overhead completely. A method annotated with `@INLINE` is unconditionally inlined by the compiler. As the

compiler must still be able to statically bind the method invocation, the method should be declared `private` or `final`. A method carrying the `@FOLD` annotation is executed at compile time. The compiler replaces the invocation with the result, treating it as a constant. The method can have parameters that must be compile-time constants as well. This annotation is useful to define abstractions that need more flexibility than a `final` field, but where constant folding is still essential for performance. A field annotated with `@CONSTANT` is treated as a constant by the compiler. This is similar to a `final` field, but without the assignment having to happen in the initializer.

3.4. Substitution of JDK Methods

The power of Java depends also on libraries, and an approachable development and build process requires that the Maxine VM works with an unmodified standard class library. The interaction points of VM and class library are small, but crucial. Class loading, thread synchronization, reflection, and thread introspection are important areas where the class library needs support from the VM.

Currently, the JDK does not have a standardized and well-specified VM interface [Oracle 2012f]. Instead, it uses native functions that are linked to entry points specific to the Java HotSpot VM. When the VM entry points are Java methods, this design forces two unnecessary transitions across the Java/native code boundary. The Maxine VM eliminates these transitions using a mechanism for substituting methods of JDK classes at boot image generation. This is achieved using the following annotations.

—`@METHOD_SUBSTITUTIONS`. This marks a class holding substitution methods for a JDK class specified by an annotation parameter.
—`@SUBSTITUTE`. This marks a method that replaces a JDK method, usually with the same name and signature. During class loading, this method is registered in the method table of the JDK class, overwriting the original entry. For nonstatic methods, this leads to a mismatch of the declared and actual types of the `this`-pointer of the substituted method: The declared type is the substitution class, while the actual type is the JDK class. Since they are not in a subclass relationship, the Java compiler will only convert between these classes using a special `@UNSAFE_CAST`. Additionally, virtual method calls on substitution classes are not possible, since the virtual method table of the JDK class would be used. We guarantee that all method calls of substitution classes can be bound statically by enforcing substitution classes to be `final`.
—`@ALIAS`. JDK classes use private fields to maintain their internal state. Substitution methods need to access and modify this state. A field with this annotation refers to the field in the JDK class, making the field accessible from within substitution methods. The annotation can also be used to make private methods of a JDK class accessible, although this is a less common use case.

Figure 5 shows an example of how these annotations work together. When an exception is thrown at runtime, the exception object stores the stack trace, i.e., the names and bytecode positions of all Java methods on the stack. This requires walking the stack, which the VM must do as the stack frame layout is VM specific. Therefore, the method `Throwable.fillInStackTrace()` is declared `native` in the JDK. The native method stores the collected stack frames in the private field `backtrace`. The stack walking code of the Maxine VM is written in Java. We substitute the method `fillInStackTrace()` in the class `JDK_java_lang_Throwable` (the actual name of this class does not matter, but we adapted this intuitive naming convention).

The declared type of the `this`-pointer in this substitution method is `JDK_java_lang_Throwable`, but the actual type is always `Throwable`. The method `thisThrowable()` resolves this static typing issue: The `@UNSAFE_CAST` annotation specifies that it just returns the unmodified `this`-pointer, but with a different static type. It

```
@METHOD_SUBSTITUTIONS(Throwable.class)
public final class JDK_java_lang_Throwable {
  // Prevent instantiation of substitute class
  private JDK_java_lang_Throwable() { }

  // Access to this-pointer with the class Throwable
  @INTRINSIC(UNSAFE_CAST)
  private native Throwable thisThrowable();

  // Access to a private field of class Throwable
  @ALIAS(declaringClass = Throwable.class)
  private Object backtrace;

  // Replaces the native method in class Throwable
  @SUBSTITUTE
  public synchronized Throwable fillInStackTrace() {
    backtrace = getBacktrace(...);
    // Returns the receiver it was called with
    return thisThrowable();
  }
}
```

Fig. 5. Example showing the substitution of a JDK method.

is a no-op at runtime, i.e., no dynamic type checks are performed. With the help of this method, we can access and return the original `Throwable` object.

The field `backtrace` is `private` in the class `Throwable`. Using the `@ALIAS` annotation, we declare another name that refers to this field. The substitution method can now access the field using normal-looking Java code. Note that a nonaliased nonstatic field in a substitution class does not make sense because the substitution class is never instantiated. The `@ALIAS` annotation is also useful outside of substitution classes to access private fields, therefore the class that holds the aliased fields must be specified as a parameter all the time, even though it is redundant here.

All substitutions are performed during boot image generation. It is neither desirable nor possible to perform additional substitutions at runtime; code that processes substitutions and aliases is therefore *hosted only*. Substitution affects class metadata and machine code *generated* during boot image generation, but not the code *executed* during boot image generation. The boot image generator runs on the host VM, which does not need the substitutions.

3.5. Field Rewriting During Boot Image Generation

The boot image generator (see Section 3.2) traverses objects on the heap of the host VM to form the initial heap of the boot image. This means that fields of JDK classes are computed and written by the host VM, and then later read by the Maxine VM. Some fields of JDK classes are VM dependent. For example, classes of the package `java.util.concurrent` use unsafe but efficient atomic field accesses using the class `sun.misc.Unsafe`. In the Maxine VM, the methods of this class are substituted using the techniques described in Section 3.4, which computes the field offsets according to the data layout of the Maxine VM. Therefore, the unsafe methods work as expected on the Maxine VM.

However, the `java.util.concurrent` classes compute the field offsets only once and then cache them. When such a class is part of the boot image, initialization was performed by the host VM and not the Maxine VM. We need to rewrite such fields when the boot image is created using the field offsets of Maxine. Additionally, several caches need to be reset, i.e., fields need to be set back to their initial value and lists need to be emptied. This is performed by customizable rewriting rules that can be registered for

fields. There is no automatic way to detect such fields, so we identified them manually by using intuition and by debugging crashes of the Maxine VM. Currently, the list in the class `JDKInterceptor` contains 160 intercepted fields. Half of them are cached `Unsafe` offsets that are recomputed, the other half are caches that are reset.

