Fine-Grained Modularity and Reuse of Virtual Machine Components

Christian Wimmer* Stefan Brunthaler† Per Larsen† Michael Franz†

*Oracle Labs †Department of Computer Science, University of California, Irvine

christian.wimmer@oracle.com s.brunthaler@uci.edu perl@uci.edu franz@uci.edu

Abstract

Modularity is a key concept for large and complex applications and an important enabler for collaborative research. In comparison, virtual machines (VMs) are still mostly monolithic pieces of software. Our goal is to significantly reduce to the cost of extending VMs to efficiently host and execute multiple, dynamic languages. We are designing and implementing a VM following the "everything is extensible" paradigm. Among the novel use cases that will be enabled by our research are: VM extensions by third parties, support for multiple languages inside one VM, and a universal VM for mobile devices.

Our research will be based on the existing state of the art. We will reuse an existing metacircular Java™ VM and an existing dynamic language VM implemented in Java. We will split the VMs into fine-grained modules, define explicit interfaces and extension points for the modules, and finally re-connect them.

Performance is one of the most important concerns for VMs. Modularity improves flexibility but can introduce an unacceptable performance overhead at the module boundaries, e.g., for inter-module method calls. We will identify this overhead and address it with novel feedback-directed compiler optimizations. These optimizations will also improve the performance of modular applications running on top of our VM.

The expected results of our research will be not only new insights and a new design approach for VMs, but also a complete reference implementation of a modular VM where everything is extensible by third parties and that supports multiple languages.

AOSD'12, March 25–30, 2012, Potsdam, Germany.

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

General Terms Algorithms, Languages, Performance

Keywords Maxine, languages, dynamic languages, Java, virtual machine, modularity, modular VM, metacircular VM, just-in-time compilation, optimization, performance

1. Introduction

Modularity enables developers to build increasingly large and complex applications. It is often not feasible to closely control the entire architecture of such applications. They are often developed by distributed teams or even multiple vendors, so it is necessary that individual modules are independent and have a well-defined interface to the rest of the application. The concept of a *module system*, i.e., a software layer that loads, unloads, and connects modules, is well understood in the field of software engineering [29]. It is used on a daily basis with industry standards like OSGi [24].

The complexity of virtual machines (VMs) is steadily increasing. This includes the development of more advanced just-in-time (JIT) compilers and garbage collectors, but also optimized run-time services such as thread synchronization. Additionally, the rise of dynamic languages has led to a plethora of new VMs, since a completely new VM is typically developed for every new language. VMs are still mostly monolithic pieces of software. They are developed in C or C++, the languages that they aim to replace. In summary, VMs offer a lot of benefits for applications running *on top* of them, but they mostly do not utilize these benefits for themselves.

The reason often cited for this is performance. Many subsystems are highly performance critical or have performance critical connection points with other subsystems. For example, when a garbage collector needs write barriers, the JIT compiler must efficiently embed them into the generated code. A callback into the garbage collector for every field access would be too slow. Using modularity and interfaces, it would be an even slower call that needs dynamic dis-

patch. Existing modular VMs therefore have carefully selected module boundaries to avoid the performance impact.

An important step towards true modularity for VMs are metacircular VMs, i.e., VMs written in the same language as they execute. The two most prominent examples for Java™ are the Jikes RVM [16] and the Maxine VM [20]. Maxine is designed with modularity in mind: the core subsystems are exchangeable using so-called "schemes". However, there is still no explicit concept of modules. Additionally, the modularity is on a coarse level. Individual modules are not designed with fine-grained extensibility in mind.

Most current VMs are designed to execute only a single input language. The monolithic design makes it difficult to adapt major parts when building a new VM for a new language. For example, Java VMs have been extensively optimized over the last decade, which has led to aggressively optimizing JIT compilers as well as parallel and concurrent garbage collectors. However, the dynamic language VMs that have become highly popular in recent years do not incorporate these implementation-intensive optimizations. The lack of modularity has prohibited the reuse of existing VM optimizations. With fine-grained modularity, implementing a VM for a new language requires only implementing the new language-specific parts, while the core runtime and optimization infrastructure remains unchanged.

The goal of our research is to investigate the impact of fine-grained modularity on VMs. We envision a VM following the "everything is extensible" paradigm, by combining best practices of existing VM design and existing module systems.

We expect our research to contribute along three axes.

- *Direct research results*: We will study the performance impact of fine-grained modularity in VMs, develop new optimizations to eliminate possible overhead, and generalize the results for the benefit of all modular applications.
- *Benefits for future research*: We will develop a VM research platform that is ideal for future research. It will make comparisons of different optimizations much easier than before.
- *Optimized VMs for many languages*: We will define a customizable family of VMs that can easily be configured for different languages (static and dynamic languages), different target systems (from embedded devices to servers), and different optimization strategies. For some of these configurations, no optimized VMs are available yet, e.g., we expect to contribute significantly to the performance of dynamic language VMs.

2. Vision of a Modular VM

While the performance and complexity of VMs have been greatly increased over the last decade, the internal structure of VMs still neglects the concepts of modern modular software development. Consequently, it is not possible to implement different VM optimizations independently from each other: all parts of a VM interdepend on each other. Figure 1 shows important subsystems of a current VM. There are multiple circular and bidirectional dependencies, e.g., between the JIT compiler and the garbage collector: the JIT compiler emits read and write barriers specifically for a garbage collector, while the garbage collector traverses stack frames and root pointers defined by the JIT compiler.

