Garbage Collection and Efficiency in Dynamic Metacircular Runtimes

An Experience Report

Javier Pimás
Palantir Solutions
javierpimas@gmail.com

Javier Burroni
javier.burroni@gmail.com

Jean Baptiste Arnaud
Palantir Solutions
jbarnaud@caesarsystems.com

Stefan Marr
Johannes Kepler University
Linz, Austria
stefan.marr@jku.at

Abstract

In dynamic object-oriented languages, low-level mechanisms such as just-in-time compilation, object allocation, garbage collection (GC) and method dispatch are often handled by virtual machines (VMs). VMs are typically implemented using static languages, allowing only few changes at run time. In such systems, the VM is not part of the language and interfaces to memory management or method dispatch are fixed, not allowing for arbitrary adaptation. Furthermore, the implementation can typically not be inspected or debugged with standard tools used to work on application code.

This paper reports on our experience building Bee, a dynamic Smalltalk runtime, written in Smalltalk. Bee is a Dynamic Metacircular Runtime (DMR) and seamlessly integrates the VM into the application and thereby overcomes many restrictions of classic VMs, for instance by allowing arbitrary code modifications of the VM at run time. Furthermore, the approach enables developers to use their standard tools for application code also for the VM, allowing them to inspect, debug, understand, and modify a DMR seamlessly.

We detail our experience of implementing GC, compilation, and optimizations in a DMR. We discuss examples where we found that DMRs can improve understanding of the system, provide tighter control of the software stack, and facilitate research. We also show that in high-level benchmarks the Bee DMR performance is close to that of a widely used Smalltalk VM.

CCS Concepts •**Software and its engineering → Object oriented languages; Runtime environments;** *Garbage collection; Dynamic compilers;*

Keywords runtimes, dynamic, metacircular, gc, efficiency

ACM Reference format:
Javier Pimás, Javier Burroni, Jean Baptiste Arnaud, and Stefan Marr. 2017. Garbage Collection and Efficiency in Dynamic Metacircular Runtimes. In *Proceedings of 13th ACM SIGPLAN International Symposium on Dynamic Languages, Vancouver, Canada, October 25–27, 2017 (DLS'17),* 12 pages.
DOI: 10.1145/3133841.3133845

1 Introduction

Over the last few decades, more and more kinds of applications are written in high-level languages. Over time this trend slowly reached virtual machine development, too. Engineers who previously wrote most code in assembly started migrating to languages like C and C++ to take advantage of their features and abstractions. The trend did not stop with systems programming languages however. Some developers wanted to make use of the advantages that the languages they implemented provided, and started implementing VMs in even higher-level languages such as Java [1, 3, 34], Python [27], JavaScript [7], and Smalltalk [15]. These projects proved that self-hosted VMs can support the full set of features of object-oriented (OO) languages. However, while for instance PyPy, Tachyon and Squeak were written in dynamic languages that allow changing code at run time, none of those projects allow changing the VM itself at run time. They do not integrate the models of the VM and the hosted language. Instead, they only expose a restricted interface through which the language and the VM communicate. The implementation details of these self-hosted VMs are static and mostly inaccessible at high level.

This work pushes this trend even further: we implement an OO VM in a highly dynamic language, Smalltalk, and enable the modification of its components freely at run time, as application-level objects in a Smalltalk system. We report on our experience with a self-hosted OO VMs with the following characteristics: (i) it is dynamically typed, as defined in the next section, (ii) it uses garbage collection, (iii) and it

DLS'17, Vancouver, Canada

facilitates changing code arbitrarily at run time. We refer to systems with these properties as *dynamic languages*.

While there have been similar projects in the past, which provided support for dynamic changes of the runtime [31, 33], none of them were able to create a complete VM that supported all the features of the language it implemented and run as fast as the original VMs. Specifically, they did not implement a garbage collector (GC) in the language and their performance was poor.

Until today, VMs for dynamic languages support only few or no dynamic changes, even if the VMs were written in dynamic languages. We address this situation in this paper.

Our experience report focuses on the following aspects:

- We identify the concept of dynamic metacircular runtimes (DMR) and we describe the problems that a DMR implemented in a metacircular environment such as Smalltalk has to solve to support all of the environment features.
- We present a solution to these problems with our Bee[1] runtime, which is a full-featured DMR for Smalltalk. We report on our experience implementing different GCs and the optimizations necessary to compete with comparable VMs written in low-level languages.
- Furthermore, we discuss the advantages we found for such a dynamic runtime and give examples where these kinds of exploratory environments are beneficial for analysis and research.

2 Dynamic Metacircular Environments

To provide context for this work, we first analyze the main concepts covered in this paper.

2.1 Terminology and Main Concepts

This section defines what we refer to as dynamic languages. It also describes self-hosting and metacircularity and how these concepts affect the implementation of VMs. Furthermore, it reviews related work and identifies the problems that need to be solved for dynamic metacircular runtimes for metacircular environments.

Common nomenclature. We use the word *VM* to refer to the set of virtual machine components shown in figure 1. Similarly, we use *runtime* to talk about the wider concept that in our work, by including the objects of the language, also includes the VM objects. This term should not be confused with the term *run time*, which refers to the time at which a program is run. Finally, we say *environment* instead of *language* when referring to dynamic aspects of a language: the language in its execution context, as can be Python with its command line interface or JavaScript within a browser, where new code can be added and executed, and objects can be freely inspected.