3.6. Low-Level Memory Access

Memory access in Java is limited to named static or instance fields and elements of arrays. Further, the safety features of Java require a number of checks to be automatically enforced on access (e.g., `null` pointer checks, type checks, array bounds checks). Writing a VM requires some amount of system programming, e.g., to implement a GC or stack walking, for which raw memory access is essential. Therefore, the Maxine VM includes a raw pointer type. Frampton et al. describe a similar approach [Frampton et al. 2009], which was developed concurrently with the Maxine VM. We identified the following requirements for raw pointers.

—Raw memory access and pointer arithmetic must be possible.
—We must not extend the Java programming language, since that disturbs tool support.
—The pointer type is modeled as a class, and not as a primitive type such as `long`. This avoids accidental conversions of numbers to pointers.
—The bit-width of the architecture is transparent, i.e., code using the pointer type can work unchanged with any architecture.

The Maxine VM uses a base class `Word` to represent raw machine words. Its subclasses represent signed and unsigned numerical values (`Offset`, `Size`), and pointers (`Pointer`). Although all such values are internally represented identically and can be converted into each other, the different classes are helpful to express the content of a variable. Additional subclasses of `Word` are used, e.g., for the Java Native Interface (see Section 3.7).

`Word` extends `Object` as any other class. However, it does not make sense to call `Object` methods on it or to cast it to an `Object`. It would lead to a hard-to-debug crash at runtime, therefore we check during bytecode verification that such conversions do not occur (see Section 3.8).

The optimizing compiler has built-in knowledge about the `Word` type and handles it in special ways. For example, in the front-end of the compiler, a `Word` is treated as an `Object`, while in the back-end it is treated as a primitive value (`long` on 64-bit architectures, `int` on 32-bit architectures). An explicit compiler phase is responsible for the conversion. Methods of the class `Pointer` that perform raw memory access cannot be implemented using Java code. We use *compiler intrinsics* to support this.

Intrinsics represent operations that cannot be expressed at all in Java code, as well as operations that cannot be expressed efficiently in Java code. The optimizing compiler replaces calls to methods annotated using `@INTRINSIC` with IR nodes that perform the low-level operations. Currently, our list of intrinsics includes raw reads and writes of memory and specific processor registers (e.g., the stack pointer or instruction pointer); low-level access to the atomic compare-and-swap operations of the architecture; and unsigned comparison and division operations. Java does not support unsigned numeric values, so the efficient machine code instructions are not accessible otherwise.

During boot image generation (see Section 3.2), special compiler support for `Word` values is not yet available. Since high performance is not necessary at this point, we use boxing to represent such values. In *hosted mode*, a machine word is represented as an object that contains a single `long` field holding the actual value. The `Word` class hierarchy contains the necessary logic. This is an elegant way to reuse code, since clients of the `Word` classes do not need to distinguish between normal and hosted execution mode.

3.7. Native Interface Support

The specification of Java includes several VM interfaces that are defined as C/C++ code. This includes the *Java Native Interface* (JNI), the *Java VM Tool Interface* (JVMTI), as well as the nonstandardized interface of the Java class library with the VM. We use the following approach to transition to Java code as quickly as possible, requiring a minimal amount of C code.

—The C code of the substrate defines the function tables, but most entries remain uninitialized.
—In Java code, we implement the functionality of the methods without any boilerplate code. The methods are annotated with `@VM_ENTRY_POINT`. The class containing these methods is a syntactically correct Java class that can be edited in any IDE.
—A code preprocessor adds boilerplate code to these methods. This includes code to transition the thread state into Java mode, exception handling, and logging. The preprocessor must be invoked manually when the input file has been changed, and creates a second syntactically correct Java class. Only this second class is a part of the boot image.
—During boot image generation, the methods are compiled by the optimizing compiler. The annotation instructs the compiler to use the calling convention of the platform ABI, e.g., the parameter registers of the platform. The boot image generator also parses the C header files in order to convert function names to table indices.
—Early during VM startup, the C-defined function tables are filled using the information collected during boot image generation.

A method defined as `native` does not have a Java implementation. Instead, the VM searches loaded libraries for a method with the appropriate name and signature and links it dynamically before invoking it. The invocation also requires code to transition the thread from Java mode into native mode so that all Java frames are visible to VM components requiring stack walking, e.g., the GC. Invocation stubs are produced at runtime using bytecode generation. The bytecode is then compiled by the optimizing compiler. Generating bytecode is portable, much easier, and less error prone than handcrafting machine code for stubs.

Native methods do not operate directly on object pointers, but use special opaque references. The Maxine VM implements these as instances of `JniHandle`, a subclass of `Word`. JNI handles implement a level of indirection that enables a GC to move objects while native code holds opaque references to them. Local JNI handles (parameters to native functions) are stack-allocated in the stub that invokes the native method. Global JNI handles (which are requested explicitly by native code) are implemented as indexes into an object array. This avoids explicit memory management for the native interface, and avoids special support from the GC.

3.8. Bytecode Verification

Maxine includes bytecode verifiers that implement the Java VM specification. While implementing bytecode verification could have been deferred, it actually proved useful to have it available earlier rather than later in Maxine for several reasons.

—Debugging internal bytecode generation: Maxine makes use of bytecode generation for implementing method invocation via reflection and JNI stubs. Bytecode generation errors are easier to detect and fix when caught in the verifier.
—Sanitizing use of `Word` types: At the language level, the `Word` type hierarchy (see Section 3.6) is part of the `Object` hierarchy, so `Word` values are accepted where `Object` values are expected. However, this is not safe in Maxine. To make programming with `Word` types easier, the Maxine verifiers are extended to be aware of the `Word` hierarchy.

```
@T1X_TEMPLATE(IALOAD)
static int iaload(@Slot(1) Object a, @Slot(0) int i) {
  ArrayAccess.checkIndex(a, i);
  return ArrayAccess.getInt(a, i);
}

@INLINE
static void checkIndex(Object a, int i) {
  int l = Layout.readArrayLen(Reference.fromJava(a));
  if (UnsignedMath.aboveOrEqual(i, l)) {
    throw Throw.arrayIndexOutOfBoundsException(a, i);
  }
}

@INLINE
static int getInt(Object a, int i) {
  return Layout.getInt(Reference.fromJava(a), i);
}
```

Fig. 6. Source code of the integer array load (`iaload`) template.

