Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Objetos vivos a fondo: removiendo las barreras entre aplicaciones y máquinas virtuales

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

Objetos vivos a fondo: removiendo barreras entre aplicaciones y máquinas virtuales

Resumen Los lenguajes orientados a objetos suelen utilizar máquinas virtuales (VM) que proporcionan mecanismos como la compilación *just-in-time* (JIT) y la recolección de basura (GC).

Los componentes que brindan estos mecanismos de VM suelen implementarse en una capa separada, aislando así la aplicación. Si bien este enfoque aporta los beneficios de ingeniería de software de una clara separación y desacoplamiento, introduce barreras tanto para entender el comportamiento de la VM como para evolucionar su implementación, ya que debilita las conexiones causales entre las aplicaciones y la VM. Por ejemplo, la recolección de basura y el compilador JIT suelen quedar fijos en el momento de construcción de la VM, limitando la adaptación arbitraria en tiempo de ejecución. Además, debido a esta separación, la implementación de la VM no suele poder inspeccionarse y depurarse de la misma manera que el código de aplicación, estableciendo una distinción entre el código de aplicación fácil de trabajar y el código de VM difícil de trabajar.

Estas características representan una barrera que dificulta que lxs desarrolladores de aplicaciones comprendan el motor sobre el cual se ejecuta su propio código, y fomenta una brecha de conocimiento que les impide cambiar la VM.

Proponemos bibliotecas de entornos de ejecución metacirculares vivos o LMRs (por Live Metacircular Runtimes) para superar este problema. Las LMRs son motores de ejecución de lenguajes que integran perfectamente la VM en la aplicación en entornos de programación vivos. A diferencia de los enfoques metacirculares clásicos, proponemos eliminar por completo la separación entre aplicación y VM.

Al aplicar sistemáticamente el diseño orientado a objetos a los componentes de VM, podemos construir motores de ejecución vivos lo suficientemente pequeños y flexibles, donde lxs ingenierxs de VM pueden beneficiarse de herramientas de programación vivas con *feedback loops* cortos, y lxs desarrolladores de aplicaciones con menos experiencia en VM pueden beneficiarse de las conexiones causales más fuertes entre sus programas y la implementación de la VM.

Para evaluar nuestra propuesta, implementamos Bee/LMR, una VM viva para

un entorno derivado de Smalltalk en 22,057 líneas de código. Analizamos casos de estudio sobre la optimización del recolector de basura, evitando recompilaciones por parte del compilador JIT y agregando soporte para optimizar el código con instrucciones vectoriales para mostrar los compromisos de extender la programación exploratoria al desarrollo de VMs en el contexto de una aplicación industrial utilizada en producción. Basándonos en los casos de estudio, ilustramos cómo nuestro enfoque facilita el trabajo diario de desarrollo de un pequeño equipo de programadores de aplicaciones.

Nuestro enfoque permite a lxs desarrolladores de VM acceder a herramientas de programación vivas, tradicionalmente reservadas para desarrolladores de aplicaciones, mientras que lxs desarrolladores de aplicaciones pueden interactuar con la VM y modificarla utilizando las herramientas de alto nivel que utilizan a diario. Tanto lxs desarrolladores de aplicaciones como lxs de VM pueden inspeccionar, depurar, comprender y modificar sin problemas las diferentes partes de la VM con feedback loops más cortos y herramientas de mayor nivel de abstracción.

Palabras clave máquinas virtuales, compiladores, recolección de basura, metaprogramación, programación en vivo, programación orientada a objetos

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

This thesis has been translated to English and is available in the author's personal website.

Agradecimientos

Este trabajo no habría sido posible sin la colaboración de muchísimas personas e instituciones que aportaron su parte para que pueda recorrer este camino.

Especialmente en este momento histórico, quiero recalcar mi agradecimiento a la Universidad Pública, Gratuita y de Calidad, sin ella nada de todo esto sería posible.

Muchas personas me ayudaron en el proceso de forma tanto académica, como laboral y personal durante estos años. Vayan mis agradecimientos a Stefan Marr, que fue clave para *extraer la ciencia* del trabajo realizado durante años en Bee. A Diego, que aceptó ser parte de este proyecto a pesar de lo *poco convencional*. A lxs jurados, por asumir este trabajo, y su colaboración para mejorarlo.

A Gera, Javi, Leandro y Valeria que iniciaron este proyecto, gracias por su tiempo, confianza, pasión y esfuerzos dedicados a ayudarme en esta tarea titánica. Al equipo de Bee, Guille, Carlos, Seba, Alejandra, Damián, Belén, Alejandro y Alfon que aportan y me aguantan desde la época de Caesar. A Jan, Boris y Jean Baptiste que han aportado conocimientos, charlas, discuciones y sobre todo, buenas ideas. A mis amigos del DC, Fran, Pachi, el Marine, Manix, Ger y todos los que se preocuparon para que este proyecto tuviera éxito.

