Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Live objects all the way down: removing barriers between applications and virtual machines

Tesis presentada para optar por el título de Doctor de la Universidad de Buenos Aires en el área de Ciencias de la Computación

Javier Esteban Pimás

Director de tesis:	Diego Garbervetsky
Consejero de estudios:	Rodrigo Castro
Lugar de trabajo:	Departamento de Computación, FCEyN, UBA

Buenos Aires, 2024

Live objects all the way down: removing barriers between applications and virtual machines

Abstract Object-oriented languages often use virtual machines (VMs) that provide mechanisms such as just-in-time (JIT) compilation and garbage collection (GC). These VM components are typically implemented in a separate layer, isolating them from the application. While this approach brings the software engineering benefits of clear separation and decoupling, it introduces barriers for both understanding VM behavior and evolving the VM implementation because it weakens the causal connections between applications and VM. For example, the GC and JIT compiler are typically fixed at VM build time, limiting arbitrary adaptation at run time. Furthermore, because of this separation, the implementation of the VM cannot typically be inspected and debugged in the same way as application code, enshrining a distinction in easy-to-work-with application and hard-to-work-with VM code.

These characteristics pose a barrier for application developers to understand the engine on top of which their own code runs, and fosters a knowledge gap that prevents application developers to change the VM.

We propose Live Metacircular Runtimes (LMRs) to overcome this problem. LMRs are language runtime systems that seamlessly integrate the VM into the application in live programming environments. Unlike classic metacircular approaches, we propose to completely remove the separation between application and VM. By systematically applying object-oriented design to VM components, we can build live runtime systems that are small and flexible enough, where VM engineers can benefit of live programming features such as short feedback loops, and application developers with fewer VM expertise can benefit of the stronger causal connections between their programs and the VM implementation.

To evaluate our proposal, we implemented Bee/LMR, a live VM for a Smalltalk-derivative environment in 22,057 lines of code. We analyze case studies on tuning the garbage collector, avoiding recompilations by the just-in-time compiler, and adding support to optimize code with vector instructions to demonstrate the trade-offs of extending exploratory programming to VM development in the context

of an industrial application used in production. Based on the case studies, we illustrate how our approach facilitates the daily development work of a small team of application developers.

Our approach enables VM developers to gain access to live programming tools traditionally reserved for application developers, while application developers can interact with the VM and modify it using the high-level tools they use every day. Both application and VM developers can seamlessly inspect, debug, understand, and modify the different parts of the VM with shorter feedback loops and higher-level tools.

Keywords virtual machines, compilers, garbage collection, metaprogramming, live programming, object-oriented programming

Contents

Chapter 1

Introduction

Over the last decades, more and more types of applications moved from lower-level languages to higher-level ones. Language implementations enabled *Live Programming Environments (LPEs)*, allowing us to modify programs as they run. This feedback-driven exploratory programming style [RRL+18] lets application developers experiment with domain problems in short feedback loops with an immediate response to their actions.

Unfortunately, this exploratory programming style is restricted to application code, leaving the underlying language implementation unreachable and virtually immutable for application developers. This is because LPEs are implemented using virtual machines (VMs).

This intentional architectural separation deals with portability and limits interdependencies. The interfaces VMs provide are designed to hide implementation details and limit the ways an application can interact with a VM and customize it to its needs, in effect making it *invisible* to the application programmer. The VM only provides an opaque interface through which the guest language can communicate in a well-defined but strictly limited way.

However, this strict separation is a double-edged sword. While it provides benefits, it intentionally keeps application developers unaware of design and implementation details, which prevents them from fully understanding performance pitfalls and restrictions of the system. This can lead to inefficient application code, perhaps causing excesive garbage collections, or inadvertently triggering

unnecessary recompilations.

The strict separation also prevents developers from improving the VM and harnessing it in their application, e.g. by leveraging the just-in-time compiler or the graph tracing facilities of the garbage collector.

As a result, the architecture of state-of-the-art VMs makes it hard for application developers to develop the most efficient code possible.

While numerous projects explored modern ways of implementing VMs [AAB+00, WHVDV+13, BSD+08, WWW+13a, RP06, CBLFD11, IKM+97], none designed the VMs in a way so that the code of the VM itself can be changed while it is executing, because VM components such as the just-in-time compiler and garbage collector are separated from the hosted language. Thus, the exploratory programming possible in LPEs is not available to VM engineers, who have to endure longer feedback loops.

Of course understanding a VM and its essential complexity is also a hurdle. However, for application developers much of the overall complexity is merely accidental and stems from the programming environment in which the VM is implemented.

The differences between the languages, tools, and basic development processes available for VM and application development significantly increase the learning curve.

While some may argue that writing a just-in-time compiler, garbage collector, or VM built-in functions may require special skills, we argue that giving developers the same programming environment, tools, and development process that they are used to in their applications is far more important to enable them to understand, and even contribute to a language implementation.

At least to a degree, this is similar to the practice of applications to *vendor*, i.e., to integrate, the code of frameworks they rely on into the application codebase. This is usually done to have more control over critical dependencies, but also to be able to better understand how one's application interacts with a framework.

In this thesis, we demonstrate that it is possible to design a VM so that it can be changed and worked on with the same tools known to application developers by avoiding the architectural separation. We argue for a classic object-oriented design that enables live object-oriented programming, enabling us to construct a

Live Metacircular Runtime (LMR). Based on the following case studies that are taken from the development of a large industrial application with 1.1 million lines of code, we show that application-level problems can be solved using solution strategies that remain in that level, showing the benefit of less opaque and more flexible VMs. The case studies are:

GCT - Garbage Collector Tuning for a particular use case.

JITC - Changing a JIT compiler to avoid recompiling methods too frequently.

CompO - Optimizing the application with vector instructions added to the compiler.

Our case studies were conducted with Bee Smalltalk [PBR14, PBAM17], which we use to implement the Bee Live Metacircular Runtime. Bee/LMR is a self-hosted VM runtime, written in Smalltalk. It replaced a previous Bee VM, known as Bee/SVM, a traditional VM design which was written in C++ and assembly. Bee/LMR allows application developers to modify the virtual machine code at run time, as they would do with any other application-level code within the programming environment. It has been deployed to clients and is used daily for development of a simulation product in the oil and gas industry. While our focus for Bee/LMR was practicality and stability, section 8.1 shows that the performance is not too far off a widely used Smalltalk VM and faster than Ruby and Python.

1.1 Problem Statement

Problem Statement. Split-layer VM design. *Current LPEs are built on top of VMs that use a two-layer design. This design presents opaque VM application programming interfaces that reduce the opportunities of understanding and changing both VM code and application runtime behavior. Furthermore, this design forces long feedback loops in VM development, pose unnecessary learning leaps to application developers when debugging VM behavior and make changes to VMs harder to implement and deploy than changes to applications. It is usually unpractical for application programmers to change the VM code after deployment and even during development.*

1.2 Thesis Statement

We state our thesis as follows:

Hypothesis. *A unified design, which places the execution environment and the applications at the same level, and which applies the same standard encapsulation tools for abstraction and complexity hiding, allows for immediate, incremental and practical observation and modification of the VM. This creates opportunities for dealing with application development problems by leveraging virtual machine components.*

To demonstrate our thesis we have applied live-programming techniques to VM-level code, giving form to Live Metacircular Runtimes (LMRs).

1.3 Contribution

The key contributions of this work are:

- The design of a runtime system without the architectural separation of VM and application, which we call Live Metacircular Runtimes.
- The implementation of a self-hosted Smalltalk with a just-in-time compiler and garbage collector, where every part of the runtime can be changed at run time, and a framework that allows to lively debug the system from the outside when low-level errors occur.
- An evaluation of the trade-offs of removing the architectural distinction between VM and application using three case studies on a large industrial application. The key benefits are the gains of immediacy of LPEs and the explorability and malleability of LMRs.

We conduct our experiments through the implementation of Bee/LMR, a self-hosted Smalltalk runtime library. Bee/LMR allows modifying the code of its components at execution time, as any other application-level code within the programming environment.

Bee is a live OO system with the following characteristics:

i) it is dynamically typed, as defined in the next section,

ii) it uses garbage collection,

iii) it facilitates inspecting, debugging and changing code arbitrarily at run time.

Bee/LMR is a runtime implementation for Bee that allows to live-program the runtime system itself. Similar projects in the past provided support for dynamic changes of a runtime [USA05, VBG+10] and were an inspiration to this work. However, none of them were able to create a completely working system. They did not support all the live programming features of the language being implemented, nevertheless could run as fast as traditional VMs. Specifically, they did not implement lively programmable garbage collectors, and their performance was poor.

Until today, runtimes for live programming environments supported only few or no live changes, even when written using dynamic languages. We address that situation in this work.

1.3.1 Published Work

This thesis focuses on Live Metacircular Runtimes. Some results presented here have been originally published in [PBAM17, PM17, PC19, PMG24]. During the PhD I worked on other topics, always related to virtual machines but not necessarily to LMRs. Details of that research, such as [Pim22], [Pim18] and [CPVF18] were not included to keep the thesis concise and structured.

1.4 Thesis Outline

The rest of this dissertation is structured as described next.

Chapter 2 Details the context under which this research has been carried out, describing live programming environments, virtual machines and Bee Smalltalk.

Chapter 3 Shows the problems faced by traditional application developers when using virtual machines designed as a split layer. We show 3 real-life case studies that exemplify those problems.

Chapter 4 Proposes our new design, which we call Live Metacircular Runtimes, and details its implementation. This includes information about the architecture of Bee/LMR, its JIT compilers, built-in functions and related work.

Chapter 5 Describes the challenges of implementing garbage collection and memory managment in LMRs, and shows the design and implementation of different algorithms using Bee/LMR as a research vehicle.

Chapter 6 Presents the Metaphysics framework, which is used to implement an out-of-process debugger for Bee/LMR, which was designed with the purpose of debugging low-level components such as the GC and JIT.

Chapter 7 Includes a qualitative evaluation were we show how LMRs improve upon state-of-the-art VMs, how they solve the problems stated in the case studies presented in section 3.2, and address concerns associated with LMRs.

Chapter 8 Evaluates LMRs quantitatively, by doing a performance comparison with other VMs, and also comparing implementation size.

Chapter 9 Presents conclusions and future work.

Chapter 2

Background

Live Programming and The Architecture of Virtual Machines

The focus of this thesis is shortening feedback loops in VM development. In this chapter we provide a background on the related concepts and, to avoid confusion, we also define their specific meaning when referenced throughout this text.

2.1 Dynamic Languages

Definition 2.1.1 *We say* Dynamic Language *to refer to languages that are designed to reduce software complexity by hiding details from programmers in two ways: they are dynamically typed, and they provide automatic memory management.*

Smalltalk, Self, JavaScript, Python, Ruby are typical examples of dynamic languages.

2.2 Live Programming Environments

Definition 2.2.1 *A* Live Programming Environment (LPE) *is a set of software tools that allow to continuously develop programs at the same time they are running, allowing to change their components without restarting them so that the development feedback loop is immediate.*

LPEs are designed to allow programmers recompile code with instant feedback loops, without requiring a separate compilation step before execution. Such environments have a long history [RRL+18], with the work on SOAR being perhaps one of the earliest examples [UBF+84] of Live Programming Environments for object-oriented languages based on Smalltalk. Self was also designed with that same philosophy [CUL89] and its Klein [USA05] implementation comes the closest to what we consider a Live Metacircular Runtime, as we discuss in Chapter 4 below.

It is not enough for a language to be dynamic and to provide reflective facilities to be a live programming environment. It also needs the tools to dynamically inspect the program state, debug the program code and lively change it. Smalltalk is the seminal example of Live Programming Environments for Object-Oriented languages. Self was also designed with that same philosophy.

Other languages such as Python, JavaScript, and Ruby were not originally implemented as LPEs, in practice requiring program restarts to evaluate new code. However, with time they have gained features and tools that shorten feedback loops and allow for exploratory programming. For example, IPython [PG07] gives developers a *computational notebooks*, which is similar to a read-eval-print with mostly immediate feedback. Since it became popular, it got renamed to Jupyter to indicate the support of a wide range of languages and is widely used for instance for data analysis tasks.

2.3 Virtual Machines

Dynamic object-oriented programming languages run of top of Virtual Machines.

Definition 2.3.1 *A Process Virtual Machine, or just* Virtual Machine*, is a piece of software that takes high-level guest programs as input and executes them on the underlying hosting hardware.*

Virtual Machines are made of different components that are put together to obtain a working *runtime system*. These components usually contain parsers, compilers, memory managers, built-in operations and interfaces for calling in and out from the guest language to the host and vice versa.

There are multiple techniques for executing the source code of programs in

Virtual Machines. Interpreters, JIT- and AOT-compilers trade efficiency for implementation ease, among others. Usually, the source code of programs is translated into an Intermediate Representation (IR), which can be more efficiently executed either through an interpreter or through compilation to native code. To maximize performance a VM can mix interpretation with JIT and AOT compilation.

Definition 2.3.2 (Interpreter) *An* interpreter *is a program that executes another target program without translating the target to native code.*

Definition 2.3.3 *A* JIT compiler *is a program that transforms another target program into native code dynamically, while the target is executing*

Definition 2.3.4 *An* AOT compiler *is a program that transforms another target program into native code before the target starts execution.*

JIT compilers were made popular by [DS84] in conjunction with other techniques to make Smalltalk systems more efficient. In JIT compilers, the nativization process can be triggered on demand, usually after the VM has detected a code section has been executed multiple times. JIT compilers trade compilation time for execution time, so they are designed in a tiered fashion: when the VM detects a piece of code is being executed with certain frequency, it invokes a fast JIT compiler that performs cheap optimizations; when it detects the piece of code is being executed even more frequently, it can invoke a JIT compiler that performs even more optimizations to the code, spending more compilation budget to produce a smaller overall program execution time.

2.3.1 The Architecture of Virtual Machines

This section gives a brief overview of the architectures provided by virtual machines. First, we discuss the classic layered architecture where the VM enforces a separation. Then, we look at variations and approaches that affect this separation. This section also introduces Bee Smalltalk, the system used for our case studies.

A Layered Architecture: Separating VM and Application Layers

As discussed in chapter 1, traditional virtual machines are designed to clearly separate the virtual machine from the application, providing a layered architecture.

This gives application developers a concrete and fixed target, enabling them for instance to port an application to compatible VMs as long they only rely on interfaces and behaviors guaranteed in a specification such as the JVM [LYBB14] or ECMAScript [Int15].

Features such as garbage collection and just-in-time compilation are automatic and VM developers will go through great efforts to make them unobservable. However, a VM may also offer APIs that enable an application to interact with the VM and configure it for its needs. For instance, many VMs allow an application to trigger garbage collection. Other common APIs may give access for instance to run-time statistics such as memory use, time spent on garbage collection or just-in-time compilation. Often these APIs will be limited to avoid disclosing implementation details and thus, to minimize the risk that an application will depend on them.

On the other hand, lower-level VM developers implement and optimize those features and provide access to them through some ad-hoc API. This creates a clear boundary between the application and VM, ensuring a clean separation of concerns and shielding application developers from the implementation details of the VM.

Popular VMs that use this layered architecture include CPython (the main Python VM), JVM, .NET's Common Language Runtime, and the ECMAScript-compatible JavaScript virtual machines, e.g., V8, SpiderMonkey, and JavaScript-Core. The common implementations of these VMs use a mixture of C/C++ and assembly code. This gives VM developers the control to reach the desired performance, in exchange for high development effort.

However, this two-layer design also has its problems, since features are provided intentionally as a black box to the application developer. This hides causal connections and prevents application developers from understanding and relying on implementation details.

To give an example, in languages such as Smalltalk and Self, it is usually said that everything is an object. These systems allow the application programmer to manipulate metaobjects such as methods, method dictionaries and classes or prototypes. By providing these components as properly abstracted objects, these systems allow programmers to more easily understand how the system works, and how to solve problems related to application and metaobjects interaction.

On the other hand, things such as the JIT compiler, the garbage collector, and the built-in runtime functions are not part of the runtime libraries, hence *not objects*. Thus, application developers cannot simply inspect the garbage collector to know under which conditions it triggers, nor observe the JIT compiler to know when a method is considered for compilation.

While many of these VMs are open source, they are typically not written in the same language as the application, and are not part of the codebase. This prevents application developers to use their usual development tools to observe, analyze, reuse, and change these components. The immediate feedback they get from their tools with few exceptions ends at the point of entering the VM layer.

2.4 Bee Smalltalk

Bee Smalltalk, our research vehicle, initially ran on top of a traditional VM implemented in C++ and assembly. Throughout this work we say Bee/SVM to refer to Bee Smalltalk running on top of that *statically compiled* VM. Bee/LMR was designed as a drop-in replacement for the traditional VM, and has a JIT compiler, garbage collector (GC), and numerous built-in functions. The main difference is that Bee/LMR is implemented in Smalltalk using a unified application/VM layer, while the traditional VM is split from the application and its C++ code is mostly invisible to application programmers.

Smalltalk methods are compiled into a bytecode format, and when first executed, are just-in-time compiled through a template JIT compiler. There is no interpreter mode.

2.4.1 Memory management

Both Bee/SVM and Bee/LMR have two garbage collectors: a generational copying collector for the new space heap and a compactor for the old space heap. The algorithms implemented in Bee/LMR for each of them are different from the ones implemented in Bee/SVM. While the characteristics of these GCs are explained in finer detail in chapter 5, we now shortly explain their main design layout.