Any VM source code that incorrectly mixes `Object` and `Word` values results in a verification error.

3.9. T1X Baseline Compiler

The T1X baseline compiler is Maxine's first line of execution (Maxine has no interpreter). As such, its primary goal is to produce machine code as fast as possible. Code quality is of secondary concern. Therefore, a template-based design is employed where code generation is little more than copying sequences of prebuilt machine code, one for each bytecode, into a code buffer.

In addition to the primary goal of being fast, T1X is also designed with approachability in mind. It is easy to modify and experiment with the translation of bytecodes. We want to minimize other concerns when writing or modifying a template compiler. In particular, we want to minimize the need for hand-crafted machine code, abstract the details of how the JVM operand stack and local variables are represented, and remove concerns about template code / GC interactions.

All of these goals are addressed by writing the templates in Java with annotations and using the optimizing compiler to compile them. This is done ahead of time during boot image generation. The mapping from logical JVM operand stack slots to template parameters is specified with a `@Slot` annotation. Reference maps of object values that are live across a GC point in the template are created by the optimizing compiler.

Figure 6 shows the source code for the `int` array load `IALOAD` bytecode. It consists of a method annotated with `@T1X_TEMPLATE` that calls two other helper methods, also shown. The `@INLINE` annotation forces the optimizing compiler to inline the helper methods. The other calls in the source code are intrinsified by the compiler, and the `throw` statement translates to a runtime call. The machine code generated for this template is shown in Figure 7. The template is preceded by a prologue emitted by T1X to move values from the stack into registers for the `a` and `i` parameters of the `iaload` method. The epilogue pushes the result onto the stack. Prologue and epilogue emission is automated by the signature of the `iaload` method and use of the `@Slot` annotation.

This automation and the fact that templates are written in Java simplifies modification of and experimentation in T1X. Furthermore, the template writers need not be concerned about object values. Figure 8 shows a modified version of Figure 6 where tracing calls have been inserted before and after the array load. Both of these tracing

```
// Prologue: loading of operand stack slots
     mov     rdi, [rsp + 16]
     mov     esi, [rsp]
// Compiled template
     mov     rax, rdi
     movsxd  rax, [rax + 16]
     cmp     esi, eax
     jb      L1
     call    Throw.indexOutOfBoundsException()
     nop
     mov     rdi, rax
     call    MaxRuntimeCalls.runtimeUnwindException
     nop
 L1: movsxd  rsi, esi
     movsxd  rax, rdi[rsi * 4 + 24]
// Epilogue: storing of operand stack slots
     addq    rsp, 0x10
     mov     [rsp], eax
```

Fig. 7. Machine code of the `iaload` template.

```
@T1X_TEMPLATE(IALOAD)
static int iaload(@Slot(1) Object a, @Slot(0) int i) {
  traceBefore(Bytecodes.IALOAD);
  ArrayAccess.checkIndex(a, i);
  int result = ArrayAccess.getInt(a, i);
  traceAfter(Bytecodes.IALOAD);
  return result;
}
```

Fig. 8. Extended `iaload` template with tracing.

calls are possible GC points. During the first one, the `a` argument is live. The optimizing compiler automatically takes care of this as it generates the correct reference map entry for `a` at the call site.

Even though T1X templates are straightforward, there is still a lot of similarity between certain templates given that the bytecode instruction set is (partially) typed with respect to types. For example, along with `IADD`, there are also `LADD`, `FADD`, and `DADD`. To factor out the commonality, we use source-code generation. The generator can itself be extended. For example, this extensibility is used to generate specialized templates for the purpose of instrumentation and the debugging functionality of JVMTI.

3.10. Compiler Interface and Optimizing Compiler

Maxine includes a well-defined compiler-runtime-interface (CRI) that separates the optimizing compiler from the rest of the VM. The CRI contains Java interfaces and classes modeling the runtime data structures needed by the compiler, as well as for communicating the result of a compilation back to the runtime. The CRI uses an assembler-like API (*XIR*) to specify how certain complex bytecode operations are lowered to machine code. XIR templates are defined by the VM, not the compiler, to make the compiler VM independent. This approach is described extensively in Titzer et al. [2010].

The CRI code is independent from the VM and the compiler. This is enforced on the source-code level: the project defining the CRI has no dependencies to any other project in the Maxine code base, so using VM classes by accident is impossible. Similarly, the source-code project for the optimizing compiler only depends on the CRI. As a consequence, the low-level systems programming features of the Maxine VM, such as the `Word` types (see Section 3.6), cannot be used accidentally in the compiler.

The optimizing compiler of the Maxine VM, *C1X*, uses the same architecture and intermediate representations as the client compiler (also called *C1*) of the Java HotSpot VM [Kotzmann et al. 2008]. C1X is a more or less literal Java port of the C++ code of C1. The main difference is the introduction of XIR into C1X, which enables the separation of the compiler from the VM. C1X and C1 use the same set of compiler optimizations, e.g., constant folding, global value numbering, and linear scan register allocation. The visualization tool [Java.net 2012] written for C1 also works also for C1X, enabling introspection of compiler data structures and intermediate representations, from bytecode parsing to machine code generation.

Another evidence of the CRI's VM independence is an implementation of the CRI classes for the Java HotSpot VM, which allows using the unmodified optimizing compiler also in the Java HotSpot VM. This latter effort was the starting point for the Graal project (see Section 5), an improved optimizing compiler.