A Gabita, por alegrar mis días, por su paciencia y motivación. A mis viejos, Liliana y don Esteban, que siempre están tratando de ayudar, mis hermanxs, sobrinxs, tixs, los amigos de toda la vida, a los mismos de siempre, que están ahí para levantar la moral (y para bajarla cuando corresponde).

A los miembros del DC, especialmente a Rodri, consejero de estudios y gran simulador. A los compañeros del LAFHIS. A la subcomisión de doctorado, que ha lidiado con la infinidad de trámites requeridos.

A la comunidad de Smalltalk en general, y en particular a Hernán, Juan, la fundación FAST que organiza cada año mi evento favorito: las *Smalltalks*. A Quorum Software, que ha confiado durante años en la factibilidad de este proyecto, y también a Labware, que en este último tiempo también ha contribuído económicamente ayudando en la divulgación de este trabajo.

Índice general

Capítulo 1

Introducción

En las últimas décadas, cada vez más tipos de aplicaciones se trasladaron de lenguajes de bajo nivel a lenguajes de alto nivel. Las implementaciones de lenguajes habilitaron *Entornos de Programación Vivos (LPEs, por sus siglas en ingles)*, lo que nos permite modificar programas mientras se ejecutan. Este estilo de programación exploratoria impulsado por feedback loops cortos [RRL+18] permite a lxs desarrolladores de aplicaciones experimentar con problemas de dominio con una respuesta inmediata a sus acciones.

Desafortunadamente, este estilo de programación exploratoria está restringido al código de la aplicación, dejando la implementación del lenguaje subyacente inalcanzable y virtualmente inmutable para lxs desarrolladores de aplicaciones. Esto se debe a que los LPEs se implementan utilizando máquinas virtuales (VMs).

Esta separación arquitectónica intencional se encarga de lidiar con la portabilidad y de limitar las interdependencias. Las interfaces que proporcionan las VMs están diseñadas para ocultar detalles de implementación y limitar las formas en que las aplicaciones pueden interactuar con la VM y personalizarla según sus necesidades, haciendo que sea *invisible* para lxs programadores de aplicaciones. La VM sólo proporciona una interfaz opaca a través de la cual el lenguaje huésped puede comunicarse de una manera bien definida pero estrictamente limitada.

Sin embargo, esta separación estricta es una espada de doble filo. A pesar de proporcionar beneficios, mantiene intencionalmente a lxs desarrolladores de aplicaciones sin conocimiento de los detalles de diseño e implementación, lo que

les impide comprender completamente los problemas de rendimiento y las restricciones del sistema. Esto puede llevar a un código de aplicación ineficiente, quizás causando excesivas recolecciones de basura, o activando inadvertidamente recompilaciones innecesarias.

La estricta separación también impide a lxs desarrolladores mejorar la VM y aprovecharla en su aplicación, por ejemplo, mediante el compilador JIT o las facilidades de *tracing* de grafos del recolector de basura.

Como resultado, la arquitectura de las VMs de última generación hace que sea difícil para lxs desarrolladores de aplicaciones desarrollar el código más eficiente posible.

Si bien numerosos proyectos exploraron formas modernas de implementar VMs [AAB+00, WHVDV+13, BSD+08, WWW+13a, RP06, CBLFD11, IKM+97, USA05, VBG+10], casi ninguno diseñó las VMs de manera que el código de la VM misma pueda cambiarse mientras se está ejecutando, porque los componentes de la VM como el compilador JIT y el recolector de basura están separados del lenguaje host. Entre las excepciones se encuentran Klein [USA05] y Pinocchio [VBG+10], que no alcanzaron un nivel de desarrollo tal que fueran utilizados en aplicaciones reales, y Squeak [IKM+97] y sus dialectos derivados, que permiten cambios parcialmente, sólo cuando se ejecutan en un modo de simulación. Por lo tanto, la programación exploratoria posible en los LPEs no está disponible para lxs ingenierxs de VM, quienes tienen que soportar feedback loops más largos, y menos aún para lxs desarrolladores de aplicaciones, a quienes la VM les es ocultada intencionalmente.

Por supuesto, entender una VM y su complejidad esencial [BK87] también es un obstáculo. Sin embargo, para lxs desarrolladores de aplicaciones gran parte de la complejidad general es simplemente accidental y proviene del entorno de programación en el que se implementa la VM. Esto se debe a que, para trabajar en las VMs tradicionales, lxs programadores deben realizar un cambio de modo de desarrollo de VM, lo cual requiere incorporar una serie de conceptos y realizar un conjunto de tareas no triviales, que describimos detalladamente en el Apartado 3.2.

Las diferencias entre los lenguajes, herramientas y procesos básicos de desarrollo disponibles para el desarrollo de VM y aplicaciones aumentan la curva de aprendizaje de forma significativa.

Si bien pueden argumentarse que escribir un compilador JIT, un recolector de basura o funciones built-in de la VM puede requerir habilidades que no muchxs desarrolladores poseen, sostenemos que dar a lxs desarrolladores el mismo entorno de programación, herramientas y procesos de desarrollo que están acostumbrados para sus aplicaciones es mucho más importante para permitirles entender, e incluso contribuir a la implementación de un lenguaje y en particular a su máquina virtual.