In Bee/LMR GCs are metacircular and can be modified at run-time. They had to

be designed to keep objects valid during the garbage collection process, as described in [PBAM17]. To ensure objects are valid, they use object copying with external forward pointers, so that the GC only changes the mark bit of object headers instead of temporarily overwriting them completely. Bee/LMR compactor for the old heap is of garbage-first (G1) type [DFHP04]. New space consists of a big **eden**, and two smaller **from** and **to** spaces. When the **eden** is out of space, the generational scavenger is run. It moves most surviving objects in **eden** and **from** to **to**, and then flips **from** and **to**. Tenured objects are moved to the old zone, where they remain until the G1 collector is run. This is triggered by a heuristic that considers the heap growth since the previous G1 collection.

The old zone is divided in equally sized *spaces*, and there is a large object zone for objects that exceed a size threshold. Tenured objects are bump-allocated in the old spaces. As computation evolves some old objects become unreachable. The G1 collector keeps track of the reachable usage of each old space, where a low usage implies higher fragmentation. Before starting tracing, the GC selects spaces to be evacuated, choosing the ones that have the lowest usage ratios. Evacuation makes objects allocated in fragmented spaces become compacted into to other free spaces.

2.4.2 Other Characteristics

The Smalltalk system itself comes with the typical features of an image-based development environment. It uses as classic class browser with object inspectors and workspaces for arbitrary code evaluation. As for most Smalltalk systems, the debugger enables developers to inspect, explore, and modify the system at run time by changing state and code at will.

Bee is the platform for a simulation application in the oil and gas industry developed by a small team. Over time, the team realized that the strict separation between VM and application caused a too high cost, because the VM had to be maintained as a separate codebase. For example, investigating crashes no matter whether they were caused by the application or a VM bug took too much effort. As a small team with a production application of 1.1 million lines of code relying on Bee, it was decided that major steps had to be taken to make maintaining Bee easier for all team members. We describe the result in chapter 4.

Chapter 3

Motivation and Case Studies

To motivate our exploration of a runtime design that removes the barriers between application and VM, we exemplify the trade-offs based on three case studies from the development of our application on top of Bee.

We guide this work by considering three case studies taken from real-life development scenarios of an LPE-based product. Each case study describes a scenario and the solution steps needed in a state-of-the-art VM.

For each case study, we identify the specific obstacles with these VMs and the features needed to enable application developers to benefit from their normal tools and short feedback loops. We examine how these VMs make it hard to deal with the presented cases. Finally, we distinguish three features are needed to create exploratory programming environments with short feedback loops, where developers can more easily understand and improve VMs. In Chapter 4, we propose the usage of Live Metacircular Runtimes as a kind of VM that include those three features.

3.1 Case Studies

For this thesis, we selected a case study on the performance impact of garbage collection, one on avoding recompilations by the just-in-time compiler, as well as one on adding support for vector processor instructions for better performance.

These cases were gathered from real word experiences with an LPE based product

that used a traditional VM. All of them expose a shared category of problems when working with state-of-the-art VMs: they are hard to debug within LPEs, they require harnessing the VM in unanticipated ways, they pose a leap to application developers to configure the system in order to work with the VM, it is not practical to resolve them using live programming.

3.1.1 Case Study 1: Garbage Collection Tuning (GCT)

The allocation behavior of an application may trigger undesirable behavior by a garbage collector. GC bottlenecks are not uncommon in applications that allocate objects freely, with memory being reclaimed automatically with minimal intervention from the application code. However, memory managers are not infallible. For instance, too many allocations in a too small heap may cause too frequent collections, resulting in performance issues.

Certain allocation patterns and heap structures may also cause long GC pauses. In some cases though more frequent collections can faster remove large numbers of temporary objects, which may make a smaller heap size have an overall better performance.

When diagnosing such issues in an application, a developer dealing with this case will first needs to identify when GC occurs, and to collect statistics on heap size, object survival rates, and heap fragmentation. For long-running applications, they may also need to filter these statistics to specific parts, isolating the computation of interest.

A state-of-the-art VM may have a range of different collectors. For simplicity, we will only discuss a single-threaded scenario. Here a VM could have a generational GC that triggers when there is no space in the nursery to allocate new objects. VMs have various heuristics to trigger a minor or a full collection. An application may be able to use an API of the VM that brings access to basic GC statistics, or that allows to manually trigger a GC.

However, changing the GC or fine-tuning its parameters, often requires to start the VM with different parameters. Full GC details may only be available via console logs enabled by command-line parameters or monitoring APIs that can be accessed via external tools to observe GC and allocation details, as it is the case

for JVMs. Thus, in most cases, an application developer has to drop out of the application development environment and possibly learn a different tool.

All of today's solutions have in common that they maintain the strict separation between application and VM and rely on external mechanisms and tools to provide the desired insights.

For the specific performance problem at hand, after collecting the data, and analyzing it with new tools to be learned, or using custom code, we may find that one solution is to set a specific heap size while the relevant code is executing, which better balances the cost of GC and GC frequency. Though, adjusting the heap size mid-execution is not something generally supported by most VMs. However, modifying the VM is in the case of JVMs, JavaScript VMs, and many others is implausible for application developers. Not only will it require them to learn how to build these systems, but also to navigate possibly hundreds of thousands of lines of code.

In summary, for application developers, the available GC systems provide opaque interfaces that make it difficult to determine what happens behind the scenes and to gather GC statistics. Modifying a single line of code of the GC requires a significant initial cost of setup and building, and is a challenging task due to the switch to an external, opaque VM codebase, especially when written in a non-live language. Feedback loops are long, and details on how time is spent on memory-management routines are scarce. It is impractical to analyze GC triggering events, and the code of the GC can not be easily inspected or changed in the same way an application would be developed.

3.1.2 Case Study 2: Recurring Recompilations by the JIT Compiler (JITC)

The problem of generating the optimal target code from a source program is undecidable in general [ASU20]. Runtime engines implement heuristics that try to perform best in most frequent scenarios. When falling into unusual cases, they just try to make sure that performance degrades as gracefully as possible. Monomorphic and Polymorphic Inline Caches [DS84, HCU91], on-stack replacement [HU96, FQ03], are just some techniques that allow to deal with compiler adaptation to

improve performance dynamically. However, no single approach can cover all cases. Furthermore, some of these techniques are complex to implement and may not be available in the system.

Since run-time compilation comes at a cost, VMs also need to determine when JIT compilation is worthwhile. However, as with all heuristics, unusual cases may result in undesirable performance.

In the case study, a product update got deployed to a customer, which reports that the new version is much slower than the previous one. The underlying problem is that in the new version, the application code dynamically adds and removes methods to an object, which invalidates JIT-compiled code causing frequent recompilations.

On a state-of-the-art VM, to detect the cause of the performance issues the developer would start to profile the application. However, since JIT compilation is transparent, a profiler normally does not show compilation time. It is also unlikely that an application developer would think to look for compilation statistics, or perhaps notice that the compiler thread may be indicated as busy for longer than usual.

Indeed, our application developers were not able to find the issue using the profiler. However, they constructed a mechanism that allowed them to use a binary search through the changes in the update to identify the cause, which led them to the code change that added the dynamic method updates. Thus, because of the strict separation between VM and application, the standard tools failed our developers, and they had to rely on other tools using a different suitable development approach to find the root cause.

After identifying the that cause, our developers need a good understanding of the JIT compiler to devise a fix and avoid the recompilations. Even though the problem can be solved with a simple native-code caching heuristic, changing the JIT compiler code at fault is likely again too complex to the application developers. Thus, they will need to rewrite the application code to avoid adding and removing methods dynamically.

3.1.3 Case Study 3: SIMD Optimizations (CompO)

Applications such as ours, with a lot of floating-point arithmetics, can gain performance by using vectorized operations in modern x86 and ARM processors. However, many high-level languages do not provide compiler optimizations or APIs to use them.[1]

To optimize the methods that do the computation, our application developers need to either directly or indirectly use the vectorized floating-point operations (SIMD: single instruction, multiple data).

With state-of-the-art VMs, one could wait for or ask the VM vendor to add support for the vectorized operations. One possibility is to just wait for VM makers to implement the vectorized operations themselves. However, this may be impractical. For instance for Java, it is likely still going to take another few years before support is finalized. To give another example, while MMX instructions were added to Intel processors in 1997, it was only after 2014 that some JavaScript VMs started adding SIMD extensions, 17 years later!

An alternative is to use an extension to the VM, that relies on low-level code, which makes the operations accessible. JVMs provide the Java Native Interface and other VMs have typically similar mechanisms. Another approach, exocompilation, [IBR+22] allows abstracting data processing algorithms from *scheduling*. This simplifies generating efficient machine code that targets varying hardware instructions and architectures.

However, these approaches require the application developer to switch the language and tools, which comes with extra effort and likely a longer-than-usual bug tail. In the worst case, there may not even be compiler intrinsics and our developer may need to use inline assembly. Depending on how the native interface works, it may also come with extra inefficiencies when switching between normal and extension code, perhaps because it needs to marshal high-level arguments into low-level arguments before computation, and possibly also to save VM state before calling native code.

[1] Even Java only has experimental support: `https://openjdk.org/jeps/438`.

3.2 Problems with State-of-the-Art VMs

The common theme in our case studies is that the strict separation between application and virtual machine for all its engineering benefits comes also at a cost.

In each case study presented, the series of actions required for dealing with the problem in state-of-the-art VMs is augmented with another series of non-trivial actions, that increase the accidental complexity of the system.

3.2.1 Limited VM Observability (PG1)

Since VMs are designed to abstract and hide implementation details, directly available APIs are typically minimal. Even tooling interfaces as available for instance for JVMs may be limited by the desire to minimize run-time overhead when collecting data.

In our case studies, this meant that collecting the desired information about garbage collection either required to learn about external tooling or process log output. To understand the behavior of the just-in-time compiler, JVMs do provide the data as part of the tooling interfaces and the Java Management Extensions.[2] However, since it is transparent to the developer, not visible in profiles, and less commonly known as source for performance issue than GC, it is unlikely an application developer will consider it as a source of the issue.

3.2.2 Separate VM Development Mode (PG2)

Since as the author of this work, the readers may be VM developers, their first instinct can be *yes, let us fix the VM*. However, in the common case, working on the VM is too different from working on the application to easily transfer language and tooling knowledge. In most cases, the VM is implemented in a more low-level language, has complex build steps that need to be understood, and requires the use of different development tools. As such, making a change to a VM requires a lot of additional learning before one can even start to explore the typical VM codebases of hundreds of thousands of lines of code. Thus, the standard solution

[2] https://docs.oracle.com/en/java/javase/17/docs/api/java.management/java/lang/management/CompilationMXBean.html

for an application developer will be to find workarounds in the application code using the tools they already know.

3.2.3 Longer edit-compile-run feedback loops for VM Components (PG3)

In cases like our case study were we want to use vectorized operations in our simulation code, dropping to the level of VM development is the only option. While building extensions is often supported by documentation and tooling that requires less learning than changing the VM itself, it is still a change of programming language and tooling and comes with the burden of longer edit-compiler-run feedback cycles than what application developers are used to from their high-level languages.

With all this required learning, one can ideally change things, explore consequences, and have immediate feedback as application developers may be used to from their live programming environments. However, because extensions require lower-level languages, changes may require restarting the VM and application before taking effect, which is a drastic change in development flow. This lack of instant feedback increases the time from ideas to experiments and the cost of building such extensions.

3.3 Summary

We showed that the use of a strict two-layer architecture separating VM and application code for Live Programming Environments does not invite developers to deal with the three case studies.

On the contrary, such design poses knowledge and practicality barriers that are artificial, unrelated to the problems being dealt with in each case. The barriers that this architecture imposes prevent developers to detect and solve the problems that appear in their actual work.

The division between application and VM cuts the causal connection that associates the problems with the solution, increasing the time needed to find the way along codebases that can have hundreds of thousands of lines. These extra tasks to

be carried out can be considered too difficult for application programmers.

While the architecture ensures benefits such as for instance portability, it also limits the observability of VMs (PG1), typically leads to a separate development mode for the VM (PG2), and causes a longer edit-compile-run feedback loop for VM components than for application code (PG3).

Chapter 4

Live Metacircular Runtimes

To overcome the limitations presented in the Section 3.2, we propose the implementation of Live Metacircular Runtimes. We argue that LMRs create a synergy with live programming environments that makes it practical to modify at run time VM components that have traditionally been considered static.

In this chapter, we describe our research vehicle, Bee/LMR, which is an implementation of an LMR on top of Bee Smalltalk, a live programming environment that was originally designed to run using a VM written in C++. [1]

An LMR basically consists of a set of modules written in the same language as the applications, that replace what is usually shipped as a separate hidden layer, the VM. These modules can implement interpreters, just-in-time and ahead-of-time compilers, garbage collectors and built-in operations, everything that is needed to allow the system to execute code. The LMR modules sit at the kernel of the live programming environment. The fig. 4.1 shows the design transformation necessary to convert Bee/SVM into Bee/LMR.

Before delving into the implementation details of the LMR module, we describe the key aspects of Bee Smalltalk to help contextualize the whole picture of the system.

[1] While this thesis describes work done in Bee/LMR, which is not freely available, we are also developing an implementation of an LMR on top of a fully open-source Smalltalk dialect known as Egg. The LMR implementation on top of Egg is freely available.

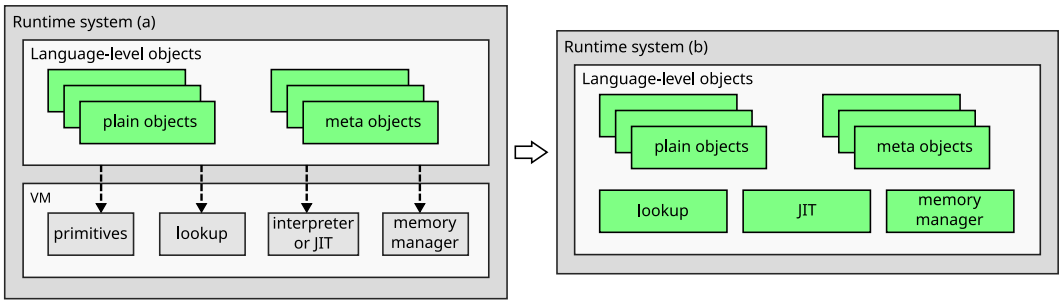


Figure 4.1: Transformation from Bee/SVM (left) to Bee/LMR (right). While Bee/SVM originally used a traditional VM, implemented in C++ and assembly in a separate layer and statically compiled, Bee/LMR is implemented as a series of common libraries written in Smalltalk, among which are the JIT, the GC, and a kernel with built-in functions such as method lookup.

4.1 Bee Smalltalk

Bee is a Smalltalk dialect loosely following the Smalltalk-80 specification [GR83] and Squeak Smalltalk [IKM+97] core classes. Its original VM, now retired, was implemented in C++ and assembly, and Bee had a two-layer VM/application design. This VM, which is statically compiled and cannot be easily modified at execution time is called Bee/SVM throughout this work.

To contextualize the changes required for the migration from Bee/SVM to Bee/LMR we start by giving an overview of the architecture of Bee and its VMs.

4.1.1 Execution Model

In Bee, Smalltalk code is precompiled from sources to compiled method objects that hold bytecoded instructions. The bytecodes use a virtual CPU architecture, so they cannot be executed directly. Bee does not implement an interpreter. Instead, just-in-time and ahead-of-time compilers translate method bytecodes to native assembly code when it is required. We generally call these compilers *nativizers*, and this last translation *nativization*.

Name	Contents	Saved by
R	message receiver and return value	scratch
S	**self** inside a method	callee
M	execution context of current method or block	callee
E	environment variables of current method or block	callee
A	1^{st} scratch argument	scratch
T	scratch temporary or 2^{nd} scratch argument	scratch
G	globals array	fixed
nil	nil object	fixed
true	true object	fixed
false	false object	fixed
SP	stack pointer	callee
FP	frame pointer	callee
PC	program counter	–

Table 4.1: Register names of the virtual Bee CPU

4.1.2 Virtual CPU Architecture

In Bee, both bytecodes and nativizers are designed around a virtual CPU architecture. This architecture consists of a stack and special purpose registers described in table 4.1. Different assembler backends map the abstract registers to concrete ones. For example, the AMD64 backend assigns **RAX** register to **R**, **RSI** to **S** and so forth. This design allows higher-level code generators to work with a single virtual architecture, so that changing the target ISA becomes mostly a matter of changing the assembler backend.

4.1.3 Native Code in Compiled Methods

When a method is nativized, it is assigned a **NativeCode** object that serves as an execution context for the method. This object contains a pointer to a byte array with the machine code that resulted from the translation of the bytecodes, and a list of literals that are referenced from that machine code. The machine code itself does not contain any pointer to Smalltalk objects, which facilitates garbage

collection as will be explained in chapter 5. Access to the literals of the method from machine code is done indirectly through M register, which gets initialized on entry to the method. There is no distinction between the byte arrays of machine code and other kinds of byte arrays, they are all objects stored in the same heap.[2]

4.1.4 Interactions Between VM and Applications

Originally, Bee ran on top of a statically compiled VM, which included the JIT-compiler, the built-in primitives and the garbage collectors. The VM was statically compiled and contained the basic functionality to allow running application code. The VM would load a Smalltalk image and start executing Smalltalk code by locating a launch method, JIT-compiling it and then jumping into its native code. The Smalltalk code would then run, occasionally calling back the VM in two ways: explicitly or implicitly. Explicitly through the execution of *primitive* compiled methods. Implicitly when causing invocation of a method that had no native code (triggering the JIT-compiler).