4. PERFORMANCE EVALUATION

While approachability is the first goal for the Maxine VM, performance is important as well. Research based on Maxine will only be considered valid when the performance is within a reasonable margin of highly optimized production-quality VMs. We compare the Maxine VM to the Java HotSpot VM in both the client and server configurations. In detail, we compare the following configurations:

—the 64-bit Java HotSpot server VM of the JDK 7 update 6;
—the 64-bit Java HotSpot client VM. Oracle does not ship this version of the VM, therefore we compiled it ourselves from the OpenJDK source code. We ensured that it is the exact same revision of the source code used by Oracle for the server VM;
—the Maxine VM, using the tip version of Maxine from September 8, 2012. It uses the same JDK 7 class library as the other configurations, and is also a 64-bit VM.

All benchmarks were executed on a two socket, dual core AMD Opteron 2214 with 2.2 GHz, a total number of 4 cores, and 4GByte main memory. The OS is Oracle Enterprise Linux, version 2.6.18. All reported numbers are the average of 10 executions.

We do not use special configuration or tuning options for both HotSpot VM configurations, other than specifying the heap size. The GC of the Java HotSpot VM uses generational collection, with a young generation composed of a large eden and two comparatively small semi-space survivor spaces. These occupy an insignificant percentage of total heap size; the biggest part of the heap is used for the old generation that is collected by a mark-and-compact algorithm. With a heap size of 4GByte or less, the significant bits of all object pointers fit into 32 bits. In such a configuration, the HotSpot VM compresses pointers to 32-bit values. The Maxine VM does not have this optimization implemented, so it uses full 64-bit object pointers. In addition, Maxine's current semi-space GC can use only half of the memory at a time. For the benchmark results shown in Figure 9, we fix the heap size to 2GByte for all three VM configurations. The HotSpot VM can fit more than twice as many objects into this heap size, requiring fewer garbage collections.

Figure 9(a) shows performance results for the DaCapo-9.12-bach benchmarks [Blackburn et al. 2006], a collection of medium to large sized Java applications. We run each application 4 times in the same VM and report the peak performance, i.e., the last run where all frequently executed methods have been compiled. On average, Maxine reaches 67% of the HotSpot client VM performance, and 57% of the HotSpot server VM performance. No benchmark shows exceptionally bad performance. We consider this a reasonable margin for research use.

The startup performance of the Maxine VM is reasonable as well. For the first run of the DaCapo benchmarks, which includes class loading and compilation, Maxine

reaches 62% of the HotSpot client VM performance, and 82% of the HotSpot server VM performance. Note that the server VM is slower than the client VM during startup since its compilation speed is an order of magnitude slower.

Figure 9(b) shows performance results for the SPECjvm2008 benchmarks [SPEC 2008], a collection of small to medium sized Java applications. The results are similar to the DaCapo benchmarks: On average, the Maxine VM reaches 72% of the HotSpot client VM performance, and 46% of the HotSpot server VM performance. In the Java HotSpot VM, all subsystems are highly optimized. This is not possible in the context of a research project. For example, the benchmark where the Maxine VM performs worst, *serial*, performs solely object serialization and deserialization, which we have not yet optimized. Since Maxine uses an unmodified JDK, we have not encountered benchmarks or applications that cannot be executed because of class library incompatibilities.

Figure 10 shows benchmark results across a range of heap sizes for the Maxine VM and the Java HotSpot client VM. We start with a heap size of 64MByte and double it until we reach 2GByte. Not all benchmarks run with small heap sizes; a missing bar means that the benchmark throws an `OutOfMemoryError` with this heap size. The Maxine VM requires a larger minimum heap size than the Java HotSpot VM. This is expected, since the Java HotSpot VM manages class metadata in the native memory heap and the *permanent generation* (a special garbage-collected area for metadata that is not included in the specified heap size). Studies show that the required amount of such non-Java memory is high [Ogata et al. 2010].

Some of the benchmarks, for example *avrora*, run well with small heap sizes. Neither the Maxine VM nor the Java HotSpot VM show much difference between a heap size of 64MByte and 2GByte. However, most of the benchmarks run faster when increasing the heap size, both for the Maxine VM and the Java HotSpot VM.

5. FUTURE WORK

The short-term plans for the Maxine VM focus on improving performance. We are working on a generational GC, which will reduce long GC pause times that currently occur with large heap sizes. Simultaneously, we are working on an improved optimizing compiler, which will work both in the Java HotSpot VM and the Maxine VM. It is developed in a separate OpenJDK project called Graal [Oracle 2012g]. The Graal Compiler-Runtime-Interface (CRI) is an improved version of the Maxine CRI, so the integration of Graal into Maxine will be straightforward. Finally, we will keep Maxine up to date with respect to improvements of the Java VM specification. We plan to implement method handles and the `invokedynamic` bytecode that were introduced for Java 7. Currently Maxine can work with a JDK 7 class library, but cannot execute applications needing VM features introduced for Java 7.

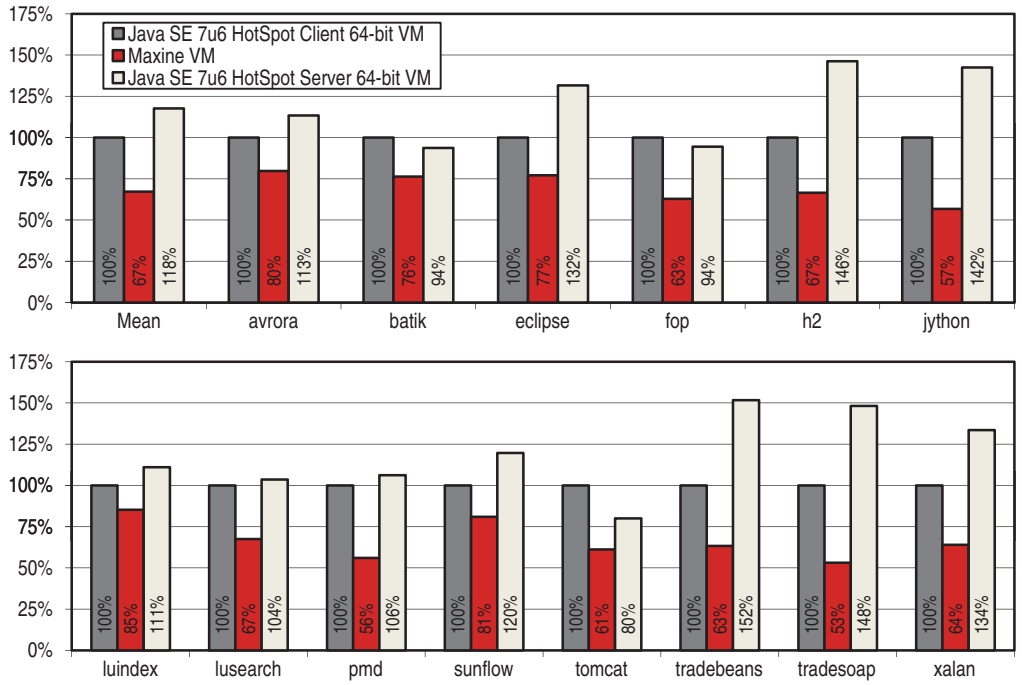
On the long term, we envision Maxine as a research platform for multiple languages. Exploiting the already modular structure and scheme abstractions, we want to make Maxine a truly modular VM. Benefits and a possible structure of a modular VM based on the Maxine VM are described in Wimmer et al. [2012].