[1] http://beesmalltalk.org

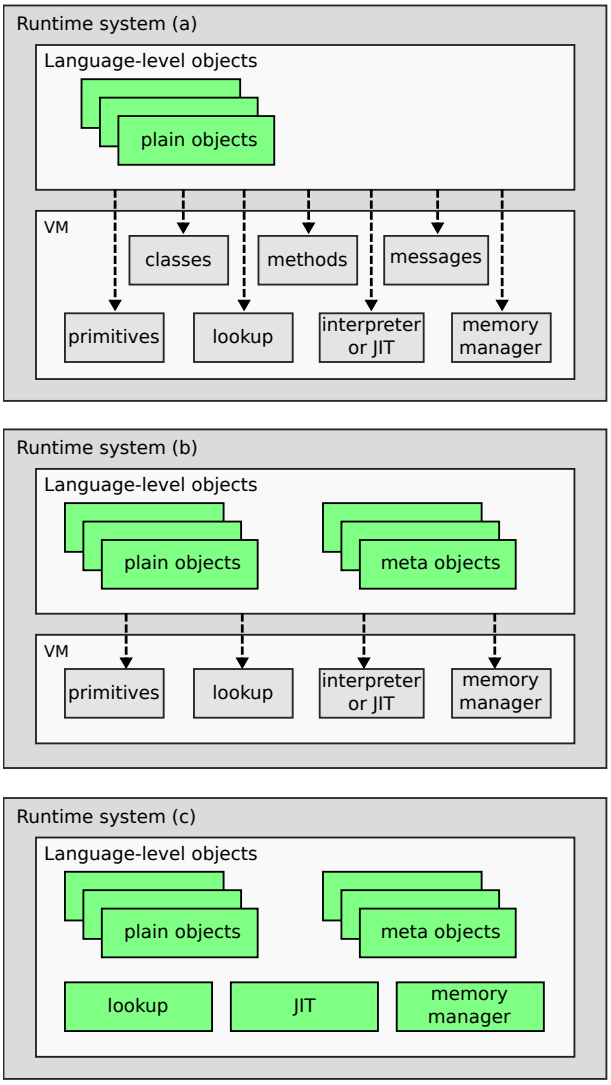

Figure 1. Different interfaces between language and runtime in OO languages, ranging from little or no access (HotSpot), to moderate access (CPython, Squeak) to full access (Bee, Klein)

Dynamic languages. A wide range of high-level languages are commonly referred to as *dynamic* languages. There are many different aspects of their "dynamicity". We focus on three aspects:

(i) Ability to change arbitrary code at run time, which enables programmers to run new code without having to restart the system.
(ii) The language neither allows type annotations or relies on type inference, e.g., JavaScript, Python or Smalltalk.[2]
(iii) Use of automatic memory management, which avoids common resource management bugs.

[2] We refer to these languages as *dynamically-typed* languages.

We use the terms *dynamic language* and *dynamic environment* to refer to systems with the above three properties, specifically in the area of OO languages.

Metacircularity. There are various definitions of metacircularity. We use Ungar et al.'s definition [31], and say that a system is *metacircular*, if it is defined in terms of the constructs it provides. Based on this definition, Smalltalk's metamodel is metacircular, as most of its own concepts (objects, messages, classes, methods) are defined in terms of themselves.

Self-hosted VMs and metacircular VMs. A programming language implementation is *self-hosted* if its build tool-chain is written in the same language. Self-hosted implementations require to be *bootstrapped*. That is, to provide an early way of executing the compilation tool-chain, after which the language implementation can generate new versions of itself. A VM can be implemented in the language it implements. Such a VM is self-hosted if it includes the execution mechanisms to generate new versions of itself. That VM could also be metacircular, if defined using its own constructs. In truly metacircular VMs, programs of the language can access the running VM components using the same constructs with which the VM is programmed. Squeak and PyPy are examples of projects that use a self-hosted but not truly metacircular VM. Once the code of those VMs is transformed to be executed (to C and then native code), the VM model becomes fixed and mostly inaccessible from programs *at run time*.

VMs implemented in higher level languages. Figure 1 sketches runtime systems with varying amounts of reflection capabilities and metacircularity. In (a), objects communicate with the VM through an exposed interface that gives limited access to its services. Through this interface the language-level code can trigger primitives, garbage collection, message lookup or JIT-compilation. The supported language objects and the VM model live in different worlds, making it hard to access the VM model and much more difficult to change it dynamically. In (b), the language acquires a richer model of meta objects like classes, methods and messages, gaining the ability of modifying them at run time. Smalltalk and Self are typical examples of these kind of runtime. In (c), all VM components have been incorporated at language level, and have become just more runtime library objects. Bee and Klein are examples of this.

Metacircular runtimes combined with dynamic environments. In a dynamic language, the VM model can be retrofitted into the language itself, making the entire runtime system dynamic. In such a system, the VM gets combined with the language objects as just more objects. We say that these systems are completely metacircular, because *all* of their concepts are modeled in and accessible by the language. We call them dynamic metacircular runtimes (DMR), and this is the focus of our work.

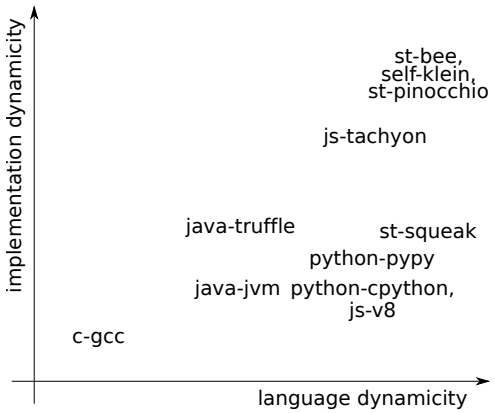

Figure 2. Language dynamicity vs. VM implementation dynamicity

2.2 Philosophy

The main idea driving this research is that *nothing that can be implemented at the high level should stay implemented at the low level.* Concepts that are typically realized as low-level parts of a system, e.g., primitives, compilers and debuggers, should be realized in the high level language as the rest of the system is. Some researchers advocate for high-level low-level programming, arguing that productivity is improved while performance impacts can be overcome and reduced to negligible levels [11].

2.3 Related Work and Background

One of the first self-hosted VMs for an OO language was Squeak [15]. While its code uses a Smalltalk syntax, it does not have an object-oriented design and is not dynamic. Its source code is directly translated to C and then compiled. However, a simulator helps debugging the VM with Smalltalk tools by executing the code as Smalltalk.

There are also a number of self-hosted virtual machines in Java [2, 3, 34]. As Java programs allow only limited code modifications at run time, all of these VMs share the same property: their behavior is mainly defined at compile time, and it is hard or not possible at all to arbitrarily modify things like lookup algorithm, GC or JIT compiler at runtime. As a way to implement support for other languages, Würthinger et al. [35] propose to reuse existing VM infrastructure, combining a Java JIT compiler, which is written in Java, and AST interpreters, also written in Java. They show how support code for new languages can easily reuse optimization components of another *host* VM to be efficient [14, 36].

The PyPy project [27] takes a similar approach, providing tools to implement languages as interpreters, while reusing common elements such as a meta-tracing JIT compiler and the garbage collector. The interpreters are implemented in a subset of Python called RPython. This makes it possible to apply type inference to the code of the VM and generate

C sources that can finally be compiled. Again, the kernel of the VM remains static. Tachyon [7] uses its own IR compiler while maintaining the separation between the VM and the language objects.