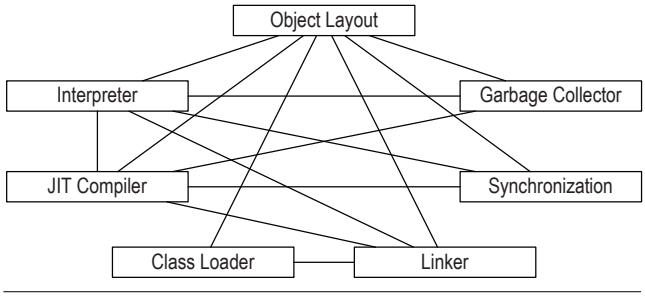


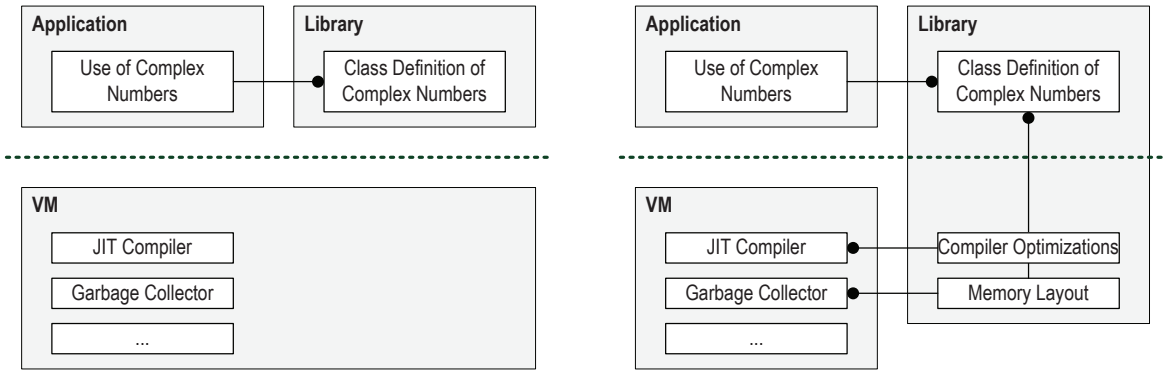
Figure 1. Structure of current VMs.

We want to disentangle the structure of the VM by introducing an explicit concept of modules and module dependencies. Modules provide well-specified extension points that are used by other modules. We outline the main features of the resulting system below.

- *No circular dependencies*: The module system prohibits circular dependencies, leading to an overall structure that is easier to understand.
- *Fine-grained modules*: Every subsystem is split into several modules with explicit dependencies and extension points in between.
- *Ubiquitous extensibility*: There is no distinction between "internal" modules that constitute the core runtime system, and "external" modules supplied by third-party vendors. All use the same extension points and have access to the same data and interfaces.
- *Metacircularity*: The VM is mostly written in one of the many managed languages that it can execute (we anticipate the VM to be written in Java). Modules that extend the VM are shipped in the same form as applications running on top of the VM. Therefore, all optimizations that improve the application speed also improve the VM speed, and it is possible to eliminate the overhead of modularity.

2.1 Performance

Performance is a key aspect for VMs because it affects all applications running on top of the VM. However, we believe that the current VM development process is too centered around performance, thereby sacrificing other important aspects such as maintainability, portability, and extensibility. Many optimizations are applied prematurely because there is



(a) Current Structure (b) Envisioned Structure

Figure 2. Case study for the support of complex numbers inside the VM.

the common belief that they are indispensable. To overcome this, we will follow a three step process.

1. Define the module boundaries so that each module is a small, self-contained, and individually meaningful entity.
2. Measure the performance impact of modularity in various configurations to identify whether there are bottlenecks and where they are.
3. Implement optimizations that eliminate the bottlenecks. Since these optimizations can be implemented as independent modules themselves, they do not erode the overall system architecture.

The ability of a VM to "optimize itself" offers unique benefits compared to, e.g., a modular operating system (OS). In a modular OS, the module boundaries remain in effect at run time. Calls that cross method boundaries are expensive because they require a table lookup to find the current implementation of an interface, or at least an indirect method call using a function pointer. In our modular VM, run-time profiling and the JIT compiler can eliminate the overhead completely. Even in current VMs, interface calls are aggressively optimized when there is only one implementation of the interface available. Therefore, method inlining and other inter-procedural optimizations across module boundaries are feasible. We therefore argue that a modular VM has a vast performance advantage compared to a modular OS.

2.2 Case Study: Third-Party VM Extensions

Suppose that a third party vendor offers a library providing classes for computations with complex numbers. Figure 2(a) illustrates this structure. The library is centered around a class that combines two floating point numbers to one complex number. Therefore, every complex number is a separate object, leading to sub-optimal performance.

The library developer cannot solve this problem because modifications of the VM are not possible. The VM developer cannot solve the problem because implementing optimizations for a certain library is out of scope for a VM—only important methods of the standard library that ships with the VM are optimized by the VM.

Using our approach, the library vendor can also supply extension modules for the VM. Figure 2(b) shows this approach. One VM module can define the object layout for complex numbers, i.e., it defines that complex numbers are handled in the same way as floating point numbers and are not separate objects. Another module can define compiler optimizations for complex numbers, e.g., perform constant folding or use special processor instructions for complex numbers. These extension modules are loaded when the library for complex numbers is loaded, i.e., the modules have no impact on applications that do not use the library.