6. RELATED WORK

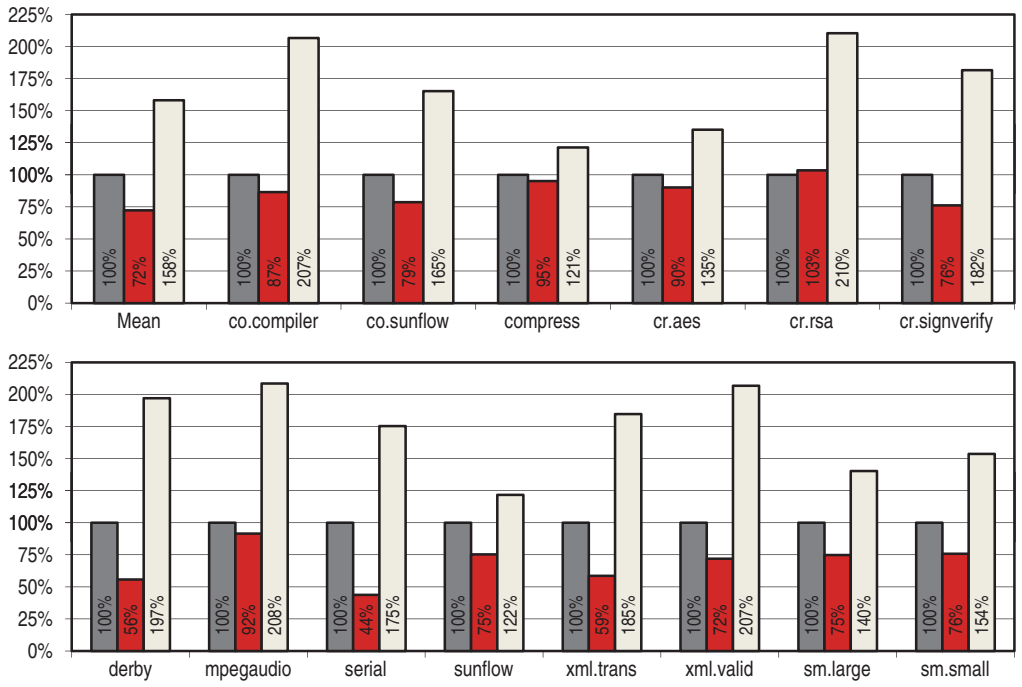
The discussion in this section first addresses metacircular VM implementations in general, followed by metacircular VMs for Java. The last part considers debugging support for VMs.

6.1. Metacircular VMs

Metacircularity originates from LISP [McCarthy 1978]. Its `eval()` function requires a LISP interpreter, which was defined in LISP itself. Also, the first successful LISP compiler was already developed in LISP.

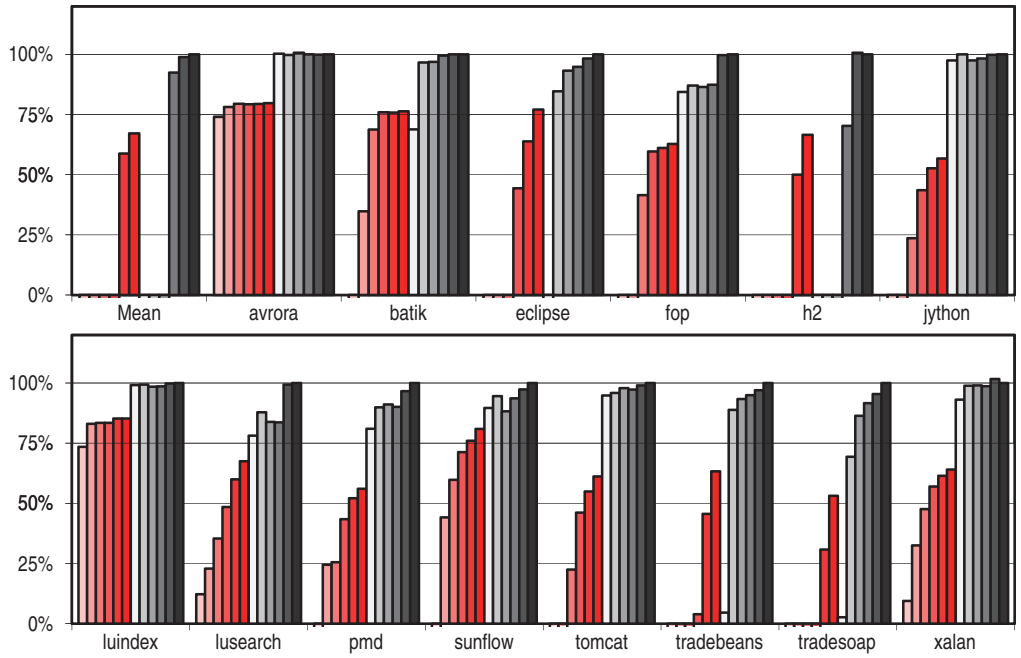


(a) DaCapo-9.12-bach benchmarks (higher is better)