Klein [31] and Pinocchio [33] are proof-of-concept implementations of DMRs for Self and Smalltalk respectively, reducing barriers between applications and VMs. Klein allowed VM programming at run time. Figure 2 depicts how the dynamicity of these implementations and the supported languages compare to each other. Unfortunately, none of these projects reached completion, nor did they achieve a fully working environment. For example, Klein did not reach the full set of features that the Self VM written in C++ provided: object modification at run time was not possible in the debugging environment, garbage collection was missing and performance was not studied.

Redmond and Cahill [26] and more recently Chari et al. [4, 5] propose alternative approaches, extending reflection [29] to VMs through metaobject protocols [18]. Those approaches can allow the implementation of highly adaptable VMs without compromising performance. However, unlike our solution, while they allow adapting the VM to unanticipated scenarios, the different capabilities that can be switched at run time (like adding immutability to objects) have to be anticipated and coded before the system is started.

Previous Bee iteration

A first iteration of Bee has been described in [25]; this implementation was a first indication of the feasibility of our approach but was not mature enough to validate key aspects of the system: it run orders of magnitude slower than the original VM and its garbage collector was not able to run in the bootstrapped system, but only attached to the original VM. Debugging of the self-hosted system had to be done with low-level tools, and no GUI was available. This is not the case anymore.

3 Obstacles to Dynamic Metacircular Runtimes

The implementation of a DMR entails different needs compared to other kinds of VMs. The previous iteration of Bee [25] is the most related and complete work, which showed the way to both efficiently accessing low-level elements such as compilers or object headers and cut chicken-or-egg circularities in the system. However, two main problems were left unsolved then:

(i) Implementing a garbage collector that is able to traverse not only the application heap but also the runtime heap.
(ii) Matching performance levels of comparable VMs while being written using dynamic languages.

While solutions to those problems have been developed for similar projects (cf. section 2.3), it was not shown that they are applicable to DMRs. In this paper, we demonstrate that by combining different approaches it is possible to implement a fully-featured DMR with acceptable levels of performance. This section details the challenges tackled in our work, that allow a DMR to support all the features of an OO environment such as Python, Ruby, Smalltalk, or Self. Solutions to these problems, both in Bee and in other implementations are discussed in section 4.

3.1 Garbage Collection

In metacircular environments, entities that represent shape and behavior of objects are also objects. Classes, metaclasses, methods, method dictionaries, processes and threads are objects. In DMRs, the garbage collector model is placed at the language level. There is no low-level interface to the garbage collector functionalities, but only a high-level unified model where *everything* is an object.

3.1.1 Garbage Collection of Runtime Objects

A first challenge is that a DMR adds runtime entities to the set of objects of metacircular environments. This includes allocation spaces, stacks, arrays, memory and even the GC itself and its objects are represented by regular instances of classes. Thus, by tracing the root pointers of a program, the GC will find itself, and all the other objects that are needed by its own implementation. Examples of these objects are the GC instance itself, its class, methods, spaces, and even the stacks. The transitive closure of these objects is also needed.[3] The garbage collector needs to determine which objects have to be followed and which should not. Objects created temporarily by the collector should not be left allocated consuming memory after collection finishes, and unlike classic VMs, the runtime needs to have its own objects be collected from time to time.

Squeak and PyPy do not suffer from this problem, as their code is translated to C. Truffle/Graal does not implement a GC but relies on the one of Hotspot, which is written at the lower C++ level. Other Java VMs like Maxine and Jikes do model VM concepts with objects and implement a GC that traverses those objects. However, as the VM code cannot be modified at run time, objects representing parts of the VM like the GC itself can be fixed in memory and do not require moving. Tachyon and Klein do not provide a GC.

In section 4.1 we describe how to implement GC in a DMR, while at the same time allowing all components of the VM to be freely changed at run time.

3.1.2 Execution Context Unavailability During Garbage Collection

A second problem arises in DMRs as a consequence of having a common paradigm for VM and application-level objects. Typical garbage collectors, even in self-hosted implementations like those of Squeak or PyPy, work within an

[3] By transitive closure of a set of objects we mean all objects that are directly or indirectly reachable from the objects.

execution environment that is split from the collected environment. Methods and classes of those garbage collectors cannot be reached through traversing roots of the collected space. Within the heap, the GC algorithm can freely change object pointers as needed to adjust references. But the GC of a DMR, on the other hand, is made of standard objects living in memory. These objects not only live in the same heap as application-level objects, but are indistinguishable from other kinds of objects. For example, application classes subclass the very same Object class from which the GarbageCollector class inherits. Native code is also represented using simple objects. Unless specially treated, code can be moved by the GC throughout memory as any other object. This means that at some point during GC different copies of the garbage collector code can be alive and reachable.

For the reasons described above, a DMR GC cannot pause all high-level execution, as its own code is written at high-level. DMR implementors have to face the problem that common garbage collection algorithms leave objects in an inconsistent state while the garbage collector is running. Object headers and pointers get mangled to detect live objects and to rearrange references, which means that at intermediate stages of collection objects are not able to receive messages. The garbage collector then needs to either be completely disconnected in some way from the collected execution environment, or be altered in a way that its objects are kept operative during its work. Two specific examples of this problem are described below: forwarding pointers in Copying Collectors (CC) and pointer reversal in Mark-and-Compact (M&C) [16]. Section 4.1 shows two different approaches that can be used either to solve those problems or to avoid them at all.

Forwarding Pointers in Copying Collectors. CCs need to mark reached objects and store a forwarding pointer to the new copies; this is done by overwriting the original header of the object with the forwarding pointer [10]. When garbage collection is finished, as no living object points to the original object, the value of its header does not matter. However, during the time where garbage collection is running many objects might have their header overwritten while there still were pointers to them. Any message sent to an object with a dirty header will cause undefined behavior. The virtual machine could be confused taking a forwarding pointer as an object type, size, or hash.

Pointer Reversal in Mark-and-Compact. Similarly, pointers of objects may be mangled with different purposes during marking phase. Tracing live objects requires the usage of a stack or queue to determine remaining objects to be scanned. The *pointer reversal* technique [28] can be used to eliminate the need of a special memory zone for the stack, using the slots of the objects to implement a stack like structure. In the M&C algorithm, *pointer threading* [17, 23] is done to efficiently find all references to each moved object.

3.1.3 Garbage Collection Debugging

As part of a metacircular system, it is desirable that the garbage collector can be debugged, at least partially, within the standard environment with standard tools. But this poses another chicken-and-egg situation: in metacircular environments, standard tools are implemented within the language, and require a fully functional user-interface handling events; but stop-the-world garbage collectors require pausing the high-level program execution, including the user-interface. Different solutions to this problem are given in section 4.1.2, trading off simplicity, dynamism and performance of the GC.