4.2 Migration from Split Layers Design to Bee/LMR Modules

The LMR modules work as a drop-in replacement of the original VM functionality. In the LMR implementation there is just no distinction between VM and application layers. The LMR code gets plugged into the system in the places where the interactions between both layers used to happen, but within Bee/LMR no specialized mechanism is needed to trigger VM code in a separate layer, as that layer *does not exist*. Objects and messages are all that is needed.

The migration from a traditional split layer design required three main components: a bytecode to native-code translator (which we call the nativizer), the built-in functions and the garbage collector. The built-in functions and the garbage collector require efficient access to parts of the VM which are usually not directly accessible from the language semantics (i.e. object headers and raw memory access).

[2]Which means all the heap is marked as executable memory, a design decision we might change in the future for the sake of improved security.

This is made possible in Bee/LMR by adding features to the nativizer, so we explain it first.

4.2.1 Philosophy

The main idea driving the implementation of Bee/LMR 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., built-ins, 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 tonegligible levels [FBC+09].

4.3 Customizable Nativizers

The first component that required replacement to implement Bee/LMR was the JIT compiler. While Bee/SVM's one was written in C++, in Bee/LMR the nativizer is written in Smalltalk. Current Bee/LMR design includes two nativizers mostly for performance reasons. A first-stage template JIT compiler iterates method bytecodes translating them to native code very quickly but doing only small optimizations. A second-stage optimizing compiler is able to perform more optimizations at the cost of lower nativization throughput.

For the purpose of allowing efficient access to parts of the VM usually hidden, compared to what the original Bee/SVM JIT did, only one modification was needed: the Bee/LMR nativizer allows customizing the translation of message send bytecodes, to perform custom actions with a set of specific message names, which are distinguished by having a starting underscore (_) character.

The mapping of special message names to special translations is determined by the configuration of a **NativizationEnvironment** object, which also knows the target architecture and other important details as explained next. The nativization environment is the entry point to translation of bytecodes into native code. This object references the components of the system that need to be known in order to produce native code: the target system CPU architecture, the special message send

Listing 4.1: Selection of **MessageLinker** by the **NativizationEnvironment**.

```
NativizationEnvironment >> emitSend: selector using: anAssembler
    | linker |
    linker := self dispatchLinkerFor: selector.
    linker emitSend: selector using: anAssembler

NativizationEnvironment >> dispatchLinkerFor: selector
    ^candidates
        detect: [:linker | linker canInline: selector]
        ifNone: [self error: 'cannot dispatch ', selector storeString]
```

configuration, write barriers, safe-point routines, globals, system ABI, etc.

4.3.1 Customizable Send Translation and Semantics

In traditional Smalltalk systems, message lookup is provided as an invisible runtime component. Usually, message sends are encoded with a send bytecode, and the VM is in charge of implementing an execution mechanism that performs the sends according to Smalltalk semantics. The VM will look for a method that implements the selector being sent in the class hierarchy of the receiverand, when found, will invoke the corresponding method.

While there exist mechanisms to workaround the dispatch behavior in traditional Smalltalks[3], those are high-level and do not allow to really modify the internal message dispatch mechanism provided by the VM. To be able to perform system-level operations using Smalltalk, such as accessing object headers and doing pointer arithmetic, it was necessary to implement, in high level, lower-level dispatch mechanisms.

Bee/LMR design allows to switch the dispatch mechanism arbitrarily at runtime on a message send basis. The base of the flexibility is the use of a series of dispatch linker hierarchy of objects, known as **MessageLinker**s, which work in collaboration with the **NativizationEnvironment**.

Currently, dispatch mechanism is determined by the sent selector: generation of native code for a message send is delegated to a composite message linker, which

[3] i.e. with **doesNotUnderstand:** or **perform:** mechanisms.

chooses a particular type of linking according to the selector being dispatched as shown in Listing 4.1. When the **NativizationEnvironment** object is sent **emitSend:using:** to nativize a send, it looks through its configured linkers to determine which one to use to generate code for the selector being passed as argument. Finally, it delegates to the subtype the generation of native code for that particular message-send. Different subtypes of message linkers implement different message send semantics.

When a method is nativized, for each message send that is not inlined, a **SendSite** object is attached to the native code of the method. The send site encodes all the information needed to execute the message send when it's time, to cache type information for inline caches and also to discard the cached information if application code gets updated.

Inline Message Linker and Underprimitives

This linker allows generating inline assembly code at a send-site instead of the native code required for normal message send semantics. Messages linked like this are known as *underprimitives*. Listing 4.2 shows the implementation of **_isSmallInteger** underprimitve. The method **assembleTestSmallInteger** interacts with an assembler object that will generate the required machine code.

Underprimitives are implemented in the lines of the ABI presented in table 4.1. In the case of **_isSmallInteger**, the assumption is that the receiver of the message will be in R register, and the result will be returned in that same register. This assumption is guaranteed by the JIT compiler and has to be maintained by the methods that implement underprimitives.

From the Smalltalk parser's point of view, the underscore at the beginning of a message name has no special semantics. It just is the nomenclature used to warn the programmer that a message has some special meaning such as being low-level or an underprimitive. Neither it has a special semantics when compiling to bytecodes. The only change in semantics occurs in the execution of the message, through the nativization mechanism.

The selectors implemented as underprimitives are defined dynamically through metaprogramming. Methods implemented in the class **InlineMessageLinker** that belong to the **underprimitive** category are automatically added to the list of un-

Listing 4.2: Example of **_isSmallInteger** underprimitive implementation and usage.

```
InlineMessageLinker >> assembleTestSmallInteger
    | integer |
    #_isSmallInteger.
    integer := assembler newLabel.
    assembler
        testRintegerBit;
        loadRwithTrue;
        shortJumpIfNotZeroTo: integer;
        loadRwithFalse;
        @ integer

ProtoObject >> behavior
    ^self _isSmallInteger
        ifTrue: [SmallInteger instanceBehavior]
        ifFalse: [self _basicULongAt: 0]
```

derprimitive selectors of the message-send linker. In the exaple of Listing 4.2, **#_isSmallInteger** is automatically configured as an underprimitive, and the JIT will emit the code generated by **assembleTestSmallInteger** for any **#_isSmallInteger** messages it sees.

In Bee/LMR, underprimitives are the base tool used to efficiently access object headers. Listing 4.3 shows the implementation of a **_size** method. When receiving this message, the receiver returns the size stored in the object header. Both **_smallSize** and **_largeSize** are implemented as underprimitives. On the other hand, **_size** and **_isSmall** are Smalltalk methods which will be statically linked, as described next.

4.3.2 Invoke Message Linker

In some particular situations it becomes useful to implement a message send as if it were a static function call. Instead of issuing a full method lookup for the type of the receiver of the sent message, this linker emits a statically linked invoke call for a preset method that corresponds to the message selector.

Listing 4.3: Usage of underprimitives that access the headers of objects and their implementation.

```
ProtoObject >> _size
    ^self _isSmall
       ifTrue: [self _smallSize]
       ifFalse: [self _largeSize]

ProtoObject >> _isSmall
    ^(self _basicFlags bitAnd: IsSmall) = IsSmall


InlineMessageLinker >> assembleBasicFlags
    #_basicFlags.
    self emitByteAtOffset: _Flags

InlineMessageLinker >> assembleSmallSize
    #_smallSize.
    self emitByteAtOffset: _SmallSize

InlineMessageLinker >> assembleExtendedSize
    #_largeSize.
    assembler
        loadZeroExtendLongRwithRindex: _ExtendedSize;
        convertRtoSmallInteger
```

Invoked messages can be used as an optimization, to save the cost of executing lookup. This is specially useful with low-level methods implemented in the root of the class hierarchy, at **ProtoObject** class. Methods implemented in **ProtoObject** are prone to send messages also implemented in **ProtoObject**. But as the concrete receivers of those messages can be of any type, these message sends tend to become megamorphic. Invoked messages are statically linked at JIT time, and do not involve class checks, making this kind of code run faster.

Consider the method **ProtoObject»_size** presented in Listing 4.3, which sends the **_isSmall** message, that is also implemented in **ProtoObject**. Suppose the message **_size** is sent to an object of type Array. During execution, the **_isSmall** message will be sent to the same receiver, of type Array, and an inline cached would be created mapping **ProtoObject»_isSmall** to class Array. If **_size** were now sent to an object of other type, the inline cache would fail, and a polymorphic cache would be created, to add this second type, but mapping the same implementation. Thus, for each specific type of **_size** receiver, the inline-cache of **_isSmall** would add an entry, degenerating into a megamorphic cache and consequently degrading performance.

4.3.3 Template-JIT Method Nativizer

The default method nativizer decodes method bytecodes iterating them one-by-one until it has generated native code for the whole method. This nativizer avoids doing optimizations such as code transformation or register allocation. Instead, it relies on optimizations done to the bytecodes by previous precompilation stages, which may for example reorder bytecodes to save stack and register-copying operations.

Listing 4.4 shows the nativization algorithm. It starts in **translateMethod**, which generates native code for the method and its blocks. This translation process involves iterating the bytecode stream, translating each bytecode at once, with a specialized bytecode nativizer for each bytecode type. An example of different bytecode nativizers is shown in Listing 4.5.

The case of send bytecodes is the only one that requires special care compared to Bee/SVM, because of the added send semantics in Bee/LMR. As shown in

Listing 4.4: Implementation of the template bytecode-to-assembly method translator

```
TemplateBytecodeNativizer >> translateMethod
    self
        emitMethodPrologue;
        translateFrom: currentPC to: self bytecodeSize;
        emitEpilogue.
    blocks do: [:blockInfo | self translateBlock: blockInfo]


TemplateBytecodeNativizer >> translateFrom: start to: end
    currentPC := start.
    [currentPC < end]
        whileTrue: [
            self translateSingleBytecode: self nextBytecode
        ]


TemplateBytecodeNativizer >> translateSingleBytecode: bytecode
    | nativizer |
    codeOffsets at: currentPC put: self position.
    self addLabelIfNeeded.
    nativizer := self templateNativizerFor: bytecode.
    ^nativizer assemble
```

Listing 4.5: Translators for two different bytecodes: **LoadFalse** and **LoadInstanceVariable**

```
LoadFalseBytecodeNativizer >> assemble
    assembler loadRwithFalse


LoadFalseBytecodeNativizer >> assemble
    self usesSelfRegister
        ifTrue: [assembler loadRwithSindex: self instanceNumber]
        ifFalse: [assembler loadRwithRindex: self instanceNumber]
```

Listing 4.6: Translation of the send bytecode into native code

```
SendSelectorBytecodeNativizer >> assemble
    methodNativizer emitSend: self selector


TemplateBytecodeNativizer >> emitSend: selector
    environment messageLinker
                    emitSend: selector
                    using: assembler
```

Listing 4.6, the nativizer for send bytecodes delegates the nativization task to the method nativizer, which in turn passes it to the corresponding message linker. The last stage of this process was shown in section 4.3.1.

For performance reasons, send bytecodes for arithmetic and logical operations are inline optimized for common cases such as comparing for equality or adding integers. Listing 4.7 shows the translation of #= message send, and the optimizations done at that translation.

4.3.4 Optimizing Method Nativizer

While the template JIT compiler generates native code of good-enough quality for most scenarios, there were cases where a more powerful compiler was desired. In particular, the Smalltalk code that replaced the built-in functions of Bee/SVM is executed intensively, hence small optimizations provide big payoffs. For that reason, an optimizing compiler was implemented in Bee/LMR.

Unlike the template JIT, the optimizing compiler generates a graph based intermediate representation of Smalltalk code, following the same directions as [Cli93, BBZ11, DWS+13, LKH15]. This intermediate representation is then optimized and finally native code is generated from it.

The main strength of this optimizer is being capable of inlining methods and block closures, doing value numbering and performing peephole optimizations. The compiler is designed to allow for more optimizations in the future. Particularly relevant shall be those related to dynamic compilation such as adaptive recompilation [HU94], escape analysis [PG92, SWM14], and also others related

Listing 4.7: Native code of send bytecode for #= message is optimized in two ways. If both the receiver and argument point to the same thing, **true** is returned without doing any call. If not, but the receiver is a **SmallInteger**, **false** is returned, neither doing a call. Otherwise, a normal call is done to the lookup mechanism (which includes inline caching).

```
SendEqualBytecodeNativizer >> assemble
    | retTrue failed unoptimized end |
    retTrue := assembler newLabel.
    failed := assembler newLabel.
    unoptimized := assembler newLabel.
    end := assembler newLabel.
    assembler
        popA;
        compareRwithA;
        shortJumpIfEqualTo: retTrue;
        ifRNotSmallIntegerJumpTo: unoptimized;
        loadRwithFalse;
        shortJumpTo: end;
        @ unoptimized;
        pushA.
    self emitSend: #'='.
    assembler
        @ retTrue;
        loadRwithTrue;
        @end.
```

to compilers in general, such as or loop invariant code motion.

The compiler backend implements an SSA-based linear scan register allocator [PS99, WF10].

This compiler is only used ahead of time, during bootstrapping, to optimize a fixed set of methods, most of them being the ones that replaced the built-in functions of Bee/SVM.

Optimizing Compiler Type System

The nodes in the IR graph of the optimizing compiler do not store type information and are assumed of the generic type **ProtoObject**. This means that intermediate numerical computations are not done using native words of the CPU architecture but using Smalltalk integer objects. This produces less efficient code, because each arithmetic and logical operation emitted includes a conversion from small integers to native and back. However, this design allows seamless interaction with the garbage collector. The code generated by the compiler is always GC-safe: no matter where the GC interrupts optimized code, all values in stack and registers are pointers to objects that can be safely traversed.

As Smalltalk code does not contain type annotations and the compiler is run ahead of time, the inlining mechanism needs a little help to be able to detect which methods the lookup algorithm would find for each message send in the code. As the set of optimized methods is fixed and consists of very simple code (in the end, it replaces C++ statically typed code), the type information required by the compiler is provided manually through type annotations.

4.3.5 Avoiding Recursive Nativizer Invocation

As Bee/LMR does not include any interpreter, all Smalltalk code to be executed has to be nativized before execution. The JIT-compiler is written in Smalltalk and uses instances of standard Bee classes. For that reason, to JIT-compile Smalltalk code the system needs to execute Smalltalk, which could in turn need to recursively invoke the JIT compiler, causing an endless loop.

To avoid that situation from happening, the kernel methods of Bee/LMR are ahead-of-time compiled during bootstrapping (as described in section 4.5.1).

Listing 4.8: The **String»#byteAt:** implementation in both Bee/SVM (up) and LMR (down).

```
String >> byteAt: anInteger
    <primitive: StringByteAt>
```

```
String >> byteAt: anInteger
        anInteger isSmallInteger
            ifFalse: [^self error: 'Non integer index'].
        (1 <= anInteger and: [anInteger < self _size])
            ifFalse: [^self outOfBoundsIndex: anInteger].
        ^self _byteAt: anInteger
```

Bee/LMR nativizer is written to only need executing methods in the kernel module, so it never triggers a recursive call to itself while working on a method. When a developer modifies methods of the kernel module, they are nativized just before being installed. The rest of the methods in the system are just-in-time compiled. This happens each time the runtime detects that a method that is about to be executed has not yet been nativized.

4.4 Implementation of LMR Built-in Functions

Bee/SVM includes around 200 primitives that allow accessing built-in functions of the VM. In Bee/LMR there are no such thing as primitives, but instead there are about 150 special selectors (starting with _). These messages usually have shorter implementations than built-in functions, are written in Smalltalk and can be changed dynamically.

Just to give an example, Listing 4.8 shows the change from the implementation in Bee/SVM to the one in Bee/LMR. Notice how a primitive that accesses raw memory can be implemented in terms of the **_byteAt:** and **_size** metamessages.

4.4.1 Message Lookup

Bee/LMR implements the typical message lookup algorithm of any Smalltalk, traversing method dictionaries in the superclass chain until an implementor is found or sending **#doesNotUnderstand:** in case no implementor is found.

The implementation is fairly simple, and its code is shown in Listing 4.9. The **_lookup:** method is invoked when dispatching a message. The result of lookup can be cached, and the **_lookup:** method is invoked only if the cache does not contain a matching entry. The entrypoint to the dispatch algorithm is shown in Listing 4.10. The caching strategies are described in section 4.4.2.

After the lookup is done[4], the algorithm prepares the method for execution, assuring that it contains native code, generating it if necessary.

The lines at the beginning of **_dispatchSend:** act as a pragma to the nativizer to generate code that expects the argument of this method to be in a special register instead of the stack.

As the algorithm is written in Smalltalk, a naive execution of the lookup algorithm would require recursively calling lookup for each of the messages sent, causing an infinite lookup loop.

To solve this problem, the **_dispatchSend:** method is nativized using a special **NativizationEnvironment** that is configured replace message lookups with direct method invocations. This static linking is possible because the methods that have to be invoked on each message send of lookup algorithm can be pre-computed. The closure of the methods involved in the lookup algorithm is a small set of methods, so it is no problem to compute it manually.

4.4.2 Lookup Caching

In Bee/LMR, each message send site is associated to an instance of **SendSite**. **SendSite** objects are designed to allow implementing inline caching mechanisms [CPL83, Ung83, DS84, Ung86]. The first slot of a send site is named **dispatch**, and points to the native code of a dispatch routine. Execution of a message send is done by pushing the arguments into the stack, loading the send site object into a register,

[4] If no method is found then **nil** is returned, which causes **doesNotUnderstand:** message to be sent.