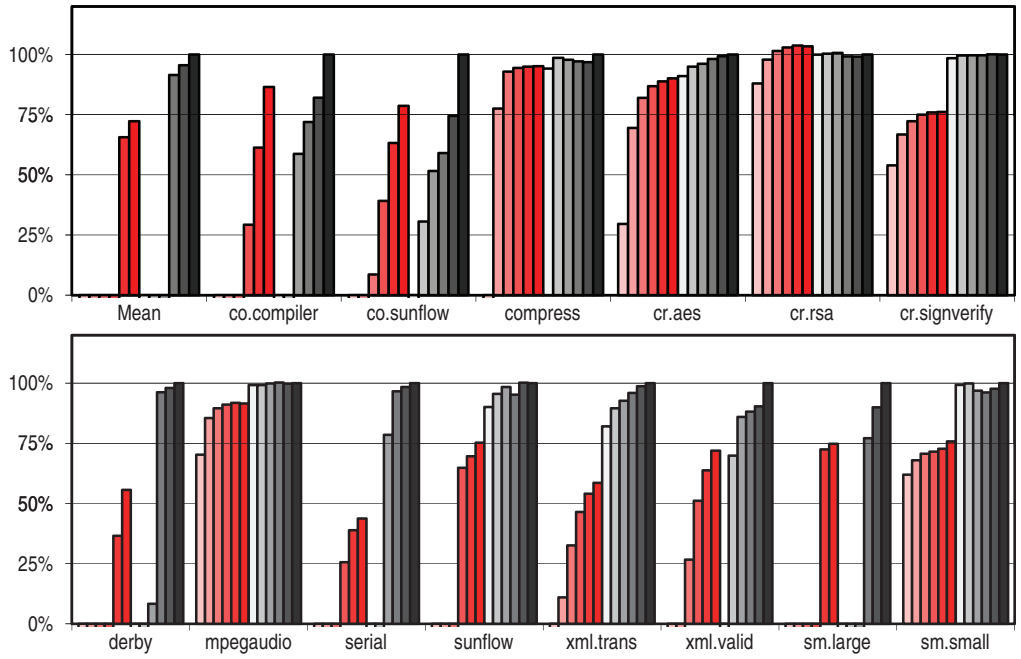


(b) SPECjvm2008 benchmarks (higher is better)

Fig. 9. Execution speed of benchmarks with a heap size of 2GByte.



(a) DaCapo-9.12-bach benchmarks (higher is better)



(b) SPECjvm2008 benchmarks (higher is better)

Fig. 10. Execution speed of benchmarks with a heap size of 64MByte, 128MByte, 256Mbyte, 512MByte, 1GByte, and 2GByte. The six left red bars show the results for the Maxine VM, the six right gray bars show the result for the Java SE 7u6 HotSpot Client 64-bit VM. The baseline is the same as in Figure 9.

The idea was then applied to other languages. For example, the programming environments and compilers for Oberon [Wirth and Gutknecht 1992] and Cecil [Chambers 1998] were written in the respective language. This leverages the benefits of the language for the language development itself, and also simplifies reflective access in the language. Modules could be loaded and unloaded on demand, and the use of a bootlinker was explored to manage VM images. However, these languages were statically compiled to machine code and not executed by a VM, so there were no metacircular runtime environments.

In Smalltalk [Deutsch and Schiffman 1984], large parts of the system were written in Smalltalk itself. Powerful reflective facilities allowed the access of class, method, and field metadata objects from within an application. Also, the modular programming style of Smalltalk could be applied to these reflective system parts. However, the core bytecode interpreter of most Smalltalk systems was still written in a statically compiled language. The "blue book" reference implementation was written in Smalltalk itself [Goldberg and Robson 1983], but it was intended only for illustrative purposes.

Squeak [Ingalls et al. 1997] is a metacircular Smalltalk VM. It is written in a subset of Smalltalk: A nonobject-oriented programming style is necessary to allow the interpreter to be translated to C code, which is then compiled to machine code. The reduced language limits the metacircular benefit because VM extensions have to be coded in a special way before they can be integrated with the VM.

PyPy [Rigo and Pedroni 2006] is a VM for Python, based on a framework useable for any dynamic language. It is written in RPython, a reduced version of Python where types can be inferred statically [Ancona et al. 2007]. Runtime optimizations are performed by a trace-based just-in-time (JIT) compiler. During bootstrapping, the VM code is translated to C code. This requires an additional C compilation step during bootstrapping.

Self [Chambers et al. 1989] is a language that offers even more dynamic reflective facilities. Every method dispatch is dynamic and can be changed at runtime. The original VM was written in C++ and was the incubator for many dynamic and feedback-directed optimizations available in today's VMs. The *Klein VM* [Ungar et al. 2005] is a metacircular Self VM written entirely in Self. According to its authors, the primary goal for Klein is to achieve feature parity with the existing Self VM, while reducing the amount of source code by two thirds. Klein achieves a high degree of reuse by trading off performance for architectural simplicity and ease of development. It uses Self's mirror-based reflection system for debugging. The normal Self debugger inspects objects of the running Self VM (the application and IDE run in the same VM). The Klein debugger extends this system by providing a new kind of mirror that inspects remote objects in the Klein VM, using the debug interface provided by Klein. The debugger reuses VM code to interpret the object layout. Our inspector's *tele* layer is comparable and inspired by this approach.

6.2. Metacircular Java VMs

The *Jikes Research VM* [Alpern et al. 1999, 2000, Rogers and Grove 2009] is a long-standing research platform well suited for experiments with memory management [Blackburn et al. 2004] and JIT compilation [Burke et al. 1999; Arnold et al. 2000]. A number of characteristics set Jikes and Maxine apart. First of all, Jikes currently binds to third-party Java class libraries such as GNU classpath or Apache Harmony, which have been discontinued. The Jikes source-code repository contains an experimental branch that integrates with the OpenJDK class library, but at the time of this writing this work is unfinished and can execute only some of the DaCapo benchmarks. Maxine integrates with standard JDK releases.