2.3 Case Study: Mixing of Languages

Figure 3 sketches the continuum of programming languages in a very coarse and superficial manner. At one end of the spectrum are statically typed languages such as *Java* and *C#* that provide type safety and highly optimized execution environments. On the other end are dynamic languages such as *Python, Ruby,* and *JavaScript*. These are more flexible and easier to use, especially for end users. In the middle range are programming languages such as *Visual Basic* that provide some (sometimes optional) static typing while being suitable for casual users. No single language can fulfill the needs for all kinds of programming tasks, so the combination and integration of languages is important.

To get the best of both worlds, a hybrid approach is increasingly used in which the *core* of an application is written by experienced developers in a statically typed language while domain experts and end-users write *extensions* in dynamically typed (and often domain-specific) languages. In many cases, the dynamic language is implemented on top of a static language that also uses a virtual machine environment for its execution (see for example [8, 15]), leading to a big overhead due to double interpretation and compilation.

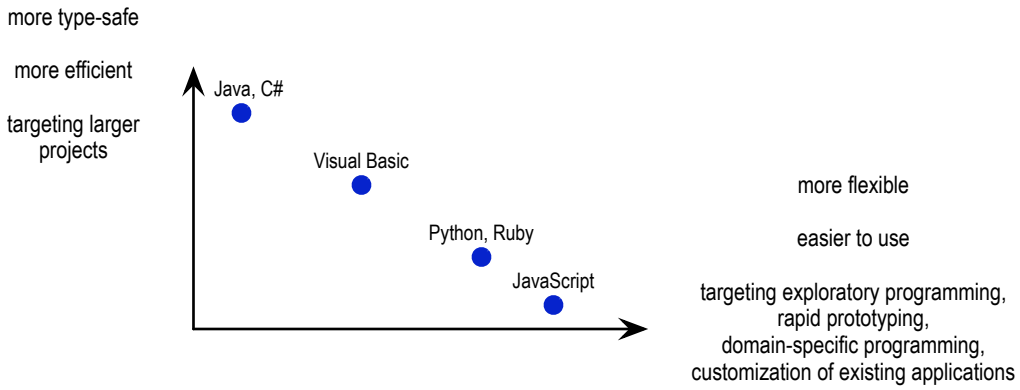


Figure 3. *A continuum of programming language characteristics.*

Our approach to handle multiple languages in one VM addresses the layering cost. All languages use the same implementation infrastructure, so data can be shared between the languages. There is no more need for conversion between different object models and wrapping or unwrapping of data. By leveraging VM extensibility, we also will support highly efficient ways to implement bytecode interpreters. Currently, all of these techniques require the use of unsafe, systems-oriented programming languages such as C and C++.

2.4 Case Study: One VM for Mobile Devices

Smartphones and other mobile or embedded devices have become ubiquitous. Although their computational power has increased, they are still much more limited than PCs. Nevertheless, users want to access web pages that use JavaScript, run Flash applications inside the browser, and execute their well-known rich-client Java applications. All these languages require a VM with the same core functionality, such as a garbage collector, an interpreter, and a JIT compiler. Figure 4(a) illustrates the current situation. It is a waste of resources to ship a mobile device with three completely separate VMs. From a security perspective, a larger code base means more potential vulnerabilities for attackers to exploit.

Using our approach, the core functionality is shared between all VMs, as shown in Figure 4(b). The language specific parts are added as additional modules to the language-independent set of core modules. No functionality is duplicated, saving valuable disk space. Additionally, the different applications can also be executed in one VM *at run time*, provided that the VM is shipped with the appropriate modules that provide isolation between multiple applications running inside one VM. This eliminates the duplication of data structures in the main memory at run time.

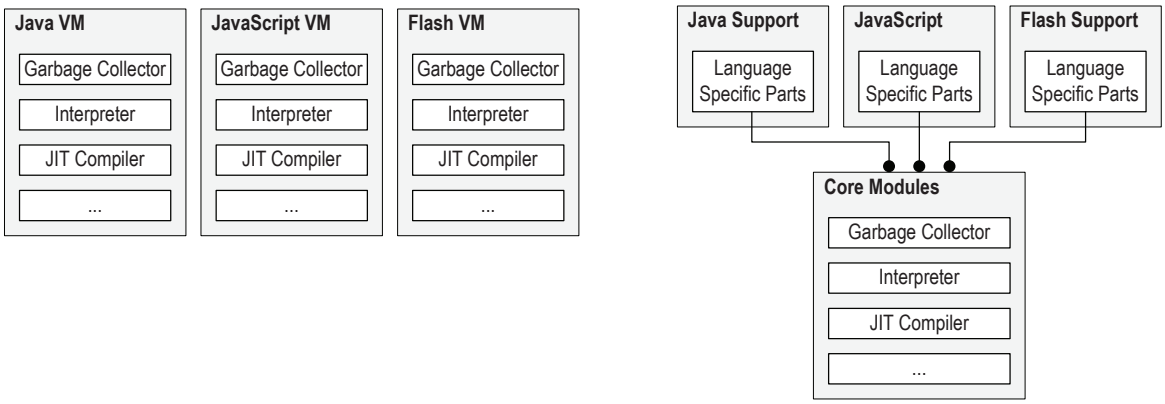
Having a fine-grained set of core module also simplifies customization of the VM for different device configurations. A small-scale device can be shipped with the bare minimum of modules, e.g., without a JIT compiler. The more powerful a device is, the more optimizations can be added. This allows to support the full scale of devices and the full scale of languages with only one customizable VM.