Listing 4.9: The lookup algorithm consists on looking for a method implementing the sent selector in the behavior of the object that receives the message. The behavior of an object is a linked list of method dictionaries that follows the corresponding class hierarchy.

```
ProtoObject >> lookup: aSymbol in: aBehavior
    | methods cm next |
    methods := aBehavior _basicAt: 1.
    cm := self _lookup: aSymbol inDictionary: methods.
    cm == nil ifFalse: [^cm].
    next := aBehavior _basicAt: 2.
    ^next == nil ifFalse: [self _lookup: aSymbol in: next]

lookup: aSymbol inDictionary: methodDictionary
    | table |
    table := methodDictionary _basicAt: 2.
    2 to: table _size by: 2 do: [:j |
        (table _basicAt: j) == aSymbol
           ifTrue: [^table _basicAt: j + 1]].
    ^nil
```

Listing 4.10: The entrypoint to dispach mechanism. The argument is received in a register instead of the stack. The implementation of **_cachedLookup:** and **when:use:** is shown in section 4.4.2.

```
ProtoObject >> _dispatchSend: aSendSite
    | cm nativeCode |
    #specialABIBegin.
    #aSendSite -> #regA.
    #specialABIEnd.
    cm := self _cachedLookup: selector.
    cm == nil ifTrue: [^self doesNotUnderstandSelector: selector].
    cm prepareForExecution.
    nativeCode := cm nativeCode.
    aSendSite when: self behavior use: nativeCode.
    ^self _transferControlTo: nativeCode
```

Listing 4.11: The assembly for a message send in AMD64 architecture. RBX register contains the **NativeCode** object corresponding to the currently executing method. This object contains all the send site objects present in the method.

```
push arg1
...
push argN
mov RDX, [RBX+off_send_site_i] ; load the i-th send site object
                               ; into RDX from the current context
call [RDX]
```

and calling the routine stored in the first slot. This is shown in Listing 4.11.

When a send site is instantiated, the **dispatch** slot is initialized to the **ProtoObject»#_dispatchSend:** routine. During execution, the first time the send site is reached, the code of **_dispatchSend:** is invoked. Instead of calling the lookup directly, this method first looks in a global dispatch cache, an array of <**Symbol**, **Class**> pairs that stores the result of the lookup for that pair. If found, the lookup in method dictionaries can be skipped, resulting in a performance gain. If not found, the algorithm performs the lookup and stores the result in the global cache. The global caching algorithm is shown in Listing 4.12.

The **_dispatchSend:** algorithm also creates an inline cache of the result of the lookup. The send site holds a **cache** instance variable to store this information, which consists of an array of <**Symbol**, **Class**> pairs. After adding an entry for the method found for the current send, the algorithm changes the contents of the **dispatch** slot, sot that it points to a stub that looks in the inline cache before going to the global cache. This makes lookup more efficient the next time that the same send site is used. The first time the site is used it creates a monomorphic inline cache, as shown in Listing 4.13. The **monomorphicStub** assigned to the send's **dispatch** instance variable is described in Listing 4.14.

As previously stated, the send mechanmism is made of common objects: **SendSite** is a normal class of the system, the **dispatch** instance variable points to just a byte array object, and the **cache** to an array. When a monomorphic cache fails, dispatch mechanism replaces the array of the monomorphic cache with a larger one,

Listing 4.12: Global lookup caching algorithm of Bee/LMR

```
ProtoObject >> _cachedLookup: aSymbol
    ^self _cachedLookup: aSymbol in: self behavior

ProtoObject >> _cachedLookup: aSymbol in: behavior
    ^GlobalDispatchCache current lookupAndCache: aSymbol in: behavior

GlobalDispatchCache >> lookupAndCache: aSymbol in: aBehavior
    | method |
    method := self at: aSymbol for: aBehavior.
    method == nil ifTrue: [
        method := self _lookup: aSymbol in: aBehavior.
        self at: aSymbol for: aBehavior put: method].
    ^method
```

Listing 4.13: Monomorphic inline caching algorithms of Bee/LMR.

```
SendSite >> when: aBehavior use: aNativeCode
cache == nil
    ifTrue: [self monomorphicMap: aBehavior to: aNativeCode]
    ifFalse: [self polymorphicMap: aBehavior to: aNativeCode]

SendSite >> monomorphicMap: aBehavior to: code
cache := self takeNextFreeMIC.
dispatch := self monomorphicStub.
cache
    at: 1 put: aBehavior;
    at: 2 put: code
```

Listing 4.14: The assembly code of the monomorphic stub. Initially, RAX, RDX and R15 contain the receiver, send site and globals array, respectively. The stub loads the cache of the send site into RCX, then the behavior of the object into RSI, then performs the test and, in case of success, jumps to the target native code. In case of failure the dispatch stub loads the **_dispatchSend:** native code from the globals array (R15) and jumps into its native code.

```
monomorphicStub:
    00:         mov rcx, qword ptr [rdx + 0x10]
    04:         mov rsi, qword ptr [r15 + 0x20]
    08:         test al, 0x01
    0A:         jnz @1
    0C:         mov esi, dword ptr [rax - 0x04]
@1: 0F:         cmp rsi, qword ptr [rcx]
    12:         jnz @2
    14:         mov rbx, qword ptr [rcx + 0x08]
    18:         jmp qword ptr [rbx]
@2: 1A:         mov rbx, qword ptr [r15]
    1D:         jmp qword ptr [rbx]
```

Listing 4.15: Polymorphic inline caching algorithms of Bee/LMR

```
SendSite >> polymorphicMap: aBehavior to: code
cache _size == 2 ifTrue: [
    cache := self takeNextFreePIC.
    tally := 0.
    dispatch := self polymorphicStub.
    self bePolymorphic].
aBehavior == SmallInteger instanceBehavior
    ifTrue: [cache at: self maxSize + 1 put: code]
    ifFalse: [
        tally == self maxSize ifTrue: [self reset].
        cache
            at: tally + 1 put: aBehavior;
            at: tally + 2 put: code.
        tally := tally + 2]
```

that has enough space to hold polymorphic cache information. The polymorphic inline caching algorithm is shown in Listing 4.15. The corresponding assembly is similar to the one of the monomorphic but with more cases.

4.5 Wrapping up the LMR into Bee

4.5.1 Bootstrapping

The process of creating an executable runtime system from code written in the very same language that the runtime executes is known as *bootstrapping*. It can pose a problem when the language lacks an already working runtime system[5], as would not be able to run the code that generates the executable. Luckily, in the case of Bee/LMR we counted with Bee/SVM, which of course counts with already bootstrapped C++ compilers and assemblers. Therefore, it was possible to generate the Bee/LMR kernel image from the running system, placing inside it the native code of all the methods that belong to the kernel module.

During the creation of this image, things that refer to the host VM are removed.

[5]This usually happens when the language is in process of being implemented for the first time.

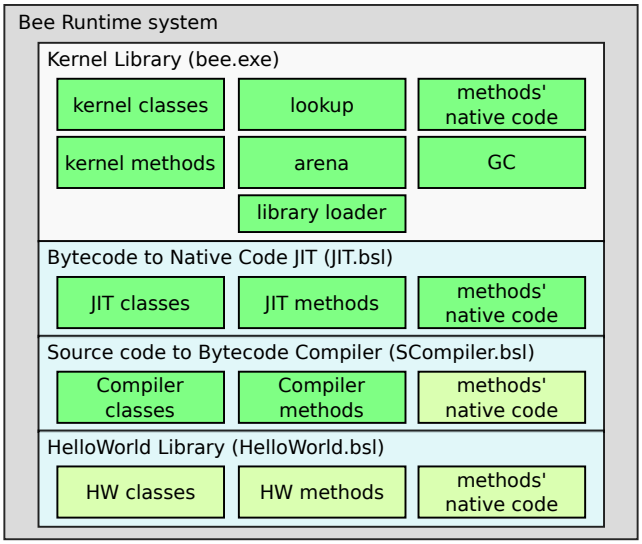


Figure 4.2: Bee modules. Kernel library contains minimal functionality to support itself and to load other libraries. Loading of JIT and Bytecode compilers is optional. Except for kernel and JIT, libraries can be shipped with just source code, with precompiled bytecodes or with native code.

Methods that refer to primitives, or which access state and code of the host VM like memory spaces, lookup, GC and the library loader are replaced with their Smalltalk-implemented complementary versions. The JIT is implemented as a separate module, which is written using the same mechanisms as the kernel module, and that can be loaded when needed. A high-level view of the ecosystem is shown in Figure 4.2.

4.5.2 LMR Development Environment

The development environment used for implementing Bee/LMR is the same used for normal applications in Bee, which is a traditional Smalltalk. Before Bee/LMR was fully functional, Bee/SVM was used to run the environment. Bee/SVM is also useful when implementing changes to critical parts of the system, if developers want to avoid crashes of the system they are running on.

4.6 Related Work

Various projects have experimented with changes and variations of the two-layer architecture.

4.6.1 Self-hosted Virtual Machines

Lisp's tower of interpreters [Smi84] and Smalltalk's self-hosted Squeak [IKM+97] are classic examples, which to a certain degree achieve the goal of implementing a language in itself. More recent examples of this approach include PyPy [RP06], Rubinius [FM08], JikesRVM [AAB+05], Squawk [SC05], Maxine [WHVDV+13], and SubstrateVM [WSH+19].

The self-hosted VMs are mostly written in the same language they support or a subset of it. While Lisp's tower of interpreters has the ability to change the language at each level, the approach more widely used for VMs can be seen with OpenSmalltalkVM for Squeak and PyPy, which are translated to C and then compiled statically with a standard C compiler. JikesRVM, Maxine and SubstrateVM use a bootstrap VM to directly generate a native executable image instead of C code.

The C-code approach gives some leeway and enables Squeak for instance to provide simulation tools that allow it to execute the VM code the same as normal Smalltalk code, which makes it possible to use the same development tools for developing the VM as for the application. However, this simulation is often around 1000x slower than the real VM [BKL+08]. Similarly, the PyPy implementation can be executed as normal Python code and developers can use their standard Python tools to work with it.

The biggest drawback is the low performance of executing a metacircular VM on top of itself, which is not optimized and meant to be used in production by application programmers. Self-optimizing AST interpreters [HWW+15, WWS+12] 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.

Furthermore, these approaches still have the two-layer architecture discussed

before, which clearly separates applications from the VMs.

We illustrate these designs in Figure 4.3. On top, fig. 4.3a shows the two-layer architecture, while next fig. 4.3b depicts self-hosted VMs that can run the VM code as an application.

4.6.2 Designs with Extensible Metaobject Protocols

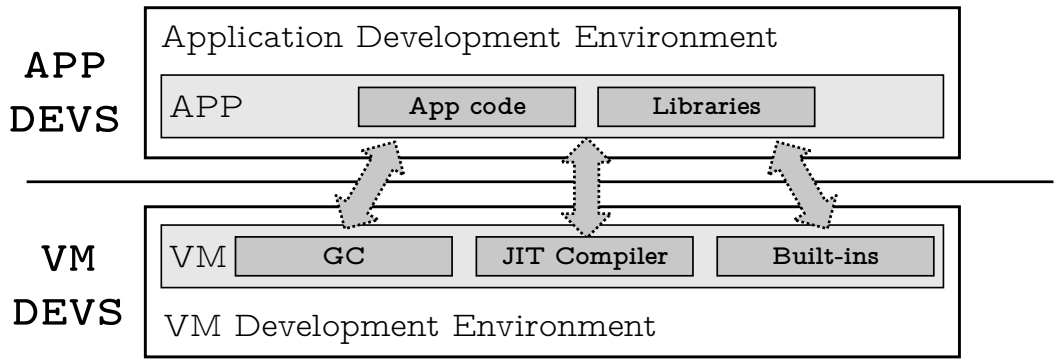
While still maintaining a two-layer architecture, Fully Reflective Execution Environments (FREE) [CGMD15, CGM16] aims to widen the APIs provided by VMs to enable the customization of ideally all aspects of a VM. This is achieved by designing metaobject protocols that can for instance change how method lookup is done, fields are accessed, or even how garbage collection and JIT compilation are done. However, as illustrated in fig. 4.3c, this still maintains the architectural distinction and does not give developers access to the implementation of the VM itself, only providing more abilities for customization. Pinocchio [VBG+10] is another example for a Smalltalk VM where the interpreter itself can be adapted via a metaobject protocol.

On the two layer designs, while theoretically it would be possible to update parts of the VM code while running, in practice no system is really meant to be lively programmable in production environments.

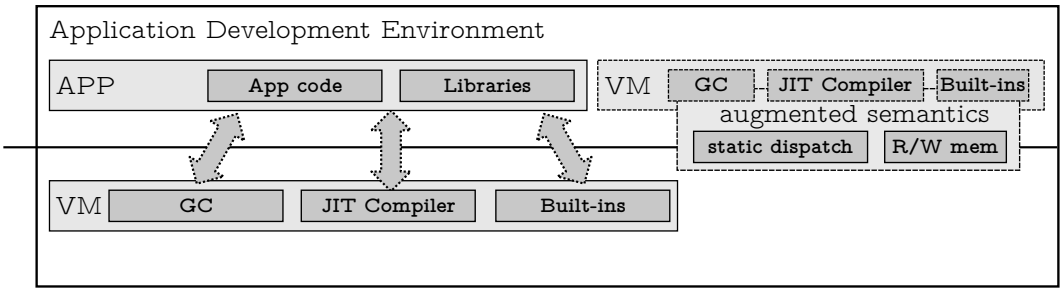
4.6.3 Beyond Object-Oriented Virtual Machines

Outside the world of object-oriented VMs, projects like SML# [OUH+14] aim to combine application and runtime by compiling both into a static binary. Though, while the end product is a combination, SML# still separates the runtime from the application.

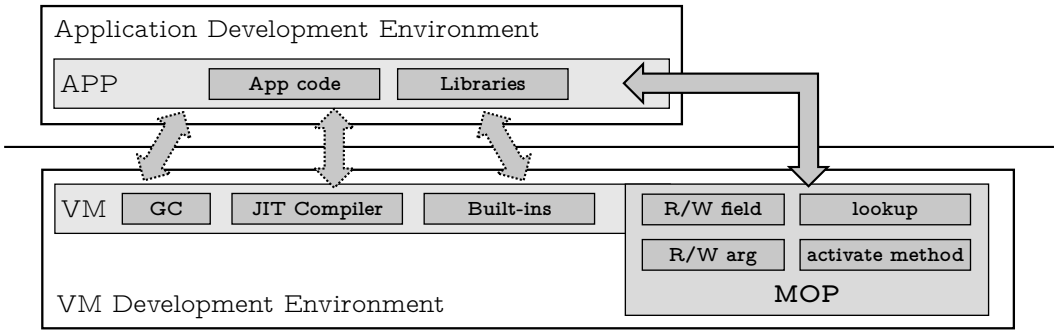
For general applications, Kiczales [Kic96] proposed the notion of *Open Implementations*. In addition to a standard interface, a module is expected to provide a meta-interface. This meta-interface can then be used to control and adapt the implementation of a module, perhaps to select an algorithm that has better performance in a specific context. This idea builds on the notion of metaobject protocols [KDRB91] and metaarchitectures, in the context of which research explored a wide range of issues and designs [Tan09], also including compile-time



(a) Two-layer architecture of traditional VMs, clearly separating application and VM, with limited APIs for interaction.



(b) Self-hosted VMs maintain the layered architecture, but can allow developers to execute the VM as if it is an application.



(c) VMs with Metaobject Protocols to customize field access, method lookup, and other aspects from within an application.

Figure 4.3: VM implementation approaches with the traditional two-layer design.

metaobject protocols [Chi95].

Though, these approaches are what Fully Reflective Execution Environments [CGMD15] built on, which still relies on a fixed meta-interface, and thus, enforces a strong boundary between modules, or in our case VM and application.

4.6.4 Designs with Reduced VM and Application Boundaries

Klein [USA05] pushed the boundary a bit further. It was a self-hosted VM written in Self for Self, that was object-oriented, metacircular, and reactive, meaning that it could be changed while running. In our understanding, it blurred the line between VM and application much more than any of the other approaches. Thus, it is an inspiration of our work, a first attempt to create a VM that does not strictly separate the application and can be evolved as one evolves an application. Unfortunately, Klein has neither implemented a self-hosted GC nor achieved the full Self functionality and performance.

Another point of inspiration is the Mist project,[6] which is explained as *a Smalltalk without a VM*. To our understanding, it also aimed at being a self-hosted Smalltalk implementation that compiles itself to native code, but is still at the early idea stage.

[6] `https://mist-project.org/`, Martin McClure, 2016

Chapter 5

Garbage Collection and Memory Management

In Bee/SVM the memory manager is invisible to the application. The interface to access it from the application is minimal, only allowing to trigger major GC passes and asking aobut the heap size.

On the other hand, in Bee/LMR the garbage collector model is placed at the language level, and modelled as any other entity of the system that has shape and behavior: with objects.

5.1 Implementation Challenges

In LMRs the object heap is managed by a lively programable set of objects, and there is no low-level interface to the garbage collector functionalities, but only a high-level unified model where *everything* is an object. At implementation time, this unification posed a series of challenges that had to be tackled in order to create a working solution.