3.2 Performance

Dynamic programming environments usually pay for high-level facilities with performance. Eliminating the overhead of features provided by the environment requires implementing or reusing highly complex compilation toolchains [35]. Even when running with those toolchains, which are implemented on top of languages like C or C++, dynamic languages do not reach the same level of performance of code directly written in languages like C o C++.

A DMR implementation will face double performance pressure: it experiences the execution overhead associated with dynamic languages and it also adds the overhead of the language to the system side of the implementation. Systems like Jikes and Maxine reach high performance levels, but they are not based on dynamic languages. PyPy/Python and Cog/Squeak [22] are written in dynamic languages, but restricted to a language subset where types can be inferred and code can be translated to C. On the other hand, Klein never reached high performance levels and Tachyon only showed evidence on an incomplete system.

Without a compilation toolchain that is as complex as the ones of HotSpot, Jikes, V8, or PyPy, we still expect a DMR to be as fast as comparable systems. For Bee, comparable systems are the so-called *host* VM, a JIT-compiled VM for a Digitalk Smalltalk derivative, from which Bee inherits its dialect, and the CogVM, a well established Smalltalk VM with JIT compiler for Squeak and Pharo. The host VM is written in assembly and C, while the CogVM is written in Slang, a restricted subset of Smalltalk that is directly compiled to C code. Section 4.2 describes a set of optimization techniques that allow our DMR to reach the performance levels of the original VM and the CogVM.

In typical VMs the optimizations done by JIT compilers are applied using structures hidden to the application. In a DMR it is desirable to model optimizations within the language. This means that an optimized DMR for an OO language should use normal objects for instances of caches that improve performance, and that those objects should be freely accessible for inspection and analysis.

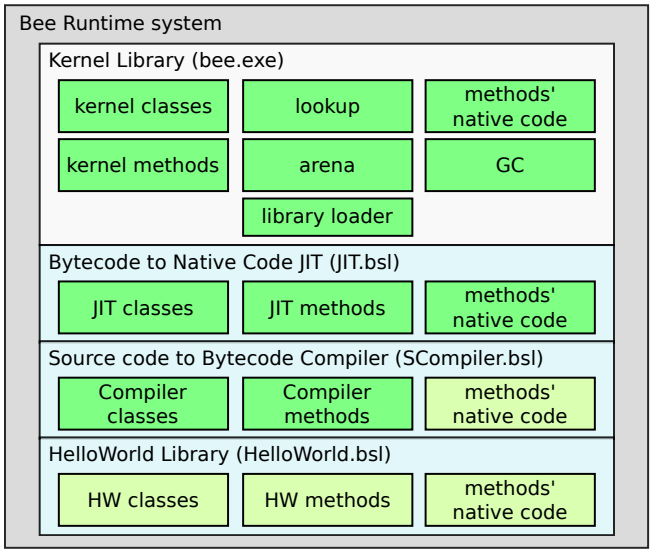

Figure 3. Bee modules. Kernel library contains minimal functionality to support itself and to load other libraries. Loading of JIT and Bytecode compilers is optional. Except for kernel and JIT, libraries can be shipped with just source code, with precompiled bytecodes or with native code.

4 The Design of the Bee GC and its Optimization Model

In this section we examine how Bee, a live programming environment, solves the problems identified in section 3.

The Bee runtime is completely written in Smalltalk, using the standard Smalltalk browsers, inspectors, and debugger. It is a self-hosted DMR per our definition. It supports GC, foreign function interface including callbacks, a graphical user interface, native and green threads[4], Smalltalk images, and support for updating arbitrary code at run time. All these features, as well as lower-level aspects such as method dispatch and primitive operations, are realized as Smalltalk methods. Code in Bee is AOT or JIT compiled from bytecodes but never interpreted. The Smalltalk stack lives on the native one, mixing managed object pointers with native return addresses and stack frame pointers. Its structure, as well as Bee object format and ABI are detailed in [25, Sect. 3].

Bootstrapping The Bee runtime is bootstrapped either from itself or by running the Smalltalk image on the host VM to generate an executable. This process packages together a Smalltalk *kernel* consisting mainly of classes and methods. Methods in this package are compiled and stored with their associated native code. During the creation of this package, things that refer to the host VM are removed. Methods that call privimitives, which refer to state and code of the host VM like memory spaces, lookup, GC and the library loader are replaced with Smalltalk-implemented complementary versions, where all state is stored in normal Smalltalk objects. The JIT is implemented as a separate Smalltalk library which can be loaded when needed. A high-level view of the ecosystem is shown in figure 3.

[4]Currently Bee allows only one thread to execute at a time.

4.1 Implementation of Garbage Collection

One of the toughest challenges when implementing Bee has been the garbage collector. As previously explained, the garbage collector needs to be able to traverse itself, to leave objects operational while running and to be debuggable with standard tools.

Two Possible Approaches. We see two possible approaches to achieve this goal. The first one is to use a standard collector and disconnect it from the garbage collected space. This avoids the problem of the GC collecting itself, i.e., traversing the data structures it uses, and it would avoid the problem of exposing inconsistent object state. The problem of this approach is that the GC cannot be debugged itself with the standard environment's tools, because once collection is started, it would freeze the UI. Thus, this approach would require the use of low-level tools for debugging.

A second approach is to modify existing GC algorithms to have objects in a consistent state throughout the complete garbage collection process. This means, it would restrict the GC design more than the first approach and might have performance implications, but allows us to use our standard high-level tools for debugging. In the following sections, we describe our implementation of both approaches.

4.1.1 A Garbage Collector in a Bottle

Using the first approach we replaced the garbage collectors of the host VM. The same algorithms were implemented, a mark-and-compact threading collector for the old space and a generational scavenging collector [6]. We statically generate a closure of the GC code and the objects required for it. This closure is put inside a dynamically-linked library (DLL) which can be regenerated on-the-fly and replaced as many times as needed at runtime. The closure is filled with a copy of all objects related to the garbage collector: classes, methods native code, etc. This makes the GC runtime independent of the original objects. The benefit is that important objects such as the Object class can be left inconsistent during GC, as the GC library uses its own copy of that same object. The collector uses a separate space for new objects it creates, which it discards after collection finishes.

Other GC algorithm variations were implemented afterwards tuning the internal structures of generational and mark-and-compact collectors, and even a multi-threaded variation of the mark-and-compact one was implemented.