2.5 Case Study: A Multi-Language VM-Based Web Browser to Increase Web Security

The web is one prominent case where the interaction between multiple languages currently poses significant problems. The core of current web browsers is implemented in C or C++, and they execute web applications written in JavaScript, Flash, and other dynamic languages. However, the boundary between the languages is not as clear as it seems at the first glance. JavaScript is also used for core browser components, for example the user interface of Mozilla's Firefox is written in JavaScript. And browser extensions can be implemented in JavaScript too. As a result, current browsers have much communication between parts written in different languages, which requires expensive conversion and duplication of data structures in these worlds.

Our solution allows a browser, where the main components are implemented in a static managed language like Java, to transparently communicate with extensions and web applications written in JavaScript, as well as web applications written in Flash. That enables seamless and combined optimization of the browser core, browser extensions, and web applications running inside the browser. The code handling a single mouse click on a web page button currently crosses the language boundary several times. We allow method inlining and other aggressive optimizations from the code that receives the mouse event to the JavaScript handler and then to the code that accesses and modifies the document object model (DOM) representing the page in memory.

An additional benefit of our approach that we want to highlight is the possibility for increased security. Attacks such as cross-site scripting and deficiencies of the same-origin policy that is currently used by most browsers can be solved by approaches such as fine grained information



(a) Current Structure

(b) Envisioned Structure

Figure 4. Case study for the support of multiple languages in one VM on mobile devices.

flow tracking inside the JavaScript VM. From a conceptional point of view, every JavaScript value has a label with its origin attached. However, when the web page source code with the DOM is part of the browser core written in C or C++, DOM elements cannot be labeled and tracked by the JavaScript VM. Our envisioned VM allows end-to-end information flow tracking in the whole browser. A single implementation of information flow tracking secures the browser and web applications, so our VM is the ideal basis for future research in this area.

2.6 Metacircular VMs

As previously mentioned, traditional VMs are written in a language different from the one they execute. In many cases, the VM is implemented using a low-level or system oriented programming language like C or C++, while the VM itself executes a high-level programming language. The VM offers productivity advantages such as type safety, memory safety, garbage collection, and JIT compilation, but the VM itself does not benefit from these features.

In contrast, metacircular VMs [14, 16, 20, 28] are implemented in the same programming language they execute. Both the application and VM code are treated uniformly. There is no internal distinction between parts of the VM and parts of the application. Metacircular design is advantageous in terms of both performance and development time. For example, the JIT compiler of the VM optimizes both the VM and application code together in the same context. A VM with run-time profiling not only optimizes the running application, but also the VM itself. Similarly, a VM with run-time profiling makes no distinction between optimizing the VM itself and the application it runs.

Although there is no difference between VM and application code inside a metacircular VM, the outside view is still different. The VM requires a bootstrapping process that translates the VM to machine code ahead of time. This requires a second, non-metacircular VM for the language, so that the JIT compiler can run the first time and compile itself. The result of the bootstrapping process is a machine code image of the VM that can be executed directly by the processor. For all VMs we are currently aware of, the image is a monolithic piece of machine code. We want to evaluate whether splitting the file into several smaller parts and preserving the module structure, i.e., a reduced image that covers only essential parts, is sufficient and beneficial.

2.7 Reliability

A modular VM design embraces and encourages extension from third-party vendors. That can raise concerns about the reliability of the VM. Code from different vendors, which is unlikely to have been tested together, must interact. Several parts of a modular VM are quite similar to ordinary modular applications. They do not need access to low-level data structures such as raw memory, threads, or synchronization primitives. For example, an optimization for the JIT compiler transforms high-level graph-based data structures and not raw memory. The dependency tracking of the module system enforces that such access does not happen. Additionally, modules that contribute optimizations are not critical for the overall functionality of the VM and can be disabled when they show erroneous behavior.

Still, the machine code generated by the JIT compiler operates on raw memory, so an erroneous optimization can lead to code that crashes the VM; and modules extending the garbage collector require arbitrary memory access to work. This problem can be tackled for example with integrity checks performed by the core modules or with module interfaces that encourage fault-tolerant programming.

We consider it an essential part of this research to evaluate reliability issues. This includes both a formal analysis of our prototype implementation, as well as field studies that combine our core modules with third-party extensions.

In addition to guaranteeing this low-level system integrity, we realize the danger of interference between third-