5.1.1 Garbage Collection of Runtime Objects

A first challenge is that LMRs add runtime entities to the already existing set of objects of metacircular environments. This includes allocation spaces, stacks,

arrays, memory and even the GC itself and its objects, which 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 traced.[1]

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.

5.1.2 Execution Context Unavailability During Garbage Collection

A second problem arises in LMRs also as a consequence of having a common paradigm for VM and application-level objects. Because of the split-layer design, typical garbage collectors, even in self-hosted implementations like those of Squeak or PyPy, work within a separate execution environment that is disconnected from the environment being collected.

Methods and classes of those garbage collectors belong to a separate layer that is unreachable while traversing roots of the collected space. Within the heap, the GC algorithm can freely change object pointers as needed to adjust references. However, on the other hand the GC of LMRs are made of standard objects living in the same memory as the application.

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.

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

For the reasons described above, a naive implementation of a GC for LMRs cannot pause all high-level execution, as its own code is written at high-level. LMR 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 heap 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 and pointer reversal in mark-and-compact [JHM11].

Since the GC is a crucial component that needs to be work at all time, we need to ensure *always working objects*, and prevent *lost writes* and *lost objects*. This restricts design choices for GC algorithms but allows us to build a reliable system.

Always working objects

As the GC code depends on the interaction of common objects, during GC it is not possible to apply GC algorithms that temporarily overwrite object data, such as those that install forwarding pointers into object headers or that thread pointers.

Forwarding [FY69] is a GC technique that overwrites the original header of moved objects with forwarding pointers to their new locations. *Pointer reversal* [SW67] is a technique used to eliminate the need of extra memory for a tracing stack, using the slots of the objects to implement a linked list that is used as a stack-like structure. In the mark-and-compact algorithm, *pointer threading* [Mor78, Jon79] is done to efficiently find all references to each moved object.

With such algorithms, if the GC tried to send a message to a temporarily overwritten object it would cause undefined behavior because it would not find the expected information of the object in its header, such as its class or its size.

Lost Writes

The garbage collector has to avoid moving objects that might change during garbage collection. During GC, objects are typically moved (copied) to another area when being reached for the first time when tracing the object graph. As the GC continues tracing, it can find more references to the moved objects and update them with the new addresses.

If the GC modified a moving object, it could cause inconsistencies because it may be impractical to determine whether an object is the new copy or the old one, and during GC some objects might still see the original version instead of the modified one because they have not been traced yet.

Lost Objects

The garbage collector switches the allocation area when a collection is triggered, so that new objects needed for GC do not get mixed with the rest and can get discarded as soon as GC finishes. This means that no new objects created during GC survive after GC ends.

In particular, it is not allowed to JIT-compile methods during GC for that reason. The system must assure that any code required for executing GC has already been compiled before the GC starts.

5.1.3 Garbage Collection Debugging

As part of a live 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 live 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. A solution to this problem is given in chapter 6, through the use of an out-of-process debugger.

5.2 Design Directions

One of the toughest challenges when implementing Bee/LMR 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.

5.2.1 Two Possible Approaches

We discovered two possible approaches to achieve our goal. The first one is to use a standard garbage collection algorithm and to disconnect its execution environment from the garbage collected space. This avoids the problem of the GC collecting itself, i.e., traversing the data structures it uses, and also the problem of exposing inconsistent object state. This approach requires generating a code closure of all the methods and objects involved in GC, and compiling them into a separate heap. As the system is dynamically typed this is a cumbersome step, because the lack of type annotations makes it hard to find such a closure.

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 restricts the GC design more than the first approach and might have performance implications, but avoids having to determine a code closure for the GC.

We implemented both approaches. We shortly describe the first one next, to then explain why we decided to only support the second, which is fully described in the following section.

A Garbage Collector in a Bottle

Using the first approach we implemented a mark-and-compact threading collector for the old space and a generational scavenging collector [Che70]. We statically generate a closure of the GC code and the objects required for it, by manually picking the methods that go into the closure. 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. The collector uses a separate space for new objects

it creates, which it discards after collection finishes. This makes the GC runtime independent of the original objects. The benefit is that important objects, such as for example the **Object** class, can be left in inconsistent state during GC, as the GC library uses its own copy of that same object.

Other GC algorithm variations were implemented afterwards tuning the internal structures of generational and mark-and-compact collectors, and even a multithreaded 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. For debugging the integration as a DLL, we first had to resort to low-level tools, however that problem was solved by the design of the tools described in chapter 6.

Manually maintaining the closure of methods and objects involved in GC proved to be a time-consuming task, because this closure consists of hundreds of methods. While the first GC approach is powerful and allows for more efficient algorithms than the second one because it does not have the limitivations mentioned before, we decided to drop support for it until we implement a type inference system that is capable of mostly automatically finding the code closure of the GC code. This was left as future work.

5.3 Overall Design of Bee/LMR Garbage Collectors

This section details the implementation of the second GC approach, a garbage collector that can traverse itself.

The main point of access to the set of objects that perform memory managment is an instance of the class **Memory**. This object provides an API to allocate memory for new objects, to iterate through the objects of the different parts of the object heap, to collect heap usage data and to release memory through a pair of generational and old-space heap garbage collectors.

The memory object also implements operations such as finding references to objects in the heap and stack, and converting references from one object to another,

that is, to implement *become*.

To be able to have a working execution environment throughout the process of garbage collection, for the old space we switched from the mark-and-compact algorithm with object threading implemented in the first approach to a standard copying collector. This has an impact on memory usage, however the copying collector algorithms have characteristics that aid in leaving objects in operational state during collection. Specifically, the main change compared to traditional copying collector implementations was the use of external forwarding tables, so that during GC the only modification done to objects in heap is the mark bit. This external forwarding table technique was used also for the generational GC.

5.3.1 Object Heap Layout

The process addressing space is divided into chunks of memory called **GCSpaces**. Each GC space represents a virtual address range allocated by the Operating System. The memory of each space can either be committed or just reserved.

A small portion of the heap is defined as a young area, delimited by **youngBase** and **youngLimit**; the rest is considered old space. The young area consists of an *eden* space an two smaller *from* and *to* spaces. Within the old area, there coexist spaces for pinned objects, large objects (which are also pinned), image segments and an allocation zone for tenured objects, that is managed by the collector described in section 5.5.

The old allocation zone is split in equally sized chunks that can be freely committed and decommitted as objects are allocated, moved and released.

5.3.2 Object Allocation

Objects are created using bump pointer allocation as shown in Listing 5.1. The fast path of this contiguous allocator bumps the **nextFree** pointer of the eden space and returns, if the obtained buffer is within space limits.

When an allocation attempt fails, it may mean that the eden space is full or that the object being allocated is too big. In the latter case, the allocator creates a new large space to allocate just that object; in the former case it may trigger a young GC pass (if GC is enabled) and then retries using a slow path allocator. This second

Listing 5.1: The allocation mechanism tries to just bump the **nextFree** pointer of the **edenSpace**. If that fails, the slow path checks for large objects, triggers GC if needed and, finally, tries to allocate again.

```
Memory >> allocate: size
    | oop |
    oop := edenSpace allocateIfPossible: size.
    oop _isSmallInteger ifTrue: [^oop].
    size > LargeThreshold ifTrue: [^self allocateLarge: size].
    CRITICAL ifFalse: [self collectIfTime].
    ^self allocateCommitting: size

Memory >> allocateCommitting: size
    | oop |
    oop := allocator allocateIfPossible: size.
    oop _isSmallInteger ifTrue: [^oop].
    ^oldZone allocate: size
```

allocator as the least resource commits more memory from the old allocation zone.

5.4 Generational Garbage Collector

For newly created objects, Bee/LMR implements a copying generational scavenger [Che70, Ung84, Ung86]. New objects are created in the *eden* space. When an allocation attempt is made and there is no space left, a minor collection is done. Survivors from *eden* and *from* are moved to the *to* space, which is swapped with *from* when collection finishes. Objects that survive a second minor collection, passing from *from* to *to* are marked so that they get *tenured* if they survive another minor collection.

After collection finishes, the memory looks at the survival statistics and adjusts the size of the *eden* as shown in Listing 5.2. The adjustment heuristic tries to keep a small eden size as long as the survival rate is kept low (less than 8%). A smaller eden is preferred because increases the chances of keeping the memory used for allocation in the processor's cache. An increase in survival rate is a hint that the generational hypothesis is not holding. For that reason, when it crosses the 8%

Listing 5.2: Adjustment of *eden* size according to survival rate.

```
Memory >> adjustSpaces
    | stats rate committed limit delta |
    stats := meter runs last.
    rate := stats survivalRate.
    committed := edenSpace commitedSize.
    limit := edenSpace commitedLimit.
    stats youngSize // 2 < (0.8 * committed) ifTrue: [^self].
    rate > 0.08
        ifTrue: [
            delta := committed * (rate * 4 + 1).
            edenSpace commitIfPossible: delta asInteger]
        ifFalse: [
            delta := committed * (0.8 - rate / 2).
            edenSpace decommitIfPossible: delta asInteger]
```

the policy quickly increases the size of the eden space to give objects more time between minor collections, increasing the chances of lowering the survival rate again.

5.5 Old-Space Garbage Collector

For the old-space heap, Bee/LMR implements a region-based one-pass opportunistic-only evacuating GC, in the tradition of *garbage-first* garbage collectors G1 [DFHP04, ZB20] and Immix [BM08].

Like G1 and Immix, in Bee/LMR the old space heap is divided into multiple equally sized regions. The default region size in Bee/LMR is 256kb and the number of regions grows as the process memory consumption goes up. To defragment the heap, the G1 algorithm includes a compaction step. After the initial marking phase is complete, it selects a group of regions to evacuate based on their occupancy rate. Live objects stored in regions with smaller occupancy are moved to different regions where they get compactly allocated one after the other.

5.5.1 Compaction Mechanisms

To perform compaction, the collector needs to both move the objects from the evacuated regions and also to update the references to the moved objects from referencing objects. The pointers to be updated could be anywhere in the heap, so a naive updating mechanism would require a full sweep through the whole memory, which would be inefficient.

Currently, there exist multiple algorithms that help to optimize this case. For example, to allow for efficient pointer updating, G1 maintains an individual remembered set for each region's incoming pointers. Therefore, to update the references to the evacuated objects it only has to traverse the incoming pointers list of the remembered set corresponding to the evacuated spaces.

Immix GC avoids fragmentation differently, using two mechanisms. On one hand, it maintains the occupancy rate of each block, and when under some predefined threshold it marks the block as *recyclable*. After marking, it sweeps through all recyclable blocks to identify their free regions. As doing so on a word per word scheme would be too costly, instead it subdivides blocks into lines, and maintains mark bits for each line, so that block sweeping can be done quickly. On the other hand, Immix avoids fragmentation by applying opportunistic defragmentation. *Before* starting the marking phase, Immix selects blocks to evacuate. As the blocks are preselected before the marking starts, the tracing algorithm can determine whether an object is moving or not at the time it finds a reference from another object. This allows for opportunistically updating the references as the object heap is being traced.

Bee/LMR old-space heap GC implements a garbage-first approach with opportunistic defragmentation only, the second Immix mechanism. Before tracing the object heap, the GC selects the regions to be defragmented based on the occupancy rate computed in the previous GC. During tracing, it updates the pointers to evacuated objects at the same time as it does the evacuation itself.

For simplicity and run time performance, there is no separate sweeping of lines and recycling of fragmented spaces; the spaces are either evacuated and released, or not released at all. The cost of this is an increment on internal fragmentation and floating garbage: space occupied by objects that are found to be unreachable is not

reclaimed until a successive GC cycle where the region occupancy goes below a predetermined ratio.

The forwarding pointers of the old-space heap GC are stored at a constant offset from the object being forwarded, in an address space reserved just after the object region. This effectively halves the addressing space available to the process, which is not a problem because Bee/LMR is implemented for 64-bit addressing spaces.

5.6 Related Work

The aim of our research in Bee/LMR was not to create new GC algorithms but to be able to apply standard GC techniques, with modifications if needed, to implement a working system.

The problem of garbage collecting self-hosted environments has been studied before. One of the main differences between Bee/LMR and other systems is the total unification of the aplication and execution runtime layers, including the VM code and the object heaps, although this is not always the case, as shown next.

Squeak [IKM+97] and PyPy [RP06] are examples of self-hosted implementations that do not unify the application heap with the VM, as the code of the latter is translated to C. With them, the VM manages the object heap, while the C runtime library manages the heap used by the VM, which basically uses **malloc**.

Truffle/Graal [WW12, WWW+13b], another self-hosted VM, does not implement a GC on its own. Instead, it relies on the one of the VM it runs over, HotSpot, which is written at the lower C++ level.

Other Java VMs like Maxine [WHVDV+13] and Jikes [AAB+05], while they do split the aplication layer from the execution runtime to the programmer, they internally model VM concepts with objects and implement a GC that traverses those objects, which from the point of view of the GC are indistinguishable. However, as the VM code cannot be modified at run time, and besides the code has type annotations, the objects representing parts of the VM like the GC itself can be determined automatically ahead of time, fixed in memory and do not require moving.

Tachyon [CBLFD11] and Klein [USA05] do not provide a GC.

Chapter 6

Metaphysics

A Framework for Working Remotely with Potentially Broken Objects and Runtimes

A main goal while designing LMRs was to make it practical to develop VMs by, instead of manipulating files, running the system and modifying its methods, classes and, more generally, objects. As manipulating kernel objects can cause fatal errors, we experimented with adapting live programming tools to make them safer for the development of core LMR components. Furthermore, we made them robust so that they could work on crashed LMRs or allow debugging systems that were not fully working.

An additional feature to the original Smalltalk IDE was implemented for the Bee/LMR project: an out-of-process debugger. This tool lets developers inspect the state of a frozen Bee/LMR process, to debug crashes and other failures. This tool reuses the GUI components of the rest of the development environment, however it required implementing a framework, called Metaphysics [PM17], that brings access to objects and memory in the remote process.

With Metaphysics it became possible to create native code debugging and profiling tools. These new tools make full use of the metacircularity of Bee/LMR and enable a dynamic, fast-paced *edit-test* workflow like the one we are used to when developing application-level code.

6.1 Context

In LMRs, vital components of the system are exposed to the programmer via the normal inspectors, debuggers and code browsers, which run within the same environment. Those components can be changed quickly, giving instant feedback. However, while changing core components, programmers are frequently faced with the problem that a small incorrect modification leads to the crash of the entire system leaving high-level tools unresponsive.

In the first stages of Bee/LMR's development, we used standard low-level tools such as GDB and IDA Pro for debugging such situations. While these tools are customizable and could be made to work with the meta information provided by Bee, they do not provide the fluent interactivity expected by Smalltalk programmers. Furthermore, dealing with metadata from these tools requires writing scripts that duplicate the logic already present for tools inside the running system.

To avoid these issues, we created a new set of tools that enrich the programming environment. Unlike the set of existing Smalltalk tools such as inspectors, browsers and debugger, the new tools are designed to communicate to a remote Smalltalk image running in a different process, allowing developers to work with objects in a remote environment which may be frozen for debugging. The process might have been paused by the OS because it executed an invalid operation, or by the developer for debugging purposes. In any case, the process is still alive, its memory can be read or written, and it may even be possible to change it and to let it continue executing.

For these new tools we developed the *Metaphysics* framework, inspired by ideas of [BU04]. This framework allows access to remote objects by providing mirrors for structural reflection and proxies for behavioral intercession, making possible to adapt message sending, i.e., method invocation semantics, based on the specific situation.

6.2 Uses

This framework was designed to deal with a *target* system that potentially was not working. This could be either a process that was paused by the operating system

or a memory dump of a program that crashed. Either way, no execution is taking place in the target. Operating systems provide different APIs that allow to read and write the target's memory, registers, processor flags, etc.

The LMR programmers access the target from a fully-working *local* IDE. This local environment is frequently very similar to the target one, but it does not need to be identical. There might be different classes loaded in each system, or their methods of the same classes could differ. The framework allows completing lacking information of the target system, like for example the shape of a class, with information that the programmer supposes to be equivalent in the local system, which they can fully access. The tools are designed to allow working in an environment were objects of the target system can be inspected with the same freedom as local ones.

The contexts in which LMR programmers use these tools can vary greatly and arbitrarily, so there does not seem to exist a single approach that matches all situations: sometimes they need to just access structures, sometimes to execute code remotely, sometimes to simulate what a remote method activation would do. The requirements may even not be defined beforehand, they might need to change from one kind of execution semantics to other dynamically as their exploratory tasks evolve.

The reminder of this section gives relevant examples that we came across while working on Bee/LMR.

6.2.1 Remote Object Discovery

A system gets paused and we want to obtain information about its most important objects: their addresses, the present classes, globals, symbols, etc. We have to discover where objects are and we do not have yet access to the symbols of the system, so it may not be possible to send messages to objects.

In the memory of the remote system we can find all metainformation we need, like the dictionary of globals, from where to start fetching well known objects such as **true**, **false** and **nil**, the class **Object**, its metaclass, the class **CompiledMethod**, etc.

In this case, there is little need for extra debugging information, and even if it

existed it would be hard to keep it up to date with moving objects in the heap.[1] Given the address of an object it is possible through meta information to access its fields and internal structure. This situation is a good candidate to be solved with mirrors [BU04], because it requires mainly structural reflection.