Jikes requires a preprocessor to generate source code depending on various configurable properties, implying that developers cannot work on a complete version of the "primordial" source code in an IDE. Instead, several configuration choices are already wired into the visible code. With Maxine, the entire code base is always editable and navigable in the IDE, with no code generation steps disrupting approachability in this sense.

Finally, debugging the Jikes VM requires using low-level tools such as `gdb`, which do not know anything about the abstractions at work inside the VM, not even about Java objects and their memory layout. The Maxine Inspector provides an integrated high-level view with abstractions for all kinds of entities that exist in a running VM and application.

OVM [Palacz et al. 2005] is a VM implementation framework that has largely been applied in research on real-time Java. From an instruction set specification described in Java, various VM components can be generated, e.g., an interpreter or a JIT compiler. The framework is partially written in Java, and partially in C; it employs code transformation tools from Java to C and C++. Java features such as JNI are not supported. In a nutshell, OVM explicitly targets real-time systems and does not make an attempt at gaining wider recognition as an *approachable* general-purpose VM framework. Maxine is compatible with standard VM releases and supports running a wide range of Java applications.

The *Moxie VM* [Blackburn et al. 2008] set out to provide a "next-generation" JVM architecture, with a strong emphasis on modularity and components as well as performance. It reused some ideas and components from Jikes, such as the MMTk memory management framework [Blackburn et al. 2004]. Unfortunately, the Moxie source code is not available for further inspection, but the available technical report suggests that the project achieved significant improvements on several accounts.

There are several resemblances between Moxie and Maxine. For example, to introduce variability, Moxie employed the *abstract factory* pattern, the usages of which were optimized by the JIT compiler so that the indirection did not lead to performance penalties. Instead of relying on the optimization capabilities of a JIT compiler, Maxine adopts dedicated annotations that concretely drive code binding decisions, e.g., `@FOLD`.

As another example, the bootstrap processes of both Moxie and Maxine make use of different *modes* the VM code can run in: Moxie's and Maxine's *hosted* and *target* modes correspond, respectively. Moxie additionally supports *hosted target* mode, which is used to run complex tests that require a running VM environment but not full stand-alone VM capabilities. To achieve the same, Maxine provides a dedicated `AbstractTestRunScheme`.

Jikes, OVM, and Moxie all use a similar framework for low-level memory access, the *org.vmmagic* package. Frampton et al. formalize this framework, define requirements, and present design alternatives [Frampton et al. 2009]. The framework includes unboxed types for representing raw pointers and compound data, as well as compiler intrinsics that operate on this data. Our framework (see Section 3.6), which was defined concurrently with this work, is similar.

6.3. Debugging Support for Java VMs

A variety of projects aim at making JVM execution more transparent by providing detailed introspective means. Two projects share important goals with, and in some ways anticipate, the Maxine Inspector. *Jikes* contained the `jdp` tool [Ngo and Barton 2000; Jikes RVM 2002] prior to its 2.2.0 release. The *HotSpot Serviceability Agent* (HSSA) [Russell and Bak 2001] accompanies the standard JVM. All three are out-of-process introspective debuggers implemented in Java that provide design-level access to important runtime state such as heap objects and stacks, without requiring runtime

support in the VM. All are closely aligned with their respective target VM implementations, which comes at the cost of a requirement for co-evolution with the target VM.

Like the HSSA and `jdp`, the Inspector can attach to running target VM processes, although platform support for this has so far been put in place only for the Virtual Edition. The Inspector and `jdp` are implemented directly atop platform OS primitives, unlike the HSSA's dependence on *dbx*. The Inspector supports a large and expanding set of *interactive views* that are closely integrated around both navigation and visualization. Conversely, the HSSA provides only API access upon which tools can be implemented, and `jdp` features a command line interface. Finally, Jikes and Maxine are implemented in Java, unlike the HSSA's target VM. This affords much more opportunity for code reuse, greatly expedites maintaining alignment between the VM and tool code bases, and enables modeling VM state at much finer granularity than is possible in the HSSA.

Another tool that builds upon the HSSA [Wright et al. 2006] exceeds the latter's capabilities by enabling users to reason about application behavior taking into account the entire range from processor events like cache misses to Java source code. Unlike the Maxine Inspector, the tool leaves no footprint in the VM code, but it requires the observed VM to be run in a hardware simulation environment. The tool is limited in that it can only observe the top stack frame of any thread. Moreover, it does have a detailed and complete model of HotSpot internals, but not of the C standard library used by HotSpot, a nonissue for the Inspector since Maxine uses standard JDK classes.

7. CONCLUSIONS

Maxine is a VM for Java written entirely in Java. The modular structure, high-level programming in Java, and the co-designed Inspector make it more approachable than traditional VMs written in C/C++ and simplify development and debugging. The pervasive use of modern Java features, e.g., Java annotations, is a key ingredient. The scheme abstractions, the low-level memory access, or the T1X template compiler would not be possible without the use of annotations that add systems programing features not present in the Java language. The Maxine VM shows that the abstractions do not have an inherent performance penalty. Even without elaborate compiler optimizations and GC algorithms, the performance is between 103% and 44% of the production-quality Java HotSpot client VM. We believe that the Maxine VM, and metacircular VMs in general, can reach the performance of or even outperform the Java HotSpot VM. There are no inherent limitations that prevent some optimizations or add overhead. With these ingredients, we envision the Maxine VM to be a platform for productive VM research.

ACKNOWLEDGMENTS

We thank Mario Wolczko, Bernd Mathiske, Thomas Würthinger, and Ben Titzer for their work on the Maxine VM. We also thank the numerous interns for their ideas and contributions, as well as for shaping the approachability of the VM.

REFERENCES

Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., and Whaley, J. 2000. The Jalapeño virtual machine. *IBM Syst. J. 39,* 1, 211–238.