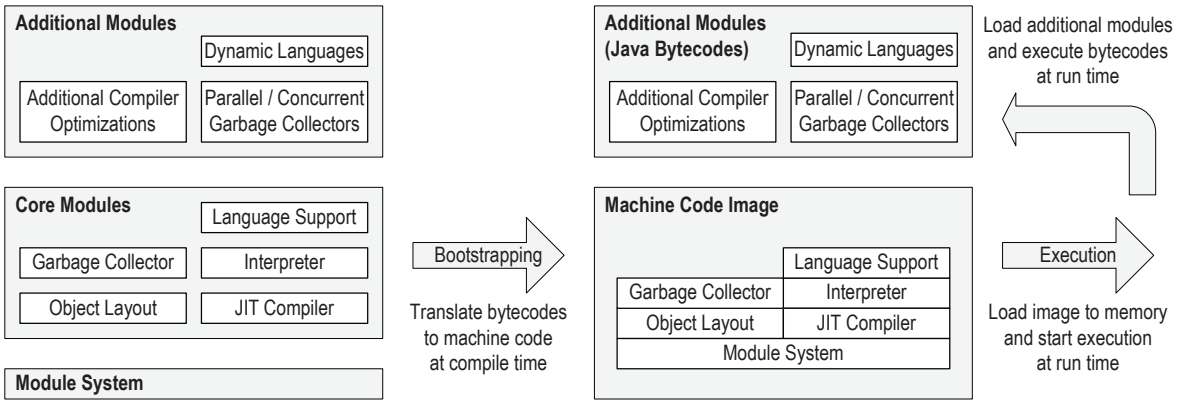


Figure 5. Architecture of the modular VM.

party modules. For example, our envisioned modular VM indicates a conflict, if there are two modules A and B, both altering the same code fragment, such as the same class, or method. In such a case, there are basically several ways to proceed. First, we resolve conflicts by restricting the access to the same piece of code such that we give permission to only one of the two conflicting modules. Next, we think that both modifications are permissible if both modules share a common namespace, similar to the same-origin-policy in web browsers. Finally, the conflict is analogous to conflicting compiler optimizations (corresponding to modules): before actually performing any transformation, each optimization ensures soundness by ensuring that its preconditions hold.

2.8 System Architecture

We envision a VM where all subsystems are split into small modules with well-defined module boundaries. On the lowest level, a module system manages the dependencies between the modules and allows module interactions in a controlled way. Figure 5 shows the overall structure. On top of the module system, a small set of core modules provides the basic runtime environment of the VM. During the bootstrapping process, this part must be compiled to machine code to get an initial running VM image. Additional modules with advanced optimizations are loaded on demand. These modules are shipped as bytecodes, so they are executed and optimized like any application running on top of the VM.

Modules are kept at the smallest feasible granularity to allow fine-grained configurability and extensibility. As an example, Figure 6 shows a possible structure of the optimizing JIT compiler. The intermediate representation and flow of compilation is defined in a central module. All other parts, especially the individual optimizations, are in separate modules that depend on the central module. Additionally, other subsystems contribute to the compilation process, such as the garbage collector, the run-time profiling, and the definition of language-specific semantics. It is also possible to integrate optimizations contributed by third-party vendors. This breaks up the current paradigm that mandates the whole VM to be defined, maintained, and shipped by a single vendor.

Splitting up a VM into modules separates the classic subsystems such as the interpreter, JIT compiler, and garbage collector, from each other. It also separates the language-specific parts from language-independent parts on a fine-grained level. For example, the JIT compiler of a current Java VM contains many global optimizations that are applicable for any language, but also Java-specific parts such as method and field linking. Separating these parts into different modules allows for an easy re-use of language-independent VM parts.

When building the VM for a new language, it is consequently only necessary to define the language-dependent parts for the new language, which is most likely only a small subset of the whole VM. This reuse of modules is especially important in the context of emerging dynamic languages. Instead of building a whole new VM for every language, our approach requires only a small addition to the existing VM framework. To demonstrate this potential, we are in the process of adding support for an additional language to an existing VM.

3. Expected Outcomes

Our project is expected to yield new insights and a design rationale for modular VMs. We have no desire to re-invent the wheel. Hence, our work benefits intelligently from prior research and existing open-source projects. For example, we are using an existing Java VM—Maxine from Oracle Labs [20]—which is already clearly divided into independent subsystems as the basis for our host VM. Maxine is our VM of choice because it is a well-structured Java VM and it is available under an open source license.

Converting Maxine into a modular VM is non-trivial. While high-level components such as the garbage collection algorithm, the layout of objects, and the JIT compiler are exchangeable, it is currently not designed to be modular. Additionally, it is tailored towards the Java VM specification,

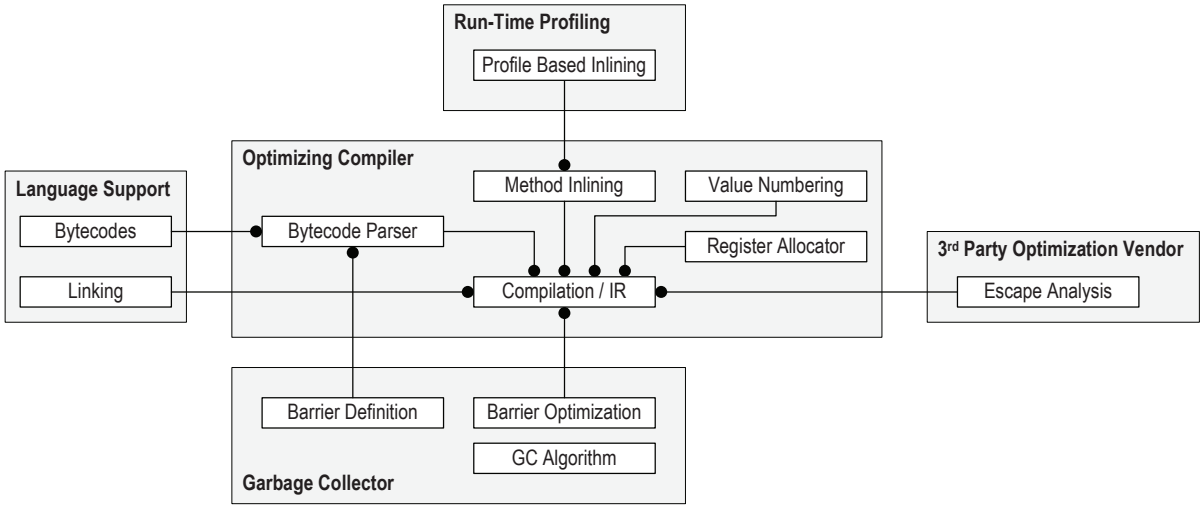


Figure 6. Examples of fine-grained modules for JIT compilation.

and not ready to support multiple languages. For example, the semantics and constraints of Java bytecodes are visible throughout the whole VM, from the class loader to the JIT compiler.