6.2.2 Remote Code Execution in the Original Process

When profiling a system, we want to gather information about the execution trace by looking at the compiled methods in the stack. The situation is in principle similar to the previous example, however it poses some new difficulties. While traversing a Smalltalk stack to gather a call trace can be easily done with mirrors, getting information such as the source code of compiled methods is a more complex operation in Bee/LMR, because Bee's mirrors do not support it directly.

The objects that represent compiled methods typically do not reference their source code directly, because it usually resides in secondary memory. Instead, methods use descriptors that allow locating the code in source files when access is needed. These descriptors are part of the internal structure of methods, which require implementing certain programming logic.

Accessing the internal structure of a compiled method to find its source code directly would be a bad practice, because it means violating encapsulation. In general, objects are designed in ways that decouple the external view from the internal structure. In practice, this means when looking at an object from the outside, it can appear very different from its actual internal representation. Most of the time, the provided view is more high-level and we would prefer to work with it through an object's external interface, and only go into the internals when it is really necessary.

Back to our example of fetching the sources of a compiled method, a method such as **#sourceCode** encapsulates all this process, which can be complex. We want to just execute it in the target system and get the result.

[1] In case the process was unfreezed later on.

6.2.3 Simulated Local Evaluation of Remote Code

In certain situations, it is necessary to completely pause a VM for debugging, keeping it alive in memory and attaching externally to the process. This can happen because something failed or because we want to analyze one of its components in detail.

We might want to inspect objects of such a system, understand the execution context, send messages to objects, and perhaps make changes to recover from a crash. However, if there has been a low-level failure, the objects we are accessing might have been damaged, which means that their own memory or the one that determines their behavior (class, methods, etc) could be corrupted.

Let us consider again the example of fetching the sources of a compiled method, this time in a crashed system. In that case, we may not be able to remotely execute code of **#sourceCode** because that would modify state of the runtime and make debugging more difficult. Instead, we could use an alternative approach: we could take the remote code of **#sourceCode** and simulate its execution on the target system by interpreting it locally.

6.3 Design of the Metaphysics Framework

This section discusses how the Metaphysics framework solves the problems identified in the previous section for the Bee/LMR.

Our design is explained taking into consideration how it evolved with the increasing needs as Bee/LMR was developed.

6.3.1 Base Metaphysics Concepts

The base of the framework consists of a reification of a remote Smalltalk runtime and its objects. It was implemented using the underlying OS API for external process debugging, which enables communication with the remote objects mainly by reading and writing the target process memory. The main design concepts of the Metaphysics framework are depicted in Figure 6.1.

A **Handle** represents an entity living inside a given **runtime**. There are two kinds of handles: **ObjectHandle**s for referring to typical objects in the runtime,

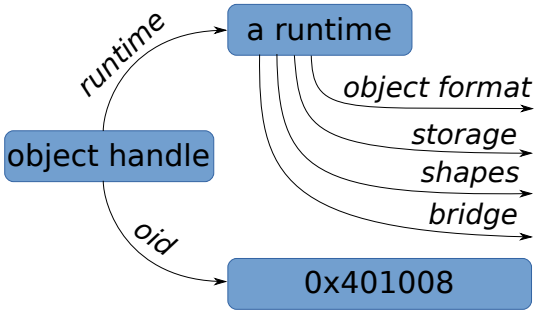


Figure 6.1: Object handles are opaque references to objects in a remote runtime. They are able to answer querys about the objects they point to only by delegating them to their runtime.

and **FrameHandle**s to refer to stack frames. All handles know the **runtime** they are associated with, and specific subclasses contain different slots that allow to individualize and access the entities they point to, be them objects or stack frames, in the associated runtime. Runtimes agglomerate a set of objects that work as a configuration:

- A **Storage**, that abstracts the OS API to read and write from the target process memory.
- An **ObjectFormat**, that understands and is able to read and write remote object headers.
- A **Metaspecies**, that knows the shape of classes in the remote system, allowing to read and write slots of objects in the target system without using their remote metadescription.[2]
- A **Bridge**, that is able to locate and deliver handles for global objects in the remote system.

6.3.2 Mirrors

Structural reflection on the objects of the remote runtime is achieved with mirrors. A **Mirror** in the Metaphysics framework is no more than a container of a **Handle**.

[2]Metaspecies are needed at initial stages, to provide early access to the metalevel of objects and for raw access to object slots using mirrors.

Listing 6.1: Use and implementation of augmented semantics.

```
ClassMirror >> #name
| name |
name := self getInstVarNamed: #name.
^name asStringMirror

StringMirror >> #asLocalString
^handle asLocalString

ObjectHandle >> #asLocalString
^runtime stringOf: oid

Runtime >> #stringOf: oid
| size |
size := objectFormat sizeOf: oid.
^storage stringAt: oid sized: size
```

Different subclasses provide an interface that allow to perform basic queries on the objects mirrored. An **ObjectMirror** allows accessing an object's header and slots, while it can also provide a mirror on the object's class.

On Bee, mirrors work as a layer that makes a distinction between remote objects and local ones. By default, methods that access object internals return new mirrors that contain the direct references pointed by the slots of objects.

Consider the following example: when a class mirror is sent the message **name**, the result is a mirror to the string stored in the **name** slot of that class. To do something meaningful with that mirror, it will normally be needed to get a local copy of that string, which is done through the method **asLocalString**.

The mirror model of our framework, in conjuntion with the design presented in section 6.3.1 is enough to solve the problem of remote object discovery stated in section 6.2.1.

6.3.3 Subjects, Gates and Execution Semantics

For more complex behavior, a model that allows varying message execution semantics was created. The experience we obtained in the situations described in

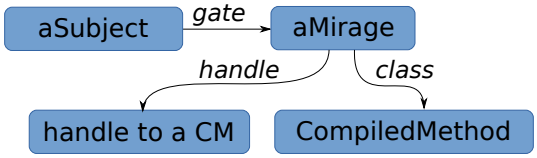


Figure 6.2: When **aSubject** receives a message, it delegates execution to **aMirage**, which in turn simulates the execution of the message. In this case the proxied object is a compiled method, so the mirage will create an interpreter for the local **CompiledMethod** class. The interpreter will perform lookup on that class and traverse the AST of the method found to generate a result.

Listing 6.2: Delegation of dispatch mechanism in subjects.

```
Subject >> #doesNotUnderstand: aMessage
        ^gate dispatch: aMessage
```

sections 6.2.2 and 6.2.3 showed that mirrors were only a first approximation to the dynamic environment we desire to work in.

Mirrors allow obtaining references to objects of the target system by reflecting on their internal structure. After that, given a reference to an object, we would like to be able to treat it as if it were a local object, but giving arbitrary semantics to the way in which messages it receives are executed. That requires the development of another model that implements the different possible execution semantics. For tackling those varying needs we modelled *subjects*.

A **Subject** is a proxy to an object in the remote system. It only understands the **#doesNotUnderstand:** message, which it overrides to delegate execution to *gates*. Gates of different types implement different execution semantics, working as a strategy pattern. Listing 6.2 shows the implementation of this mechanism.

The only slot of a subject is its **gate**, and in turn the gate points to the object handle. By implementing differently the **#dispatch:** method, different gates can produce arbitrarily different behavior on the subject when receiving a message. Separation of subjects and gates is done to allow keeping the subject interface clean, so that most messages fall into the **#doesNotUnderstand:** trampoline.

During Bee/LMR development we have identified 3 kinds of gates, which we can choose to use depending on the situation:

Trigger Gates cause execution of the message on the remote process, by modifying the process state, resuming the process and returning a result. They correspond directly to the semantics needed to tackle problems described in section 6.2.2

Direct Gates cause the local interpretation of the message, fetching the source code of methods from the remote object to which the message was sent.

Mirage Gates cause the local interpretation of the message, fetching the source code of methods from a local class that is equivalent to the one of the remote object.

Direct and Mirage gates correspond to the different kinds of execution semantics desired in section 6.2.3. They required the implementation of a source code interpreter. This interpreter takes as input an AST, a receiver and an array of arguments. It iterates the nodes of the tree, evaluating them and finally returning the result of the evaluation.

Figure 6.2 depicts the collaboration of different objects of the framework. Execution of messages locally for remote objects can give place to handles of local objects. For example, when a new object needs to be created during interpretation, it is instantiated locally by the interpreter, which returns back an object handle pointing to the new object. Gates do not need to know if the handles are local or remote, as both respect a same abstract API.

Changing the semantics of a subject is easy: the programmer only has to take the object handle from the gate and create a new subject with a new kind of associated gate. Messages with arguments are allowed, as long as each of them is also a subject.

6.4 Related Work

As mentioned, we built on the ideas of mirrors [BU04]. Similar work includes for instance Mirages [MVCTT07], which try to reconcile mirrors with behavioral intercession in AmbientTalk, an actor-based distributed OO language.

The general notion of remote debugging of OO environments has been studied, too. For example, the Maxine Inspector [Mat08], Jikes RDB [MH13] and Mercury [PBF+15] implement remote debugging using reflection via mirrors or other kind of middleware. While Mercury provides similar functionality to Metaphysics, its mechanisms differ widely.

On one hand, Mercury introduces a modified language, *MetaTalk*, where the meta-level is structurally decomposed (via stateful mirrors), and the target system has to run a modified VM that is able to exploit the language features; it provides an adaptable middleware, *Seamless*, to communicate both systems, a runtime-debugging support layer has to be embedded to the target environment, and reflection support is limited to components wrapped by mirrors.

On the other hand, Metaphysics uses the well-known Smalltalk-80 metamodel, and the target environment is run unmodified, without middleware or any special debugging support layer. Reflection on the system is based on the already existing reflective facilities of both the target and the host system. Metaphysics was thought for debugging remote processes where the communication to the remote environment is trusted and fast, like a remote process on the same machine.

Unlike Mercury, Metaphysics does not require a middleware layer because it connects directly to the target system by making use of its already included meta-information. Furthermore, in Metaphysics, a model of the debugged application is not needed in the developer machine, but can be lazily downloaded from the target system.

Other proposals include using holographic objects to deal with snapshots of crashed programs [SK13], and virtualization infrastructures for object-oriented high-level language runtimes [PDFB13, Pol15].

Chapter 7

Qualitative Evaluation

This section first discusses how LMRs improve upon state-of-the-art VMs to solve the problems identified in section 3.2. Then it details how they ease solving each of the case study problems stated in Chapter 3. Finally, it discusses the trade-offs LMRs introduce.

7.1 How LMRs Improve Upon the State-of-the-Art VMs

We argue for the removal of the strict two-layer architecture and the separation between VM and application. Relying merely on object-oriented techniques for structuring programs allows us to resolve the problems identified in section 3.2 as follows.

7.1.1 Limited VM Observability (PG1)

As previously discussed, the strict two-layer architecture purposefully restricts what can be observed in the VM. Though, for our case studies, it would be beneficial to more freely monitor not just the application but also the VM's behavior. With LMRs being *part of the application*, there is a more direct connection between application and VM code, which allows us to observe it in the same way as the application.

Programming systems implemented using LMRs also provide a more direct causal connections between application and VM code than traditional VMs, which can also be harnessed for instance to adapt more easily to unanticipated scenarios and helps us to improve applications in novel ways.

Unified Programming Language

One implicit benefit is that application developers do not necessarily have to learn and work with a different language. At least in the simplest realization of LMRs, the language used for the VM is the same as the one used for the application, with the small differences discussed in section 4.3, which are more a set of conventions that can be learned on the fly. Like with objects and metaobjects in a system like Smalltalk, in LMRs the is no differentiation between what is VM code and what is application code.

Unified Programming Tools

Another benefit is that application programmers do not need to learn a new sets of tools, because their regular tools for code browsing, profiling, inspecting values, and debugging, are the same they normally use. However, changing a running system comes of course with the added risk of making changes that result in crashes. In our experience, this will result in a more careful experimentation, but does not discourage it.

7.1.2 Separate VM Development Mode (PG2)

Since LMRs are essentially part of the application, they benefit from the same programming model as the application code, and support for Live Programming Environments. This also means there is no separate toolchain or build environment needed. Furthermore, all VM code is readily available to the developer since the system is downloaded, and there is no need to compile parts of the VM. Benefiting from the same tooling also means that code is automatically compiled as the user accepts changes to VM code in the same way as it is done for application code. Thus, the application development mode is also the VM development mode.

Incrementally Learnability

Application developers using LMRs can browse the code of the different parts of the VM and inspect the relevant pieces, as they do with the code of the large application codebase. Similarly, instead of merely imagining how the VM code behaves at run time, they can debug and observe it. For example, instead of imagining how or when the compiler triggers recompilation, it is possible to place a breakpoint in the compiler itself and debug the mechanism as it executes. Thus, it is possible to run the VM code step by step to understand how the pieces fit together, when and where it is needed.

7.1.3 Long edit-compile-run feedback loops for VM Components (PG3)

From the previous outlined benefits also follows that we have the same immediate feedback for VM code that we have for application code. This instant feedback reduces the time from ideas to experiments. In LMRs, the VM code is available in the development environment in exactly the same way as application code. Thus, there are no additional compilation toolchains for the VM and compiling the VM does not require any different actions from the ones used for the application. There is no need for the developer to perform differentiated actions to read and write the code of the VM. Changes to code are applied instantly, which allows for scripting-alike actions during VM development with the application running at the same time.

Unified Programming Interface Between VM and Application

Another benefit of removing the architectural distinction between VM and application is that the components that make the VM in an LMR are reusable as libraries within the application and vice versa. There is no need to devise a lower-level interface for passing objects to the VM from the application, nor an interface for using higher-level objects within the VM, since both sides use a unique uniform representation.

Furthermore, it is possible to perform queries on the state of the system lively. For example, to know how much time is being spent in GC, there is no need to

activate a debug mode in the VM, to export GC statistics into data files, or to parse it with other tools to finally analyze it. Instead, the developer can directly access the garbage collector objects, write a few lines of code in the application language to collect and analyze the live system data and, with a single press of a key inspect the result of that analysis.

7.2 LMR-based Solution Approaches for the Case Studies

Based on the case studies described in section 3.1, we now show how Bee/LMR enables us to solve the problems and overcomes the limitations identified in section 3.2.

7.2.1 Garbage-Collection Tuning (GCT)

Our first case study identified performance issues related to garbage collection. Specifically, opening the application from scratch can take over a minute and a significant amount of time is spent on allocating objects and garbage collection.

The first step is to understand where exactly time is spent. Since the GC is implemented as a library, it can be identified with the standard profiling tools together with the application code in context, which is often not possible in the same way on other VMs, because they try to make GC transparent and unobservable and tools present it separately.

Because the GC is profiled the same as application code, the developer can see from a profile how much time was spent on allocating objects, how many garbage collections were triggered, and which parts of the GC, if any, are an area of concern.

Once an area is identified as a performance bottleneck, we would want to collect more specific statistics from the GC to better understand why it spends its time there. Figure 7.1 shows the tools used during the case study, which are already known to the application programmer.

Since the GC objects are directly available to the programmer, it becomes possible to directly modify them, while the system is running, to collect statistics such as the heap sizes over time, object survival and tenuring rates, as well as the

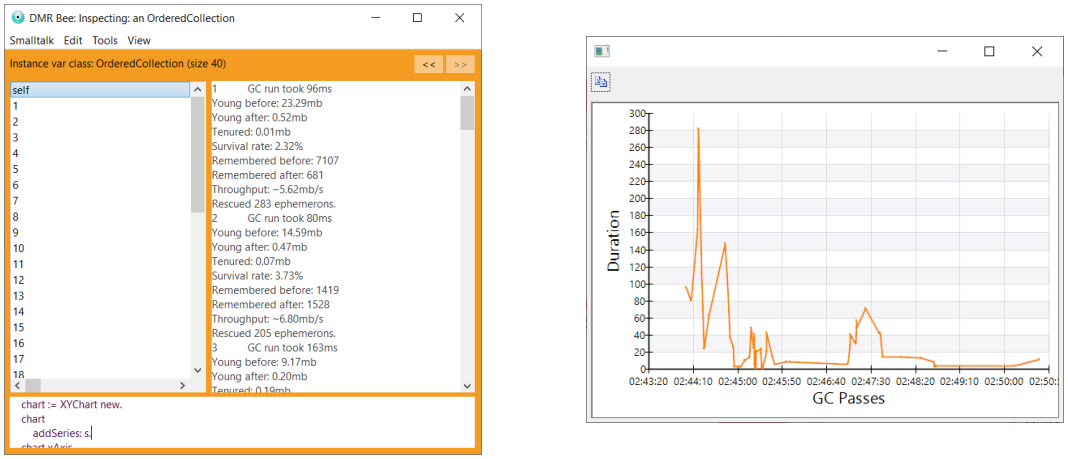


Figure 7.1: Debugging a garbage collector with standard application development tools. On the left, an inspector displays the garbage collection statistics of the live system, while on the right, a plot illustrates the time taken by the garbage collection process based on those statistics.

remembered set sizes.

For instance, we can add an instance variable to the class **GenGCPass**, which is in charge of collect statistics of the generational GC pass, to store the start time of each GC pass, and then modify the GC code so that it effectively stores that value when GC is triggered. In our case study, the developer used the obtained data to experiment with different Generational GC heap sizes and triggering heuristics while the system was running where it was possible to modify those things with the system running and to observe and measure the changes in real time, having immediate feedback and data to improve the application startup time. With this tuning, the application startup time was reduced by about 25%.

7.2.2 Recurring Recompilations by the JIT Compiler (JITC)

Similar to our GC tuning case study, profiling our application on Bee/LMR also helps identifying performance issues with the JIT compiler.