ALPERN, B., ATTANASIO, C. R., COCCHI, A., HUMMEL, S. F., LIEBER, D., MERGEN, M., SHEPHERD, J. C., AND SMITH, S. 1999. Implementing Jalapeño in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 314–324.

ANCONA, D., ANCONA, M., CUNI, A., AND MATSAKIS, N. D. 2007. RPython: A step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the Dynamic Languages Symposium*. ACM Press, 53–64.

ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. 2000. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 47–65.

BACON, D. F., KONURU, R., MURTHY, C., AND SERRANO, M. 1998. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 258–268.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. 2003. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM Press, 164–177.

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. 2004. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the International Conference on Software Engineering*. IEEE.

BLACKBURN, S. M., GARNER, R., HOFFMAN, C., KHAN, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 169–190.

BLACKBURN, S. M., SALISHEV, S. I., DANILOV, M., MOKHOVIKOV, O. A., NASHATYREV, A. A., NOVODVORSKY, P. A., BOGDANOV, V. I., LI, X. F., AND USHAKOV, D. 2008. The Moxie JVM experience. Tech. rep. TR-CS-08-01, Department of Computer Science, The Australian National University.

BURKE, M. G., CHOI, J.-D., FINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M. J., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 1999. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM Conference on Java Grande*. ACM Press, 129–141.

CHAMBERS, C. 1998. The Cecil language specification and rationale, version 3.0. Tech. rep., Department of Computer Science and Engineering, University of Washington.

CHAMBERS, C., UNGAR, D., AND LEE, E. 1989. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 49–70.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 952, Springer, 77–101.

DEUTSCH, L. P. AND SCHIFFMAN, A. M. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, 297–302.

FRAMPTON, D., BLACKBURN, S. M., CHENG, P., GARNER, R. J., GROVE, D., MOSS, J. E. B., AND SALISHEV, S. I. 2009. Demystifying magic: High-Level low-level programming. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*. ACM Press, 81–90.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 32–43.

INGALLS, D., KAEHLER, T., MALONEY, J., WALLACE, S., AND KAY, A. 1997. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 318–326.

JAVA.NET. 2012. Java HotSpot client compiler visualizer. http://java.net/projects/c1visualizer/

JIKES RVM. 2002. The Jikes™ research virtual machine user's guide v2.1.1. http://sourceforge.net/projects/jikesrvm/files/jikesrvm/2.1.1/jikesrvm-2.1.1.tar.gz, file: userguide.ps

KOTZMANN, T., WIMMER, C., MÖSSENBÖCK, H., RODRIGUEZ, T., RUSSELL, K., AND COX, D. 2008. Design of the Java HotSpot™ client compiler for Java 6. *ACM Trans. Archit. Code Optim. 5,* 1, Article 7.

LINDHOLM, T., YELLIN, F., BRACHA, G., AND BUCKLEY, A. 2012. The Java virtual machine specification, Java SE 7 edition. http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf

MCCARTHY, J. 1978. History of LISP. In *Proceedings of History of Programming Languages*. ACM Press, 173–185.

NGO, T. AND BARTON, J. 2000. Debugging by remote reflection. In *Proceedings of Euro-Par 2000 - Parallel Processing*. Lecture Notes in Computer Science. Springer, 1031–1038.

OGATA, K., MIKURUBE, D., KAWACHIYA, K., TRENT, S., AND ONODERA, T. 2010. A study of Java's non-Java memory. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 191–204.

ORACLE. 2012a. The Java HotSpot performance engine architecture. http://www.oracle.com/technetwork/java/whitepaper-135217.html

ORACLE. 2012b. Maxine virtual edition. http://labs.oracle.com/projects/guestvm/

ORACLE. 2012c. Maxine VM source code. http://kenai.com/projects/maxine/

ORACLE. 2012d. Maxine VM wiki. https://wikis.oracle.com/display/MaxineVM/

ORACLE. 2012e. OpenJDK. http://openjdk.java.net/

ORACLE. 2012f. OpenJDK: Common VM interface. http://openjdk.java.net/projects/cvmi/

ORACLE. 2012g. OpenJDK: Graal project. http://openjdk.java.net/projects/graal/

PALACZ, K., BAKER, J., FLACK, C., GROTHOFF, C., YAMAUCHI, H., AND VITEK, J. 2005. Engineering a common intermediate representation for the OVM framework. *Sci. Comput. Program. 57,* 3, 357–378.

RIGO, A. AND PEDRONI, S. 2006. PyPy's approach to virtual machine construction. In *Companion to the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*. ACM Press, 944–953.

ROGERS, I. AND GROVE, D. 2009. The strength of metacircular virtual machines: Jikes RVM. In *Beautiful Architecture*, D. Spinellis and G. Gousios, Eds. O'Reilly, Chapter 10.

RUSSELL, K. AND BAK, L. 2001. The HotSpot serviceability agent: An out-of-process high level debugger for a Java virtual machine. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology*. USENIX Association, 16–16.

RUSSELL, K. AND DETLEFS, D. 2006. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 263–272.

SPEC 2008. SPECjvm2008. http://www.spec.org/jvm2008/

TITZER, B. L., WÜRTHINGER, T., SIMON, D., AND CINTRA, M. 2010. Improving compiler-runtime separation with XIR. In *Proceedings of the ACM / USENIX International Conference on Virtual Execution Environments*. ACM Press, 39–50.

UNGAR, D., SPITZ, A., AND AUSCH, A. 2005. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*. ACM Press, 11–20.

WIMMER, C., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. 2012. Fine-Grained modularity and reuse of virtual machine components. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, 203–214.

WIRTH, N. AND GUTKNECHT, J. 1992. *Project Oberon*. Addison-Wesley.

WRIGHT, G., MCGACHEY, P., GUNADI, E., AND WOLCZKO, M. 2006. Introspection of a Java™ virtual machine under simulation. Tech. rep. SMLI TR-2006-159, Sun Microsystems Labs.

Received May 2012; revised September 2012; accepted September 2012