For development, we were able to use our standard high-level tools to analyze and debug the algorithms. For example, we implemented a set of around one hundred GC tests with high-level tools, where external spaces with different objects

inside are dynamically generated. GCs can be debugged as they traverse these spaces, to check which objects are marked, collected and copied. However, for debugging the integration as DLL, we had to resort to low-level tools. Thus, the approach limits the useable tooling and isolates the GC implementation from the application, which we strive to avoid in our DMR.

4.1.2 A Collector that can Traverse Itself

To run the GC in the self-hosted environment with better debugging tools, we implemented the second approach and plugged it into the bootstrapped environment. To be able to have a live environment throughout the process of garbage collection, for the old space we switched from the mark-and-compact algorithm with object threading to a standard copying collector. This has an impact on memory usage, but the copying collector leaves objects in a operational state during collection. Specifically, we use an external forwarding table so that the GC modifies objects only to set the mark bit. This approach was used also to add generational GC support.

With this approach, it is possible to use the standard tools even while a GC is active. The main reason is that objects remain valid during copying, which makes it possible to open a debugger window and go step-by-step throughout the process. Using this debugger makes development easier than using a low-level debugger, as it allowed to understand the context of execution on the level of the Smalltalk implementation, which includes the ability to inspect object graphs with normal tools, a feature that is typically sorely missed when debugging GCs.

However, there remain limitations. During GC, objects start at *eden* and *from* spaces, and survivors are moved to *to* and *old* spaces. In the meantime, copied objects are alive in both places, until all references to objects in eden/from are forwarded to their copies. Running arbitrary Smalltalk, e.g. using the debugger, may mutate objects in eden/from spaces that have already been copied to their new location. These changes will be lost when GC ends and might corrupt the system state. Thus, debugging the GC requires great care. Currently, we have not extended the tools to be aware of GC and ensure that both objects are modified. In practice we found that debugging complex issues over an extended amount of time can lead to inconsistencies in the system, typically causing the user interface to break. Such issues can currently only be recovered from by restarting the Bee runtime.

Experience obtained from this made us implement an external debugger, based on the operating system API, that let us debug our Smalltalk remotely, and safely navigate through its memory. This debugger presents both a low and high-level vision of objects in memory and prohibits their modification while inspecting them in the debugged environment. In practice both debuggers are needed: the typical Smalltalk one for quickly finding simple bugs, the low-level one for detecting more subtle ones.

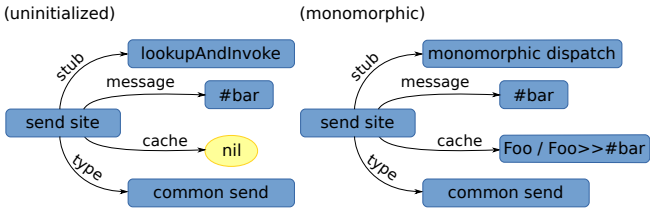

Figure 4. A send site modeling #bar being called on a variable. Initially it is empty so its cache is nil and its stub points to the lookup routine. When send is executed the send site is modified by the stub to become a monomorphic send and save a cache, which is a pair of class and compiled method.

4.2 Modeling Optimizations in DMRs

One goal of DMRs is to represent runtime data structures as plain objects, where possible. This section discusses our experience of applying this idea to a set of standard and ad-hoc optimizations that will let DMRs perform as efficiently as a classic VM with a baseline JIT compiler. While efficiency can be further improved through the implementation of more complex compilers, we focus on the feasibility of approach, showing that optimizations can be modeled within the language, and also that if needed the semantic of the language can be altered at specific points to make the DMR run faster.

4.2.1 Message Dispatch Model

In order to create a design that models optimizations of message dispatch, Bee implements objects of type SendSite. Each message in a compiled method is dispatched through an indirect call to a send site. A send site points to a code stub to be executed when activated, the message name, its type and a cache. This way, it can be configured with different policies dynamically. Initially it is set to perform a lookup. When executed, the stub changes itself to behave as a monomorphic send site, and on failure it will be transformed dynamically to a polymorphic send site. The structure of SendSite objects is shown in figure 4. This approach is a realization of classic polymorphic inline caches [12]. We discuss our experience with it in section 5.1.1.

4.2.2 Hand Tuning of Compiler Optimizations

Bee's compiler supports inlining, does simple optimization passes, and performs register allocation. It first transforms code from AST to an SSA-based intermediate representation, where it then performs standard optimizations such as value numbering or redundant load and dead-store elimination.

However, Bee does not yet support speculative optimizations. Instead, methods are manually selected for optimization. So far, we focused on performance critical methods that are typically considered *primitives*, and would normally

be implemented in C/C++. For this code, our compiler is configured to convert specific dynamic calls to static dispatches, where it is assumed that method lookup will not fail. Given a send site, the compiler either configures it to statically invoke a method, or either removes it and inlines the method's code. For both cases, the compilation environment is setup with a dictionary which maps message names to compiled methods.

5 Benefits and Drawbacks of DMRs

In this section, we report on our experience with Bee as a DMR and the benefits as well as drawbacks we see for such a system. We discuss a set of proof-of-concept applications of the concepts and experiments with our implementation.

5.1 Dispatch Analysis and Experimentation

Currently, Bee supports only basic compiler optimizations and thus, does not yet reach the performance of state-of-the-art JS Virtual Machines. One reason is that it does not yet do speculative optimizations. This makes the dispatch overhead in the system a critical component for overall performance. Since all code is implemented in Smalltalk, the impact of dispatch overhead is exacerbated compared to classic VMs were part of the implementation is in precompiled C/C++ code. When analyzing the performance of our system using only basic global lookup caches (GLC) [8, 32] and monomorphic inline caches [9], we found that about 85% of run time was spent in message lookup. This was excessive and our goal was to understand the reason and to find a solution to this.

As a consequence we explored how to use the DMR to improve performance and finally implemented the aforementioned polymorphic inline caches (cf. section 4.2.1) and method-specific optimizations (cf. section 4.2.2).

5.1.1 Polymorphic Inline Caches

As a first experiment, we changed the SendSite implementation to count the amount of times each message name was dispatched. The goal was to gain insight into the message send behavior at send sites to determine the cause of the high overhead.

After running the benchmarks and then using the standard object inspectors to examine the data for each SendSite, we were able to determine that the main problem was an unusually high degree of polymorphic send sites in our system.