Even the recently added C1X JIT compiler provides no support for different input languages; it only provides an interface between the compiler and the rest of the VM using the intermediate language XIR [31]. We will use C1X as the starting point, but we need to open up the intermediate representation to support both static and dynamic typing, and we need to incorporate run-time type feedback in the compiler. Significant research is necessary throughout the whole VM to isolate the Java-specific parts from the language agnostic parts and re-connect them using a well-defined module interface.

There are many high-quality implementations of dynamic languages available as open-source projects. We want to support at least one dynamic language in addition to Java. For all major dynamic languages, VMs written in Java are available as open-source projects. Among applicable candidates are the Rhino JavaScript VM [27], the JRuby VM for Ruby [17], and the Jython VM for Python [18]. All these VMs currently run *on top* of a Java VM, i.e., from the point of view of the Java VM they are applications. This leads to double interpretation and compilation overhead, which is no longer present in our approach. Incorporating code from an existing VM not only saves development resources, but also allows direct comparison with the original VM. We expect our dynamic VM to be faster than the original because it leverages the optimization system designed to accommodate dynamic languages. However, our system is not limited to existing dynamic languages, but also simplifies and encourages the development of new domain-specific and simple end-user programming languages targeted for special limited programming tasks.

In summary, our research has four expected outcomes.

- The definition of a module system at the VM level. We believe this will significantly lower the cost of supporting new languages efficiently.
- A set of new optimizations that eliminate the possible overhead of modularity and layering of languages.
- Evaluations and experiments with different module configurations, especially with different amounts of code to be bootstrapped.
- A platform and baseline for other researchers that simplifies the development of new optimizations and the comparison of optimizations of different research groups.

4. Realizing the Vision

In previous work, we have experienced both the shortcomings of existing VM structures and the benefits of modular applications. Based on the insights and experience, we are confident that a broad and systematic research on this topic can revolutionize the implementation of VMs.

4.1 Enabling Modularization

The first and fundamental basic step is to define an explicit module structure for VMs. As we do not want to reinvent the wheel, we are using the existing Maxine VM.

The initial module structure is rather coarse-grained and matches the existing subsystem boundaries of Maxine. In future steps, we will then refine the modules and split them to get the desired fine-grained modularity. This process includes the definition of suitable extension points for all parts of the VM. Once all modules are present, we will investigate the minimum set of modules that are necessary to run a Java application.

To get a broader variety of modules and configurations, we will then add a trace-based JIT compiler as an alternative

to the existing JIT compiler. This trace-based JIT compiler re-uses several of Maxine's compilation systems. Its integration in the existing system architecture is the first proof-of-concept that the module interfaces are useful to re-use parts of the existing compiler.

Finally, we will start to separate the Java-specific parts into separate modules. This is the first step towards a language-independent VM where special language features are supplied using additional modules.

4.2 Exploiting Modularity

In the second step, we want to add support for a dynamic language. Since all major dynamic languages (e.g., Python, Ruby, or JavaScript) have VMs available written in Java, a smooth integration is possible. We plan to modularize the dynamic language VM and strip away the parts that are already covered by Maxine. The challenge is to re-use as many modules as possible from the previous step. For example, the optimizing JIT compiler can be re-used immediately without larger changes, while the garbage collector needs to be opened to support dynamically growing objects. Therefore, the process of integrating the dynamic language also involves a broadening of the existing modules by providing even more extension points. The resulting dynamic language VM has the potential to outperform existing VMs because it leverages compiler and garbage collector optimizations that were not previously available for dynamic languages.

4.3 Eliminating Module Overhead

In the final stage, we will analyze and optimize inter-module calls in the VM. Following the spirit of feedback-directed dynamic optimizations, we will implement optimizations that specifically target frequently executed inter-module calls. Using profile information and dynamic code specialization, overhead introduced by modularization can be eliminated.

Additionally, we will develop inter-language optimizations. The modular VM approach enables support for multiple static and dynamic languages inside one VM. All code is optimized together, so we can study the commonalities and the differences in the respective optimization approaches. This will demonstrate the full power of our modular VM approach.

Finally, we will perform a quantitative and qualitative analysis of the module overhead. By studying different methods of bootstrapping, the full spectrum from the minimal set of modules to the whole VM can be bootstrapped, while the remaining modules are loaded on demand. This evaluation will answer our primary research question: which amount of modularity and bootstrapping is best for a VM.

5. Related Work

Most research and production quality VMs are internally designed in a fairly modular way. The different subsystems are separated in different packages, namespaces, directories, or other means offered by the language the respective VM is written in. The "modularity" is only expressed as different namespaces and directories and not in minimized dependencies and clear interfaces, and no explicit module system is present. Therefore, we do not consider these VMs to be modular in a fine-grained way. Additionally, these VMs are not extensible at run time. Only a small number of research projects explicitly target the VM structure and modularity.