Since with an LMR the JIT compiler is merely another library, new solutions become possible. Our application developer can use the profiler to navigate to the JIT compiler's code, and identify why it takes too much time.

In this case, the developer notices that the VM is repeatedly invalidating and recompiling code, because for the same class, it sees two different methods: the

Listing 7.1: The method **NativizationEnvironment » nativeCodeFor: aBytecodeMethod** obtains the native code needed to execute the method passed as argument. Adding the cache, adds support for instance-specific methods in the JIT compiler, preventing repeated recompilations.

```
NativizationEnvironment >> nativeCodeFor: aBytecodeMethod
    cache at: aBytecodeMethod ifPresent: [:cached | ^cached].
    nativeCode := methodNativizer nativeCodeFor: aBytecodeMethod.
    (self shouldCache: aBytecodeMethod)
        ifTrue: [cache at: aBytecodeMethod put: nativeCode].
    ^result.
```

original one and the instance method added to an individual object. To provide a quick solution for the customer that found the problem in production, they try to modify the VM to enable the JIT compiler to cache the compiled code of both methods. Instead of triggering a recompilation when seeing an instance-specific method, they add a lookup in the compiled-code cache and compilation is only triggered if there is no matching cached entry.

The adapted code in Listing 7.1 implements this simple trick in the JIT compiler. While this method belongs to "VM code", it is a simple Smalltalk method and can be changed as any other method in the live programming environment. Furthermore, the developer does not need any specific knowledge of how the compilation works. They only needed to add the check of a dictionary and the caching of the native code in it. The tools, the language, and the concepts used are well known to the developer. Figure 7.2 shows a debugger halted at the point of assembling a bytecode. The liveness of the system allows developers to see the results instantly, without restarting the system, even coding in the debugger. Thus, this fix could be delivered to clients without requiring them to restart their application.

To summarize, when developers use an LMR to profile their applications, they have additional details on the operation of the JIT compiler. If the JIT compiler causes performance issues, it appears as any other part of the application, making it possible to identify opportunities for improvement. Thus, LMRs remove accidental complexity barriers and invite the developers to solve the problems more directly,

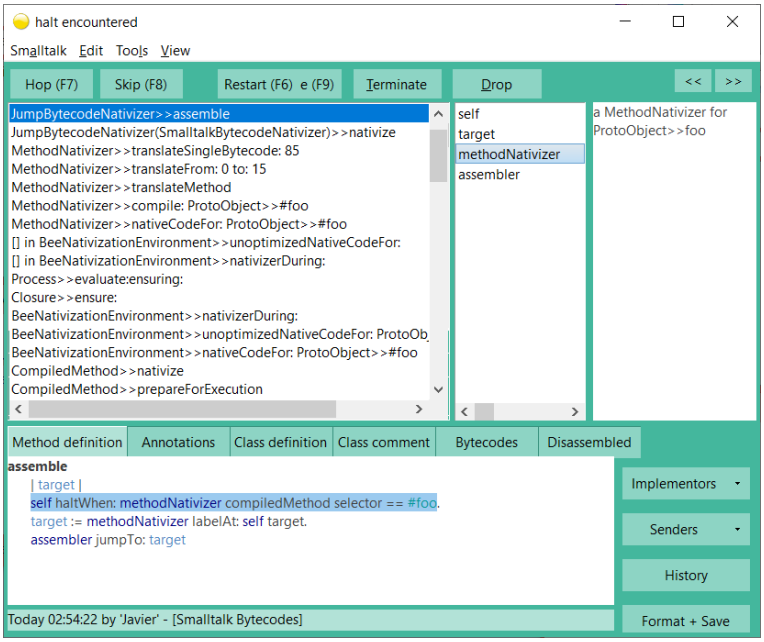


Figure 7.2: A debugger displaying the template JIT-compiler assembling a jump

with instant feedback without having to restart the system. Those optimizations can be delivered to clients without even requiring them to perform an application restart.

7.2.3 SIMD Optimizations (CompO)

The third case study aimed to add support for vectorized operations to speed up the floating-point arithmetics that are used in the simulation code of the application. Bee/LMR enables us to harness the compilation tool-chain of the VM for unanticipated scenarios. In particular, it is possible to adapt the compiler to optimize the relevant code for us, which is not normally an option with state-of-the-art VMs.

Since the compiler optimization can be application-specific, it is possible to implement the bare minimum support for the concrete application instead of building a generic system that is suitable for a wide range of code structures. Not only does this give immediate benefits, but it also simplifies the problem significantly and allows a developer to incrementally support code patterns that are judged important in the specific application.

Listing 7.2 shows two implementations of element-wise addition of two float arrays. In this implementation, elements are added one by one. A SIMD-optimized

version is shown in Listing 7.3. The programmer has to implement a compiler node that gets assembled to the desired **addps** instruction. The assembly can be debugged with a conventional Smalltalk debugger that includes native code information. As in the previous case studies, there is no need to perform tasks unrelated with the problem being solved, such as VM recompilation. The implementation of the compiler support to generate assembly code for the new node type is done while the system is running, without restarting, and can be changed and debugged as much as needed.

7.3 Discussion

By removing the strict separation between applications and VMs, we can benefit from more insight into the execution by utilizing our standard tools. Changing the VM and runtime libraries becomes as direct and immediate as changing any application code. In the following, we will briefly discuss other benefits and drawbacks we found with this approach.

7.3.1 Additional Benefits

The flexibility LMRs bring provided us with a wider range of options than traditional VMs when tackling problem scenarios and unanticipated change requirements. Since LMRs reduce the length of the feedback cycle for VM changes by orders of magnitudes, experimentation and exploration takes much fewer time and effort. As noticed in the section 7.2.2 when caching results of the JIT compiler, such kind of experimentation, and numerous changes can be done without particularly deep knowledge of the garbage collector, JIT compiler, native code, or similar concepts that might be new to application developers. Often it is as simple as caching some result or collecting some data for getting a better understanding of what the system is doing.

For example, by looking at high-level native-code characteristics, one can learn about how the JIT compiler works and how optimization decisions of the JIT compiler affect the result. This let us observe the decisions made by the JIT compiler, analyze and modify them. For instance, we can detect methods being

Listing 7.2: The method **FloatArray » += aFloatArray** performs element-wise addition of the argument into the receiver. It invokes a naive implementation that performs one-by-one addition. **_floatPlus:**, **_floatAt:** and **_floatAt:put:** are metamessages that get evaluated at compile time and generate specialized compiler nodes.

```
FloatArray >> += aFloatArray
    (self checkAdditionArguments: aFloatArray)
      ifFalse: [
        ^self withIndexDo: [:f :idx |
          self atValid: idx put: f + aFloatArray]].
    self basicPlus: aFloatArray

FloatArray >> basicPlus: aFloatArray
    1 to: self size do: [:i | | a b |
        a := self _floatAt: i.
        b := aFloatArray _floatAt: i.
        self at: i put: (a _floatPlus: b)]

Node >> _floatAt: indexNode
    ^LoadNode
      base: self
      index: indexNode
      type: #Float32

Node >> _floatAt: indexNode put: valueNode
    ^StoreNode
      base: self
      index: anONode
      value: valueNode
      type: #Float32

Node >> _floatPlus: rightNode
    ^FloatPlusNode left: self right: rightNode.

X64CodeEmitter >> assembleFloatPlus: aFloatPlusNode
    left := allocation at: aFloatPlusNode left.
    right := allocation at: aFloatPlusNode right.
    self assemble: 'addss' with: left with: right
```

Listing 7.3: The SIMD version is quite similar, but it applies the operation to multiple data on each cycle. The number of iterations is divided by the amount of parallel additions, the type of stores and loads is adjusted to make compiler use appropriate SIMD registers and operand sizes. The assembly for the addition is changed to **addps**, a packed single-precision float addition.

```
FloatArray >> basicSimdPlus: aFloatArray
    1 to: self simdSize do: [:i | | a b |
        a := self _simdFloatAt: i.
        b := aFloatArray _simdFloatAt: i.
        self _simdFloatAt: i put: (a _simdFloatPlus: b)]

FloatArray >> simdSize
    ^self size // self floatsPerSimdRegister

Node >> _simdFloatAt: indexNode
    ^LoadNode
      base: self
      index: indexNode
      type: #SIMDFloat32

Node >> _simdFloatAt: indexNode put: valueNode
    ^StoreNode
      base: self
      index: anONode
      value: valueNode
      type: #SIMDFloat32

Node >> _simdFloatPlus: rightNode
    ^SIMDFloatPlusNode left: self right: rightNode

X64CodeEmitter >> assembleSIMDFloatPlus: aSIMDFloatPlusNode
    left := allocation at: aSIMDFloatPlusNode left.
    right := allocation at: aSIMDFloatPlusNode right.
    self assemble: 'addps' with: left with: right
```

inlined or not being inlined. Depending on the context too few or too much inlining can result in worse performance. Since it is a live system, one can then tune heuristics and see the results without having to restart the system.

7.3.2 Importance of Liveness in LMRs

From the architectural perspective, it is possible to build runtime system that remove the separation between VM and application, without supporting liveness. Those runtime systems will likely also reduce the knowledge gap, improve causal connection between runtime and code, and enable application developers to better understand the runtime system.

However, the instant feedback and observability of LMRs also turns the mere potential of these benefits into serendipitous encounters that will happen, which is key for the understanding of large codebases. Just because one may have access to the code, does not mean that one will look at it. In an LMR however, the code will be right there in the debugger or profiler one is using to diagnose a problem, and one can instantly experiment with it without any extra action to be taken. We believe this to be a qualitatively different situation than in classic environments.

7.3.3 Drawbacks and Concerns Associated with LMRs

Removing the distinction between VM and application comes naturally with concerns about abandoning the benefits of this architecture. We now briefly discuss these concerns and others such as software safety.

Maintainability and Portability

As previously discussed, the two-layer design is chosen by many VMs on purpose to ensure a strong separation of VMs and application. With VMs such as the Java Virtual Machine, this gives applications the opportunity to change between JVMs being confident that the application will continue to work. With LMRs, this strict separation and guarantee is not given anymore. However, we would argue that in some situations this is a trade-off worth making.

Considering that many large applications would use the technique of *vendoring* frameworks they rely on into their own codebase, an LMR is essentially the practice

of vendoring the runtime library into the application. Vendoring is typically done to gain more stability and be able to fully control a framework or library. On the flip side, this comes at the cost of maintaining this fork and upstreaming changes into the original. Arguably, the same is true for LMRs and as with vendored frameworks, it gives all the flexibility and cost of doing so. However, one may always decide not to change anything, which will typically make it relatively simple to stay current with any changes made upstream.

Safety

When creating a virtual machine (VM) for a programming language, software safety is a crucial concern. We argue that moving the code between architectural layers does change the situation only superficially and any concrete threats are specific to the programming language of the LMR. As such, there is no general answer and whether an LMR introduces new attack surfaces depends on the language in question.

For Bee/LMR, the situation is not ideal from the start. As a Smalltalk, there are many opportunities to attack the system by evaluating arbitrary code at run time or by using the `#become:` operation to arbitrarily swap objects with each other. Languages such as Java and JavaScript typically have mechanisms to restrict what can be changed.

For instance, Java's module system is also able to prevent reflective accesses. JavaScript has the `freeze()` method that allows it to make objects immutable. Such mechanisms could in theory be used to provide some form of protection. However, having a JIT compiler, its assembler accessible, and the ability to access arbitrary memory in theory allows the execution of arbitrary code and as such would require careful design to prevent an attacker to gain access to this capability. Type and capability systems, for instance using mirrors [BU04], could facilitate a suitable system design.

Stability

From a practical standpoint, developers have flexibility in choosing their preferred workflow. Changing the system while it is running is risky, as even a minor mistake

can cause the system to crash. In Bee/LMR, there are no special crash avoidance mechanisms than the ones given by the language itself. That includes safe indexing of arrays and mechanisms such as **doesNotUnderstand**, which let programmers catch and fix errors as they show up.

It is possible and common practice to save the image right before applying such changes. Additionally, if a crash does occur, a remote debugger pops up, allowing to inspect the frozen image before terminating the process, to aid understanding the cause of the crash. Alternatively, since Bee/LMR can be edited as code offline, one can also modify it and then bootstrap it, which allows applying more complex changes that might not be safe in a running system.

In practice, developers often start with the live workflow and switch to safer approaches once they found the boundaries of what is possible in a live system. The same holds for designing the overall system. Often we find ourselves to chose more conservative design changes, which facilitates live updates and reduces the risk of stability issues.

7.3.4 Metamodel Dynamicity and System Scalability

Our LMR implementation does not expand on the metaobject protocol to allow changes such as modifying object layout format on-the-fly. While in principle, one can treat all objects, or rather the programs memory as bits, and run a script over them to do such updates, Bee does not provide any special support or API to make such changes convenient. Small changes like modifying the meaning of a free bit in object headers may be explored while running the program. Bigger changes, on the other hand, usually get done by modifying source files and re-bootstrapping the system.

When considering the scalability of an LMR, one may be concerned about the support of large object heaps as well as the support for large codebases. Since we use the G1 GC design [DFHP04], we have not noticed any scalability issues. G1 is designed for large heaps and to give soft real-time guarantees on garbage collection pauses.

When it comes to large codebases, the Smalltalk approach of an image-based system, that contains the code, scales quite naturally. Since code is compiled at run

time and development time one method at a time and on-demand, the growing codebase has not been a notable concern, even with our 1.1 million lines of code application.

7.3.5 Real-life Usage

Usually, application developers do not try to optimize an LMR's GC or JIT directly, but use the richer information obtained from the live system to modify their application, so that they can make better design decisions within their code to avoid hitting VM bottlenecks.

Application developers using LMRs count with the tools they use daily in their LPE to inspect, browse and debug the GC code if they want to. Furthermore, if they do want to modify components like the JIT and the GC, they can do it with the system running, with instant feedback.

At development time, even for application developers, it can be handy to try little changes to the VM, make small improvements and run little experiments, even if small mistakes might make the system might crash during the development session. These kinds of experiments can result in changes that, if desired, can then be submitted to VM experts to evaluate their safety and inclusion upstream.

7.4 Additional case studies

While not part of the core case studies (GCT, JITC, CompO), here we describe other scenarios where the use of an LMR showed practical benefits in contrast to the two-layer VM/application design.

7.4.1 Memory Leaks Detection

The presence of a live system allowed for novel uses of the memory management runtime that are typically not practical with static VMs. The GC can be modified lively and harnessed in unanticipated ways.

In Bee/LMR, we harnessed the GC to find out memory leaks in an HTTP request server. Bee includes a generational GC made of an **eden** space and two flipping **from** and **to** areas where objects go before moving to **old** area. Because the

GC is integrated as a library, it is possible to track live objects in particular spaces, so finding leaked objects can be done by the following actions:

1. Before handling a request, trigger a generational GC to clean **eden** space.

2. Disable GC during the handling of the request, so that all objects get allocated in the **eden**, making **eden** grow if necessary during the request.

3. After the request gets handled, trigger a generational GC so that live **eden** objects get moved to the **from** space.

4. Using the GC API, iterate through the objects in the **from** space adding them to a weak collection. The contents of this collection is a superset of the leaked objects. It may still retain objects that were pointed from remembered set but not really alive.

5. Run a full GC to nil out the entries of the collection that are unreachable when tracing the full object graph. The contents of this set is now the set of leaked objects.

The code for performing these actions fits in one method, and was added to the memory manager and tested with immediate feedback. The API receives a block closure as argument and returns the newly created objects that survived the evaluation of that block. The code is shown in Listing 7.4.

As in other situations, this API is mostly useful at development time rather than at deployment time. In the traditional VM scenario, implementing this would have required passing through all the barriers detailed in Section 3.2, implementing the change in the VM, recompiling it restarting the application with the patched VM and then trying the experiment. In the LMR, the experiment was done without delays. Actually, the first try saved the results into a standard not-weak collection, and let us discover that we needed an extra filtering of objects through full GC. In the traditional VM case, that would have required another recompilation and restart step that was not needed in the LMR.

Listing 7.4: **Memory»objectsSurviving: aClosure** method evaluates the closure passed as a parameter and returns the objects that were leaked. **HttpWorker»processRequest: anHttpRequest** uses that API to graphically show the result.

```
Memory >> objectsSurviving: aClosure
    | set finalizable |
    set := WeakIdentitySet new.
    self collectYoung; disableGC.
    aClosure value.
    self enableGC; collectYoung.
    fromSpace objectsDo: [:o | set add: o].
    self collect; collect.
    ^set

HttpWorker >> processRequest: anHttpRequest
    leaked := Smalltalk memory
        objectsSurviving: [ self doProcessRequest: anHttpRequest].
    leaked inspect.
```

Listing 7.5: **TestSuite » coverage** method clears the code cache, runs a test suite and finally counts how many of the methods in the system have been executed by checking how many have been assigned native code.

```
TestSuite >> coverage
    | methods executed |
    Smalltalk clearCodeCache.
    self run.
    methods := CompiledMethod allInstances.
    executed := methods count: #hasNativeCode.
    ^executed / methods size
```

7.4.2 Implementation of Code Coverage of Tests

Bee contains tools for analyzing code coverage of tests. These tools were based on instrumentation of compiled methods. While useful for most of the typical scenarios, the instrumentation approach was too slow to be executed on all the 15000 tests of the system that stress the 1.1 million lines of code of the application.