With the object structure for SendSites already in place, it was straightforward to add support for polymorphism inline caches (PICs), too. Now, when a send site in a monomorphic state fails the check for the receiver type, it switches its own stub into polymorphic state and changes its cache into an array of pairs for receiver type and the looked up target method. In case the array is filled with entries, the SendSite will revert to the megamorphic case and always perform a lookup. This reduced the execution time of our benchmarks by a factor of 4. However, more importantly, it did not require any ventures into low-level code or tools. Instead, the whole analysis and the changes were done with the standard Smalltalk tooling.

5.1.2 Altering Message-Dispatch Semantics

While PICs improved performance significantly, we further analyzed how to optimize the system. With the PICs implemented, we wanted to understand what their content is. We did another experiment to analyze the SendSite caches. Specifically, we changed polymorphic send sites to register that switched to a megamorphic state, storing the contents of their caches in a dynamically growing collection. This allowed use to analyze in detail why send sites become megamorphic. The results showed that that methods that implement basic DMR functionality to replace primitives have a tendency to be megamorphic and overflow PICs. For example, consider the access of an object field with the #at: method.

```
Object>>#at: i
  ^self _isBytes
    ifTrue: [self basicByteAt: i]
    ifFalse: [self basicObjectAt: i]
```

Being implemented at the root of the class hierarchy, the receiver of at: is often of various different types. This means, the SendSite caches for the dispatch of #basicByteAt: and #basicObjectAt: are going to be megamorphic, likely seeing all subclasses of Object, and being orders of magnitude slower. Lower-level VMs do not suffer from this problem as the primitives are written in languages that do not do dynamic dispatch.

We solved the issue by creating another kind of send site, which completely remove method lookup in specific methods. Thanks to the design explained in section 4.2.2, it was possible to make the compiler prefill send site caches with specific compiled methods for specific methods. These stubs do not check the class of the receiver but invoke the cached method directly, effectively making the send static.

As with the previous optimization, analysis, implementation, and debugging of this performance issue were done entirely using the standard tooling without need to revert to low-level tooling that is foreign to application developers.

5.1.3 Discussion

To conclude, with the DMR approach, we were able to dynamically analyze, modify and interact with dispatch mechanisms within the environment. In static VMs, the same is not possible and the caches are neither plain objects that can be easily queried, inspected, and visualized, nor directly accessible without special tooling and potentially creating a custom build of the VM.

Furthermore, the SendSite model has shown to be adaptable and since the native code compiler is dynamic and available at application level, it is possible to create ad-hoc optimizations to the system to obtain better performance. Those optimizations can also be added by application programmers without the requirement of upstream support from VM developers. They can be tested and their performance impact can be measured while the system is running.

5.2 Memory Usage Analysis

For application and VM development alike, it is sometimes necessary to determine the cause of unusual memory usage, that might be caused by inefficient data representations or memory leaks.

During the development of Bee, we frequently used the capability to quickly explore the complete heap structure, including the VM's data structures. For such use cases it is beneficial that DMRs pose less barriers to obtain fine-grained information about the contents of memory and lifetime of objects than static VMs. The reason is that the latter only provide interfaces to predefined aspects of the VM. Asking a question that has not been anticipated by VM designers can require VM modifications, rebuilding and restarting the system. In a DMR, VM modifications are no different than normal code modifications, and do not require rebuilding nor restarting. Hence, arbitrary questions can be answered quickly and efficiently, by writing and executing few lines of code. For example, in Bee, it takes only one method to create a histogram of the types of allocated objects in memory, and it does not require any anticipated support from the VM. Similarly, we can create an object size histogram or map the objects that use most space.

Thanks to having a unified programming model, results of different analysis can be connected directly to high-level tools for better visualization and understanding.

5.3 Ideal Low-Level Laboratory

The ecosystem devised to develop, build and debug Bee led to a compelling feedback loop in the low-level area. Tools were built on top of the ones made previously, taking advantage of the synergy created by using a single programming model. Compilers, optimizers and disassemblers, being written in high-level, were combined with dynamic tools like inspectors and visualizers, giving birth to a full range of low-level tools: a native code debugger for generic executables, which was extended to become a specialized remote VM debugger for Bee runtime; a graph visualizer for the optimizing compiler; and a native code profiler, among others. The remote VM debugger is not only able to show and modify all the low-level state of a VM like registers, memory and stack frames, as in Maxine inspector [21], or Jikes RDB [19], but it is also capable of showing a high-level view of the objects, and source code being executed as Smalltalk methods. We also were able to implement a customized experimental back-in-time debugger for native code, specially useful for fixing bugs in the garbage collector. Furthermore, the debugger itself can be modified while the debugged application is running, allowing for highly interactive debugging sessions.

5.4 Limits of the Implementation

Our goal is to allow changing all the runtime dynamically. In this respect, it is important to understand what is possible and what is not yet possible in our implementation:

Lookup The implementation of lookup can be freely changed at run time. Besides saving the methods involved, it requires an additional installation step to plug in the native code and flush the inline caches where needed. This step is instantaneous.

Methods Native code of methods is generated lazily. This could mean that changing a kernel method to replace a previous one leaves the kernel without native code required to continue executing. To avoid this, the method installer JIT-compiles methods when it detects a previous version with native code present.

Object Header Format We chose implementation simplicity over dynamism in the design of the API that offers access to object headers. Unlike stratified metaobject protocol implementations, Bee does not offer any functionality to change object header format at run time nor to switch from immediate to boxed integers.

Maturity In its current state, Bee is able to run any code that was written to run on top of the original VM. This includes not only small code snippets but things like the whole IDE (with browsers, inspectors and debuggers), and even parts of the simulations implemented in the products of the company. However, the implementation is not yet stable, and currently we are at a first step of making self-hosted Bee be used not as a production environment but as a development one.

5.5 Applications

The Bee project was started by a software company which markets a mature and sophisticated simulation environment for more than two decades. The development team of said company wanted tighter control of the development stack to improve the quality of their software. The company's experience shows that on-the-fly changes aid in incremental improvement of the system. While this may not be a common perspective, in their Smalltalk system, they continuously improve kernel classes on a daily basis. They found it so useful, that they wanted to extend it to the VM itself. From their point of view, the distinction between application and system programming can be minimized.

This work can also be beneficial in other areas like research and education. From our perspective, user lack of knowledge in the area of systems programming is not the reason they avoid modifying a system. Instead, from our

experience, it is more often the case that the lack of familiar tools and the inaccessibility of the system's code results in users avoiding to modify the system at hand. Within a DMR, programmers can benefit from a unified abstraction level and use of the same language for all tasks. Both VM and application developers are provided a unified model and the same set of tools of the environment.