Haupt et al. propose to disentangle VM architectures using an explicit architecture description language [12, 13]. They use the *VM Architecture Description Language* to describe the interface of a module. It is a mixture of declarative specifications and actual code that is merged into the VM source code. The merging is similar to code weaving of aspect oriented programming. All configuration is performed at compile time, so the resulting VM is not extensible at run time. They then define a product line of VMs using a standard product line modeling tool. Our approach does not require the definition of a separate architecture description language. Instead, we propose to embed the module definitions directly into the source code.

Thomas et al. introduce a *Micro Virtual Machine* (MVM) that is then extended to the JnJVM [30]. MVM is a minimal but nevertheless complete VM that consists of a code loader, an extensible JIT compiler, and I/O functionality. Additionally, it is extensible using aspect oriented programming techniques. For this, MVM includes an aspect weaver that can integrate new VM parts at run time. A small Lisp-like language is used to specify the extensions. JnJVM is a Java VM developed on top of MVM. It is written in the Lisp-like language, and the MVM compiles the JnJVM on the fly when it is loaded. This leads to a high run-time overhead. Our solution will perform all source code compilation at compile time, and provide the option to optimize and combine the modules either at compile time or run time.

Harris presents a prototype *eXtensible Virtual Machine* (XVM) [11] that allows application code to interact with the VM. Because of the inherent safety risk of untrusted application code modifying the VM, the interaction is limited. We do not plan to have direct interactions of applications with the VM (although such modules would be possible to implement), instead we focus on extensibility inside the VM.

Geoffray et al. present I-JVM [9], a Java virtual machine for component isolation in OSGi. They address problems of module isolation. One malicious module can accidentally or deliberately crash the complete application platform. I-JVM isolates the modules so that they can be reliably terminated without affecting other modules. The VM itself is not based on OSGi. However, their ideas on module isolation apply to our envisioned modular VM.

Metacircularity originates from LISP [21]. Its `eval()` function requires a LISP interpreter, which was defined in

LISP itself. Also, the first successful LISP compiler was already developed in LISP.

The idea was then applied to other languages. For example, the programming environments and compilers for Oberon [33] and Cecil [4] 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.

Similar ideas apply to Pascal p-Code, one of the predecessors of modern bytecodes [26]. In this case, even whole operating systems were written in the system, allowing the operating system and all applications to be ported easily to different platforms. On the lowest level, p-Code was still interpreted by a small runtime layer that remained separated from the code it executed.

In Smalltalk [7], 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. Only the "blue book" reference implementation was written in Smalltalk itself [10], but it was intended only for illustrative purposes.

Squeak [14] is a metacircular Smalltalk VM. It is written in a subset of Smalltalk: a non-object-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.

SELF [5] is a language that offers even more dynamic reflective facilities. Every method dispatch is dynamic and can be changed a run time. 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* [32] is a metacircular SELF VM written entirely in SELF. The dynamic nature and run-time configurability of this VM probably makes it the VM most closely related to the system we envision. 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. Rather than minimizing the codebase, we seek to make a different set of tradeoffs. We will leverage metacircularity and modularity to support multiple languages while retaining high performance through new, feedback-directed optimizations.

PyPy [28] is a VM for Python, based on a framework useable for any dynamic language, written in Python. It is an example of a dynamic language VM written in a dynamic language itself. Run-time optimizations are performed by a trace-based JIT compiler. During bootstrapping, the VM code is either translated to C code or to the .NET common intermediate language. This requires an additional C compiler or .NET runtime environment for execution.

The two major metacircular research Java VMs are Jikes [2, 16], which was originally developed by IBM and is now used in many research projects, and Maxine [20], which is a novel research VM developed by Oracle. The execution environment of Jikes is fairly modular and provides different JIT compilers, and its *Memory Manager Toolkit* (MMTk) allows to integrate new garbage collection algorithms. However, there is no explicit concept of modules in the VM. Maxine provides modularity using the concepts of schemes and snippets. A scheme encapsulates the different subsystems, e.g., there is a scheme that specifies how fields are accessed. The code for the field access is then compiled to a snippet when the VM is started, and the JIT compiler uses the snippet when compiling field access bytecodes. Although this design disentangles different VM subsystems quite well, there is still no extensibility at run time and no fine-grained modularity.

The *Open Runtime Platform* (ORP) [6] developed by Intel is a research VM targeted not only towards multiple architectures, but also towards multiple languages. It can run either Java or Common Language Infrastructure (CLI) applications by applying the compile-only approach. It is available as open source from [23], but this version seems to be quite old and does not reflect the current development version of Intel. ORP offers a flexible compiler interface, and different JIT compilers were developed. An example is the *StarJIT* compiler [1].

The *Dynamic Runtime Layer Virtual Machine* (DRLVM) of Apache Harmony [3] is a Java VM written in C++. Modularity is a key feature of the VM: it is separated in a small number of coarse-grained modules with well-defined interfaces. Modules are compiled to separate libraries that are dynamically linked at run time. However, inter-module calls cannot be optimized at run time because the VM is written in C++. There is no fine-grained modularity.