This problem was solved by accessing JIT-compiler information from the application, instead of using instrumentation: As Bee/LMR only executes code by JIT-compiling methods, if the JIT code cache is cleared before running tests, and the size of that cache is big enough, then it is possible to determine which methods have been executed by just checking whether they got added to the code cache. This information is readily available in the system, and can be collected with a script like shown in Listing 7.5.

The required change was implemented by application developers in the unit test library, and allowed them to obtain coverage statistics without paying performance penalties.

7.4.3 Other Optimizations

At some point our simulation application incorporated a **BitField** type that simplified working with bit fields. The type allowed extracting bits of an integer as if it were a C bit field, and also writing bits back. The usage is simple, the client creates a bit field with something like **flags := BitField bits: 4 to: 6** and then

Listing 7.6: a **BitsAt** node contains a pair of left and right sub-nodes. The left one can be anything, the right one is a constant pointing to a BitField object. The **optimized** method accesses the constant and generates optimized code (a bit and, followed by a bit shift).

```
BitsAt >> optimized
    | bitfield and shift |
    bitfield := right value.
    self assert: bitfield class == BitField.
    and := BitAnd left: left right: bitfield mask.
    shift := BitShift left: and right: bitfield shift.
    ^shift
```

they can extract and write the bits back with for example **flags bitsAt: field** and **flags bitsAt: field put: aValue**.

After some time, it was discovered that this simple abstraction was causing a small performance penalty. As the compiler could not prove that bit fields were immutable, the operations for *shifts* and *ands* were using generic code with fallback cases for non-integer types. Adding general immutability support to the compiler was out of the scope, but instead of removing the abstraction it was possible to hand tune Bee/LMR compiler to incorporate an ad-hoc optimization. With this optimization, when the compiler sees a **bitsAt:** message sent to a bit-field object, it assumes it is immutable and generates optimized code for the specific size of the bit field, removing all performance penalties.

The implementation of this optimization involved two steps: the first one was adding **bitsAt:** and **bitsAt:put:** to a list of special messages in the optimizing native-code compiler. When seeing such messages, the compiler converts the **MessageSend** node to a specialized **BitsAt** node, a subclass of **BinaryMath** nodes. The second step was the optimization itself. For that, the **BitsAt** node implements the **optimized** method, that in turn converts itself into a series of *shift* and *and* nodes, as shown in Listing 7.6.

As the optimization allowed zero-cost bit fields, this abstraction was then incorporated back into the rest of the runtime code, simplifying the accesses to bit fields stored in compiled methods and classes.

Chapter 8

Quantitative Evaluation

To assure the viability of our approach, we run a series of performance tests that provide a lower bound to the performance of a LMR-based systems. Practical performance is not a problem for LMRs. Bee/LMR is used daily by a team of four developers and is deployed to run our product in production. It receives two major releases per year, with monthly minor releases for specific customers.

8.1 Performance Evaluation

The goal of this evaluation is to show that LMRs can reach the performance levels of traditional VM-based systems. We compare Bee/LMR against the following systems:

Pharo/Cog Pharo is a Smalltalk dialect which runs on top of OpenSmalltalk VM, the most widely used Smalltalk VM. We run OpenSmalltalk run in dual JIT/interpreter mode (Cog) [Mir11]. OpenSmalltalk is written in Slang, transpiled to C and compiled with a standard C compiler.

Python/CPython The standard Python VM (v3.10.7) which is written in C and uses an interpreter.

Ruby/MRI The standard Ruby VM (v3.0.4p208), also written in C.

JavaScript/V8 The JavaScript engine behind Node.js (v18.16.0), written in C++, that includes an interpreter and several optimizing JIT-compiled stages.

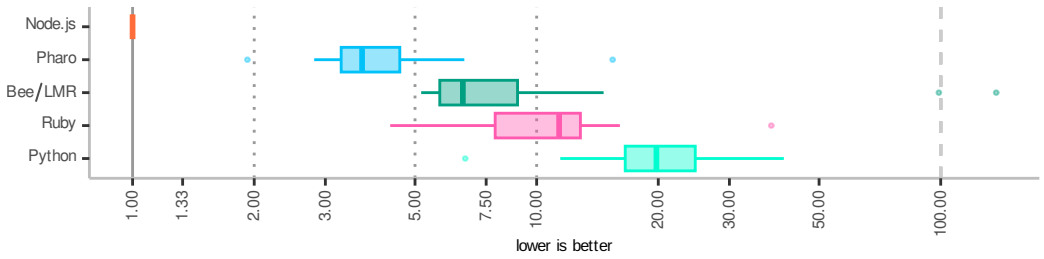


Figure 8.1: Bee/LMR benchmark times compared to other dynamic object-oriented systems.

We use the *Are We Fast Yet* benchmarks which include 9 micro and 5 macro benchmarks [MDM16]. They are designed to compare performance across different language implementations and thus were easy to adapt to Bee.

The benchmarks were run on a machine with a 2.8Ghz 4-core Core i7 7700HQ with hyper-threading and 16 GB of memory. The operating system is a 64-bit Ubuntu 22.10. Bee is run through Wine compatibility layer, as it only supports Windows. All the implementations are 64-bits.

We measure 100 iterations of each benchmark, and in the case of Node.js 3000 to minimize noise from late JIT compilation. For each benchmark, we take the median of the measurements and report the summary as a standard boxplot in Figure 8.1. The results are normalized to Node.js, which is the fastest system.

Overall Bee/LMR is slightly slower than Pharo/Cog. This is expected, since the Cog VM has been optimized for a much longer period than Bee/LMR. For example, string copying in Bee is done byte-by-byte, checking bounds at each character. Bee is around one order of magnitude faster than Ruby/MRI and Python/CPython interpreters. SinceBee/LMR uses a template-based JIT compiler [Ayc03], there is a lot of room for improving its performance with classic compiler optimizations, when comparing with Node.js and Pharo.

However, since CPython and MRI are classic bytecode interpreters without JIT compilation, Bee/LMR outperforms them roughly at the level one would expect from a template-based compiler. Python is known to be one of the slower interpreters, which newer versions starting with Python 3.11 aim to fix.[1]

[1] https://docs.python.org/3.11/whatsnew/3.11.html#faster-cpython

8.2 Runtime Implementation Size

We evaluate the size of Bee/LMR by counting the lines of code of different parts of the system associated with VM components: JIT compiler, GC and built-in functions. We compare that metric against the other systems to give an idea of the difference. Bee/LMR is significantly smaller, because it is much more specialized and does not support the same variety of operating systems, processor architectures, and usage scenarios.

Subsystem	**LoC**
GC	1,689
Baseline JIT	3,329
Optimizing Compiler	5,567
Intel Assembler	8,868
Built-ins	2,604
Total	**22,057**

(a) Lines of codes of Bee/LMR subsystems

VM	**LoC**	
V8 (JavaScript)	1,111,919	C++
CPython	245,538	C
MRI (Ruby)	210,120	C
OpenSmalltalk	124,741	Smalltalk
Bee/LMR	**22,057**	Smalltalk

(b) Lines of codes of different dynamic, object-oriented virtual machines, compared to Bee/LMR

Figure 8.2: Size of the LMR module in lines of code

Chapter 9

Conclusions and Future Work

We propose Live Metacircular Runtimes (LMRs), a runtime design that combines VM implementation and application, replacing the traditional architecture that separates them into layers. Our approach uses basic object-oriented techniques instead to ensure for instance encapsulation and to enable changes to the runtime systems at run time, the same as normal application code.

In this work, we use case studies on tuning the garbage collector, changing the just-in-time compiler to avoid unnecessary recompilations, and adding support for vector instructions to the compiler to argue for the benefits of removing the architectural separation to enable shorted feedback cycles and better understanding of the runtime system by application developers.

Bee/LMR is our implementation of Bee Smalltalk that runs on top of a live metacircular runtime, instead of a traditional VM. It is used in production to run a 1.1 million lines of code application. This LMR-based system has allowed application developers to incrementally understand VM components as needed, and even to modify them when required as argued with our case studies. Specifically, we show that this approach avoids the limited VM observability, the separate VM development mode, and the long edit-compile-run cycles for the traditional two-layer architecture.

9.1 Future Work

In future work, we want to explore how to improve performance without sacrificing the ability to change the runtime at run time. While compilers such as Graal demonstrate that compilers in high-level languages can produce state-of-the-art performance, our design of the object layout and garbage collector made careful decisions to preserve a working system at all times. There are similar circular dependencies in the just-in-time compiler, which means that a change at run time could break it. This could be mitigated by supporting multiple versions of such subsystems, where the old working version remains available as a fallback.

We also expect to work in a type system that performs inference, to both give type feedback to the JIT compiler and also to help to maintain code closures for the GC.

Other future research directions could explore how to apply this design to other languages, or build other languages on top of our LMR. The key research question here is what the appropriate trade-offs are between enabling live programming and achieving practical performance of such systems, without introducing unwarranted complexity.

It would also be possible to apply the infrastructure developed for Bee/LMR to implement a VM for other more static languages such as Java, inside the live programming environment.

Bibliography

[AAB+00] 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.

[AAB+05] 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. The jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[ASU20] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, principles, techniques, and tools.(Rep. with corrections.)*. Addison-Wesley Pub. Co., 2020.

[Ayc03] John Aycock. A Brief History of Just-In-Time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.

[BBZ11] Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. *Firm-a graph-based intermediate representation*. KIT, Fakultät für Informatik, 2011.

[BKL+08] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. Back to the future in one week—implementing

a smalltalk vm in pypy. In *Workshop on Self-sustaining Systems*, pages 123–139. Springer, 2008.

[BM08] Stephen M Blackburn and Kathryn S McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. *ACM SIGPLAN Notices*, 43(6):22–32, 2008.

[BSD+08] 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. The moxie jvm experience. *cluster computing*, 2008.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 331–344, 2004.

[CBLFD11] Maxime Chevalier-Boisvert, Erick Lavoie, Marc Feeley, and Bruno Dufour. Bootstrapping a self-hosted research virtual machine for javascript: An experience report. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 61–72. ACM, 2011.

[CGM16] Guido Chari, Diego Garbervetsky, and Stefan Marr. Building Efficient and Highly Run-time Adaptable Virtual Machines. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS'16, pages 60–71. ACM, 2016.

[CGMD15] Guido Chari, Diego Garbervetsky, Stefan Marr, and Stéphane Ducasse. Towards fully reflective environments. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 240–253. ACM, 2015.

[Che70] Chris J Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, pages 285–299. ACM, 1995.

[Cli93] Cliff Click. From quads to graphs: An intermediate representation's journey. Technical report, Citeseer, 1993.

[CPL83] Thomas J Conroy and Eduardo Pelegri-Llopart. An assessment of method-lookup caches for smalltalk-80 implementations. *Kra83*, 1983.

[CPVF18] Guido Chari, Javier Pimás, Jan Vitek, and Olivier Flückiger. Self-contained development environments. *ACM SIGPLAN Notices*, 53(8):76–87, 2018.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 49–70. ACM, 1989.

[DFHP04] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48, 2004.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302. ACM, 1984.

[DWS+13] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic

compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pages 1–10, 2013.

[FBC+09] Daniel Frampton, Stephen M Blackburn, Perry Cheng, Robin J Garner, David Grove, J Eliot B Moss, and Sergey I Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 81–90. ACM, 2009.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language: Everything You Need to Know*. " O'Reilly Media, Inc.", 2008.

[FQ03] Stephen J Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 241–252. IEEE, 2003.

[FY69] Robert R Fenichel and Jerome C Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38. Springer-Verlag, 1991.

[HU94] Urs Hölzle and David Ungar. 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, pages 326–336. ACM, 1994.

[HU96] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, July 1996.

[HWW+15] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing ast interpreters. *ACM SIGPLAN Notices*, 50(3):123–132, 2015.

[IBR+22] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for Productive Programming of Hardware Accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 703–718. ACM, June 2022.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 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, pages 318–326. ACM, 1997.

[Int15] Ecma International. *ECMAScript 2015 Language Specification*. Ecma International, Geneva, 6th edition, 2015.

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

[Jon79] HBM Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):26–30, 1979.

[KDRB91] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT, 1991.

[Kic96] Gregor Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, 13(1):8–11, January 1996.

[LKH15] Roland Leißa, Marcel Köster, and Sebastian Hack. A graph-based higher-order intermediate representation. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 202–212. IEEE, 2015.

[LYBB14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[Mat08] Bernd Mathiske. 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, pages 739–740. ACM, 2008.

[MDM16] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-Language Compiler Benchmarking—Are We Fast Yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS'16, pages 120–131. ACM, 2016.

[MH13] Dmitri Makarov and Matthias Hauswirth. 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, pages 169–172. ACM, 2013.

[Mir11] Eliot Miranda. The cog smalltalk virtual machine. In *VMIL'11: Proceedings of the 5th workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, 2011.

[Mor78] F Lockwood Morris. A time-and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–665, 1978.

[MVCTT07] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, and Eric Tanter. Mirages: Behavioral intercession in a mirror-based architecture. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 89–100. ACM, 2007.

[OUH+14] Atsushi Ohori, Katsuhiro Ueno, Kazunori Hoshi, Shinji Nozaki, Takashi Sato, Tasuku Makabe, and Yuki Ito. SML# in Industry: a Practical ERP System Development. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP'14. ACM, August 2014.

[PBAM17] Javier Pimás, Javier Burroni, Jean Baptiste Arnaud, and Stefan Marr. Garbage collection and efficiency in dynamic metacircular runtimes: an experience report. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, pages 39–50, 2017.

[PBF+15] Nick Papoulias, Noury Bouraqadi, Luc Fabresse, Stéphane Ducasse, and Marcus Denker. Mercury: Properties and design of a remote debugging solution using reflection. *The Journal of Object Technology*, 14(2):36, 2015.

[PBR14] Javier Pimás, Javier Burroni, and Gerardo Richarte. Design and implementation of bee smalltalk runtime. In *International Workshop on Smalltalk Technologies, IWST*, volume 14, page 24, 2014.

[PC19] Javier Pimás and Guido Chari. Powerlang: a vehicle for lively implementing programming languages. In *International Workshop on Smalltalk Technologies*, 2019.

[PDFB13] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, and Noury Bouraqadi. Virtual smalltalk images: Model and applications. In *21th International Smalltalk Conference-2013*, pages 11–26, 2013.

[PG92] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 116–127, 1992.

[PG07] Fernando Pérez and Brian E. Granger. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.

[Pim18] Javier Pimás. Migrating bee smalltalk to a different instruction set architecture: An experience report on porting a dynamic metacircular runtime from x86 to amd64. In *International Workshop on Smalltalk Technologies*, 2018.

[Pim22] Javier Pimás. Powerlangjs: A quick way to get your smalltalk to the web? In *FAST Workshop 2022 on Smalltalk Related Technologies*, 2022.

[PM17] Javier Pimás and Stefan Marr. Metaphysics: Towards a robust framework for remotely working with potentially broken objects and runtimes. In *2nd Workshop on Meta-Programming Techniques and Reflection*, 2017.

[PMG24] Javier Pimás, Stefan Marr, and Diego Garbervetsky. Live objects all the way down: Removing the barriers between applications and virtual machines. *arXiv preprint arXiv:1909.12795*, 2024.

[Pol15] Guillermo Polito. *Virtualization Support for Application Runtime Specialization and Extension*. PhD thesis, Universit e des Sciences et Technologies de Lille, 2015.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

[RP06] Armin Rigo and Samuele Pedroni. 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, pages 944–953. ACM, 2006.

[RRL+18] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming*, 3(1):1–33, July 2018.

[SC05] Doug Simon and Cristina Cifuentes. The squawk virtual machine: Java™ on the bare metal. In *Companion to the 20th annual ACM*

SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 150–151, 2005.

[SK13] Robin Salkeld and Gregor Kiczales. Interacting with dead objects. *ACM SIGPLAN Notices*, 48(10):203–216, 2013.

[Smi84] Brian Cantwell Smith. Reflection and Semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'84, pages 23–35. ACM, 1984.

[SW67] Herbert Schorr and William M Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, 1967.

[SWM14] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 165. ACM, 2014.

[Tan09] Éric Tanter. Reflection and Open Implementations. Technical report, DCC, University of Chile, Avenida Blanco Encalada 2120, Santiago, Chile, 2009.

[UBF+84] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson. Architecture of SOAR: Smalltalk on a RISC. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 188–197. Association for Computing Machinery, 1984.

[Ung83] David M Ungar. Berkeley smalltalk: Who knows where the time goes? *Smalltalk-80: bits of history, words of advice*, pages 189–206, 1983.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM Sigplan notices*, 19(5):157–167, 1984.

[Ung86] David M Ungar. The design and evaluation of a high performance smalltalk system. Technical report, CALIFORNIA UNIV BERKELEY GRADUATE DIV, 1986.

[USA05] David Ungar, Adam Spitz, and Alex Ausch. 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*, pages 11–20. ACM, 2005.

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

[WF10] Christian Wimmer and Michael Franz. Linear scan register allocation on ssa form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 170–179, 2010.

[WHVDV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, January 2013.

[WSH+19] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize Once, Start Fast: Application Initialization at Build Time. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, October 2019.

[WW12] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 13–14, 2012.

[WWS+12] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing

ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, pages 73–82, 2012.

[WWW+13a] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward!'13, pages 187–204. ACM, 2013.

[WWW+13b] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204. ACM, 2013.

[ZB20] Wenyu Zhao and Stephen M Blackburn. Deconstructing the garbage-first collector. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 15–29, 2020.