5.6 Disadvantages, Drawbacks, and Tradeoffs

While presenting many interesting advantages, there are also tradeoffs that come with a DMR.

System Instability During Runtime Development Changing core components of the runtime while it is running can easily cause the system and the IDE to crash. However, this is not unlike modifying core classes of classic Smalltalk systems. For instance, changing Class, Object, or Collection in a Smalltalk system can lead to crashes and undefined behavior, because the expectations of the underlying runtime system are not met anymore. To aid in this problem during the development of the DMR, we expanded our support for remote debugging, which retrofitted as a benefit for the whole ecosystem, as it also makes it easier to change core classes of the Smalltalk system.

Restricted Access to Libraries Despite the benefits of programming in a higher-level language with more flexible tools, there can appear a subtle problem. When writing a VM in a language like C++, all C++ libraries are available to use within the VM code. However, coding kernel features in the DMR is prone to introduce circular dependencies, and the developer has to be more careful when writing code. There is a limited amount of features that can be accessed at some points. Nonetheless, this has not been a big issue during the development of the project.

Always Present Native Code In a DMR the code cache can never be flushed completely. At least the parts of the kernel that are in use must always be present. This can make some system components more difficult to implement than in traditional systems. For example, the garbage collector cannot just throw away the native code cache [9] away, it has to be smart enough to traverse native code and update all references in it. The same happens when updating a method in a class, it requires selectively flushing inline caches of all methods that use the selector of that method, instead of just throwing all code cache away.

6 Quantitative Evaluation

To provide an impression of quantitative aspects of Bee, we briefly discuss performance as well as the size of the system.

6.1 Performance Evaluation

The goal of this evaluation is to show that Bee, as a DMR, can reach the performance of the original non-DMR implementation. Since the host VM is not generally available, we also include the CogVM running Pharo 5 as an open source VM and Smalltalk platform. For Pharo, we include results for the VM when using a JIT compiler. We also include results for V8 JavaScript engine as in Node.js v8.1.4. All Smalltalk implementations are 32-bit while V8 is 64 bits. The benchmarks were run on a machine with a 2.8Ghz 4-core Core i7 7700HQ with hyperthreading and 16GB of memory. The operating system is a 64-bit Ubuntu 17.04. Measures were taken collecting 50 iterations for each benchmark.

For the performance comparison, we consider peak performance only and discount start-up, warm-up, and JIT-compilation times. When running on the host VM, all methods are lazily nativized before execution. To reduce JIT-compilation measurement noise during benchmarking, we run a warm-up phase of one iteration before timing for the host VM and Pharo. In Bee, the benchmarks are AOT compiled and run without even loading the JIT-compiler.

The benchmarks are run with an initial heap size of 64 MB, to minimize noise introduced by the GC. The results are normalized to Pharo to use it as the baseline for the performance comparison. We report averages and confidence intervals with $\alpha = 95\%$.

To measure performance, we use the Are We Fast Yet benchmarks ranging from micro to macrobenchmarks [20]. They are designed to compare performance across different language implementations and thus were easy to adapt to Bee's Smalltalk dialect. DeltaBlue and Richards are classic benchmarks evaluating the performance of object-oriented applications. Havlak is an optimization algorithm for a compiler but is representative for many application-level optimization problems, too. And the Json benchmark parses a larger JSON document, which is relevant for the performance of many REST services used in todays web applications or micro services. The rest is a collection of numerical and OO benchmarks stressing particular aspects of the implementation.

Figure 5 shows the results. For the macrobenchmarks, Bee DMR consistently surpasses the performance of the host VM and is close to the one of Pharo 5, matching it in Richards and Havlak. The results in Json are expected as Bee has not been optimized for string operations yet. For example, string copying is done byte-by-byte, checking bounds at each character. Results of the majority of microbenchmarks follow the same trend than macrobenchmarks, with the exception of cases that stress areas that have not been optimized yet in Bee: object allocation, immediate floats and the GC. Specifically, Mandelbrot, NBody and Storage stress allocation in Bee. Mandelbrot and NBody have higher allocation rates

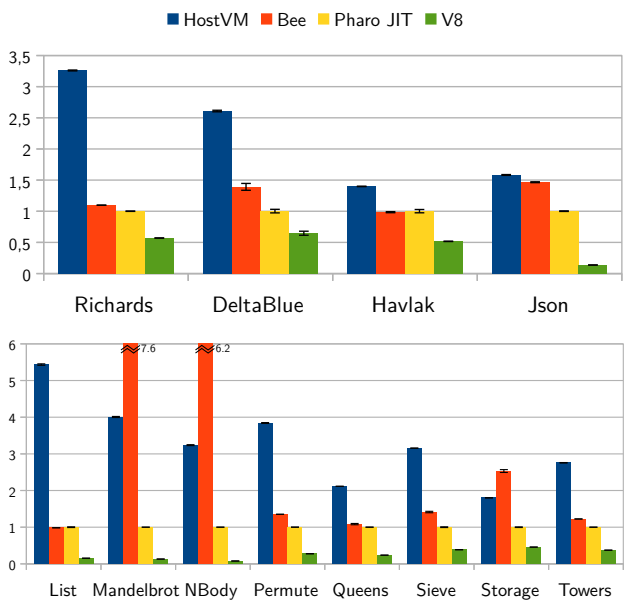

Figure 5. Normalized high- and medium-level benchmarks execution times, relative to Pharo 5 (lower is better).

on Bee than in other systems, since Bee uses a boxed representation for double values. Regarding to V8, a VM that implements an adaptive compiler

Overall results look promising, specially considering that there still remain many well known optimizations that can be implemented in the near future. Particularly relevant shall be those related to dynamic compilation such as adaptive recompilation [13], escape analysis [24, 30], and also others related to compilers in general, like peephole optimization or loop invariant code motion to mention a few.

6.2 Implementation Size

To evaluate the size of our runtime implementation we report metrics for its code base. The Bee kernel runtime is small: 4122 lines of code, within 962 methods that are added to the already existing Bee kernel library, which is used when running on top of the host VM and consists of 5483 methods and 20774 lines of code. Those lines added include the implementation of lookup, primitives, and GC. 56% of those methods are one liners. Only 72 methods take more than 10 lines, with a maximum size of 31 lines. The JIT compiler adds 14099 lines of code, distributed in 3105 methods.