The Java HotSpot™ VM of Oracle [22] is a production-quality open-source Java VM written in C++. The source code base contains two interpreters, two JIT compilers, and multiple different garbage collection algorithms (some of them being parallel or concurrent). However, these parts are not separated into modules, so it is sometimes difficult to identify the boundaries of and interfaces between these subsystems in the source code. Only the compiler interface, which separates the client compiler [19] and the server compiler [25] from the rest of the VM, is explicitly defined.

6. Conclusions

Modularity for virtual machines is a promising research area: our project has the ability to completely change the way production-quality virtual machines are built. When companies see that modular VMs are possible without performance overhead, they will quickly adapt this paradigm and use it when they design new VMs, or even retrofit their existing VMs.

Our preliminary work extends the Maxine VM and serves as a foundation towards addressing issues central to the three mentioned case studies. By doing so, we expect to validate our research hypotheses and in turn inspire further research on modular VM architectures.

Acknowledgments

Parts of this effort have been sponsored by the National Science Foundation under grant CCF-1117162 and by Samsung Telecommunications America under Agreement No. 51070. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the National Science Foundation (NSF), nor that of Samsung Telecommunications America. The authors also gratefully acknowledges gifts from Adobe and Google.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

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, 2003.

[2] B. Alpern, C. R. 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. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] Apache. *Apache Harmony, Dynamic Runtime Layer Virtual Machine*, 2010. http://harmony.apache.org/subcomponents/drlvm/.

[4] C. Chambers. The Cecil language specification and rationale, version 3.0. Technical report, Department of Computer Science and Engineering, University of Washington, 1998.

[5] C. Chambers, D. Ungar, and E. Lee. 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*, pages 49–70. ACM Press, 1989. doi: 10.1145/74878.74884.

[6] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The open runtime platform: a flexible high-performance managed runtime environment. *Concurrency and Computation: Practice and Experience*, 17(5-6):617–637, 2005. doi: 10.1002/cpe.852.

[7] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–302. ACM Press, 1984. doi: 10.1145/800017.800542.

[8] Dynamic Language Runtime. *Dynamic Language Runtime*, 2010. http://dlr.codeplex.com/.

[9] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot. I-JVM: a Java virtual machine for component isolation in OSGi. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 544–553. IEEE Computer Society, 2009. doi: 10.1109/DSN.2009.5270296.

[10] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[11] T. L. Harris. *Extensible Virtual Machines*. PhD thesis, Computer Laboratory, University of Cambridge, UK, 2001.

[12] M. Haupt, B. Adams, S. Timbermont, C. Gibbs, Y. Coady, and R. Hirschfeld. Disentangling virtual machine architecture. *IET Software*, 3:201–218, 2009. doi: 10.1049/iet-sen.2007.0121.

[13] M. Haupt, S. Marr, and R. Hirschfeld. CSOM/PL - A virtual machine product line. *Journal of Object Technology*, 10:12:1–30, 2011. doi: 10.5381/jot.2011.10.1.a12.

[14] 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 *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 318–326. ACM Press, 1997. doi: 10.1145/263698.263754.

[15] Java Specification Request 223. *Java Specification Request 223: Scripting for the Java™ Platform*, 2006. http://www.jcp.org/en/jsr/detail?id=223.

[16] Jikes. *Jikes RVM*, 2010. http://www.jikesrvm.org/.

[17] JRuby. *JRuby*, 2010. http://www.jruby.org/.

[18] Jython. *Jython*, 2010. http://www.jython.org//.

[19] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):Article 7, 2008. doi: 10.1145/1369396.1370017.

[20] Maxine. *Maxine Research Virtual Machine*, 2010. https://wikis.oracle.com/display/MaxineVM/.

[21] J. McCarthy. History of LISP. In *Proceedings of History of Programming Languages*, pages 173–185. ACM Press, 1978. doi: 10.1145/960118.808387.

[22] Oracle. *The Java HotSpot Performance Engine Architecture*, 2006. http://www.oracle.com/technetwork/java/whitepaper-135217.html.

[23] ORP. *Open Runtime Platform*, 2010. Intel Corp. http://sourceforge.net/projects/orp/.

[24] OSGi. *OSGi - The Dynamic Module System for Java*, 2010. http://www.osgi.org/.

[25] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 1–12. USENIX, 2001.

[26] S. Pemberton and M. Daniels. *Pascal Implementation: The P4 Compiler and Interpreter*. Ellis Horwood, 1983.

[27] Rhino. *Rhino: JavaScript for Java*, 2010. http://www.mozilla.org/rhino/.

[28] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 944–953. ACM Press, 2006. doi: 10.1145/1176617.1176753.

[29] R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 499–514. ACM Press, 2007. doi: 10.1145/1297027.1297064.

[30] G. Thomas, N. Geoffray, C. Clément, and B. Folliot. Designing highly flexible virtual machines: The JnJVM experience. *Software: Practice and Experience*, 38(15):1643–1675, 2008. doi: 10.1002/spe.887.

[31] B. L. Titzer, T. Würthinger, D. Simon, and M. Cintra. Improving compiler-runtime separation with XIR. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 39–50. ACM Press, 2010. doi: 10.1145/1735997.1736005.

[32] D. Ungar, A. Spitz, and A. Ausch. 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*, pages 11–20. ACM Press, 2005. doi: 10.1145/1094855.1094865.

[33] N. Wirth and J. Gutknecht. *Project Oberon*. Addison-Wesley, 1992.