7 Conclusions and Future Work

In this paper, we report on our experience implementing a dynamic metacircular runtime (DMR). We detail Bee, a dynamic Smalltalk runtime completely implemented in Smalltalk. The focus is on identifying the problems that need to be solved when building a DMR, as well as discussing our solution approaches and their benefits as well as drawbacks. Specifically, we focused on garbage collection support and optimizations. We detail the tradeoffs between a GC that is disconnected from the runtime system, and one that is completely integrated and allows us to utilize the standard Smalltalk tooling for implementation, testing, and debugging. Similarly, we discuss how standard optimizations can be applied to a DMR by detailing how polymorphic inline caches can be built with standard objects, which consequently can be implemented and analyzed like normal application code. Finally, we have demonstrated based on benchmarks that a DMR can be as fast as other widely used static VMs for Smalltalk.

Overall, we believe this experience report shows that it is feasible to build completely dynamic VMs, which avoid classic static VM components and instead make the VM part of an application, which lowers the barrier to VM understanding, development and evolution. We hope that this approach will lead to more productive VM development in the future and allow for application-specific customizations as it is already the case for common standard libraries for a wide range of dynamic languages.

Future Work In future work, it remains to be explored what other new functionalities can be implemented by taking advantage of the dynamicity and standard tooling for VM development. It also remains to be verified if this methodology can be applied to other dynamic languages like Python, Ruby or JavaScript. From the description of the problems presented throughout this paper, we strongly believe the implementation of runtimes similar to Bee in those languages is feasible.

While we were able to reach the performance of VMs with baseline compilers, more performance work remains to be done. One important question is how the complexity of adaptive compilation affects the maintainability of DMRs.

Finally, we think that there is space for research on implementing new languages inside the environment. This could include low-level language extensions, support for different hardware architectures such as GPGPUs, as well as higher-level abstractions for distributed computing.

Acknowledgments

The authors want to thank Gerardo Richarte, Valeria Murgia, Leandro Caniglia, Jan Vrany and the development team of Palantir Solutions for providing valuable ideas, discussions and reviews, and being in charge of the development and maintenance of all Bee libraries. This work was funded by Palantir Solutions. Stefan Marr was funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31.

References

[1] 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. 2000. The Jalapeño virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238.

[2] B. Alpern, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K.S. McKinley, M. Mergen, J. E B Moss, T. Ngo, V. Sarkar, and M. Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–417.

[3] Stephen M Blackburn, Sergey I Salishev, Mikhail Danilov, Oleg A Mokhovikov, Anton A Nashatyrev, Peter A Novodvorsky, Vadim I Bogdanov, Xiao Feng Li, and Dennis Ushakov. 2007. The Moxie JVM experience. *cluster computing* (2007).

[4] Guido Chari, Diego Garbervetsky, and Stefan Marr. 2016. Building Efficient and Highly Run-time Adaptable Virtual Machines. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS'16)*. ACM, 60–71.

[5] Guido Chari, Diego Garbervetsky, Stefan Marr, and Stéphane Ducasse. 2015. Towards fully reflective environments. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, 240–253.

[6] Chris J Cheney. 1970. A nonrecursive list compacting algorithm. *Commun. ACM* 13, 11 (1970), 677–678.

[7] Maxime Chevalier-Boisvert, Erick Lavoie, Marc Feeley, and Bruno Dufour. 2011. Bootstrapping a Self-hosted Research Virtual Machine for JavaScript: An Experience Report. In *Proceedings of the 7th Symposium on Dynamic Languages (DLS '11)*. ACM, 61–72.

[8] Thomas J Conroy and Eduardo Pelegri-Llopart. 1983. An assessment of method-lookup caches for Smalltalk-80 implementations. *Kra83* (1983).

[9] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, 297–302.

[10] Robert R Fenichel and Jerome C Yochelson. 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM* 12, 11 (1969), 611–612.

[11] Daniel Frampton, Stephen M Blackburn, Perry Cheng, Robin J Garner, David Grove, J Eliot B Moss, and Sergey I Salishev. 2009. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 81–90.

[12] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. Springer-Verlag, 21–38. http://dl.acm.org/citation.cfm?id=646149.679193

[13] Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, 326–336.

[14] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2015. A domain-specific language for building self-optimizing AST interpreters. *ACM SIGPLAN Notices* 50, 3 (2015), 123–132.

[15] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*. ACM, 318–326.

[16] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC.

[17] HBM Jonkers. 1979. A fast garbage compaction algorithm. *Inform. Process. Lett.* 9, 1 (1979), 26–30.

[18] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. 1991. *The art of the metaobject protocol*. MIT.

[19] Dmitri Makarov and Matthias Hauswirth. 2013. Jikes RDB: a debugger for the Jikes RVM. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13)*. ACM, 169–172.

[20] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS'16)*. ACM, 120–131.

[21] Bernd Mathiske. 2008. The Maxine Virtual Machine and Inspector. In *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA Companion '08)*. ACM, 739–740.

[22] Eliot Miranda. 2011. The cog smalltalk virtual machine. In *VMIL'11: Proceedings of the 5th workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*.

[23] F Lockwood Morris. 1978. A time-and space-efficient garbage compaction algorithm. *Commun. ACM* 21, 8 (1978), 662–665.

[24] Young Gil Park and Benjamin Goldberg. 1992. Escape analysis on lists. In *ACM SIGPLAN Notices*, Vol. 27. ACM, 116–127.

[25] Javier Pimás, Javier Burroni, and Gerardo Richarte. 2014. Design and implementation of Bee Smalltalk runtime. (2014).

[26] Barry Redmond and Vinny Cahill. 2000. Iguana/J: Towards a dynamic and efficient reflective architecture for Java. In *ECOOP 2000 Workshop on Reflection and Metalevel Architectures*.

[27] Armin Rigo and Samuele Pedroni. 2006. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, 944–953.

[28] Herbert Schorr and William M Waite. 1967. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM* 10, 8 (1967), 501–506.

[29] Brian Cantwell Smith. 1984. Reflection and semantics in Lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 23–35.

[30] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial escape analysis and scalar replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 165.

[31] David Ungar, Adam Spitz, and Alex Ausch. 2005. Constructing a metacircular Virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 11–20.

[32] David M Ungar. 1983. Berkeley Smalltalk: Who knows where the time goes? *Smalltalk-80: bits of history, words of advice* (1983), 189–206.

[33] Toon Verwaest, Camillo Bruni, David Gurtner, Adrian Lienhard, and Oscar Niestrasz. 2010. Pinocchio: bringing reflection to life with first-class interpreters. *ACM Sigplan Notices* 45, 10 (2010), 774–789.

[34] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (Jan. 2013), 24 pages.

[35] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 187–204.

[36] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 73–82.