Al menos en cierto grado, esto es similar a la práctica de *vendoring* de las aplicaciones, es decir, integrar el código de frameworks en los que se confía en el repositorio de código de la aplicación. Esto se hace generalmente para tener más control sobre dependencias críticas, pero también para poder entender mejor cómo interactúa la aplicación con un framework.

En esta tesis demostramos, mediante ejemplos obtenidos en casos de estudio, que es posible diseñar una VM para que pueda ser cambiada y trabajada con las mismas herramientas conocidas *por lxs desarrolladores de aplicaciones* al evitar la separación arquitectónica. Abogamos por un diseño clásico orientado a objetos, que permite la programación orientada a objetos en vivo, permitiéndonos construir una *biblioteca de tiempo de ejecución metacircular viva* (LMR). Basándonos en los siguientes casos de estudio que se toman del desarrollo de una gran aplicación industrial con 1,1 millones de líneas de código, mostramos que los problemas de nivel de aplicación pueden ser resueltos con estrategias de resolución que se mantengan en ese alto nivel, demostrando el beneficio de utilizar VMs menos opacas y más flexibles. Los casos de estudio son:

GCT - Ajuste del Recolector de Basura para un caso de uso particular.

JITC - Cambio de un compilador JIT para evitar recompilar métodos con demasiada frecuencia.

CompO - Optimización de la aplicación con instrucciones vectoriales agregadas al compilador.

Nuestros casos de estudio se llevaron a cabo con Bee Smalltalk [PBR14, PBAM17], que utilizamos para implementar el entorno de ejecución metacircular vivo Bee/LMR (por Live Metacircular Runtime). Bee/LMR es una VM autohospedada o *self-hosted*, escrita en Smalltalk. Reemplazó a la VM anterior de Bee, llamada Bee/SVM, que

tenía un diseño tradicional y estaba escrita en C++ y ensamblador. Bee/LMR permite a lxs desarrolladores de aplicaciones modificar el código de la máquina virtual en tiempo de ejecución, como lo harían con cualquier otro código a nivel de aplicación dentro del entorno de programación. Es utilizada por clientes y también diariamente para el desarrollo de PetroVR, un producto de simulación en la industria del petróleo y el gas. Si bien nuestro enfoque para Bee/LMR fue la practicidad y la estabilidad, el Apartado 8.1 muestra que el rendimiento no está demasiado alejado de una VM de Smalltalk ampliamente utilizada y es más rápido que Ruby y Python.

1.1. Declaración del Problema

Diseño de VM en capas separadas. *Las LPEs actuales están construidas sobre VMs que utilizan un diseño de dos capas. Este diseño presenta interfaces de programación de aplicaciones VM opacas que reducen las oportunidades de comprender y cambiar tanto el código de la VM como el comportamiento de ejecución de la aplicación. Además, y particularmente para lxs desarrolladores de aplicaciones, este diseño causa feedback loops largos en el desarrollo de la VM, plantea curvas de aprendizaje pronunciadas cuando se depura el comportamiento de la VM, y hace que los cambios en las VMs sean más difíciles de implementar y de poner en produción que los cambios en aplicaciones. Usualmente, es poco práctico para lxs programadores de aplicaciones cambiar el código de la VM después del deployment e incluso durante el desarrollo.*

1.2. Declaración de Tesis

Planteamos nuestra tesis de la siguiente manera:

Hipótesis. *Un diseño unificado, que ubica en el mismo nivel al ambiente de ejecución y a las aplicaciones, y que utiliza en ambos componentes las mismas herramientas estándar de encapsulamiento para la abstracción y ocultamiento de la complejidad, permite la observación y modificación inmediata, incremental y práctica de la VM. Esto crea oportunidades para abordar problemas de desarrollo de aplicaciones aprovechando los componentes de la máquina virtual.*

Para demostrar nuestra tesis, hemos aplicado técnicas de programación en vivo al código de la VM, dando forma a entornos de ejecución metacirculares vivos (LMRs, por sus siglas en inglés).

1.3. Contribución

Las principales contribuciones de este trabajo son:

- El diseño de un entorno de ejecución sin la separación arquitectónica entre VM y aplicación, que llamamos entornos de ejecución metacirculares vivos.
- La implementación de un Smalltalk self-hosted con un compilador just-in-time y un recolector de basura, donde cada parte del entorno de ejecución puede ser cambiada en tiempo de ejecución, junto con un framework que permite depurar externamente y en vivo al sistema cuando se producen fallos de bajo nivel.
- Una evaluación de los compromisos de eliminar la distinción arquitectónica entre la VM y la aplicación utilizando tres casos de estudio en una gran aplicación industrial. Los principales beneficios son las ganancias de inmediatez de los LPEs y la explorabilidad y maleabilidad de las LMRs.

Realizamos nuestros experimentos a través de la implementación de Bee/LMR, una biblioteca de tiempo de ejecución de Smalltalk self-hosted. Bee/LMR permite modificar el código de sus componentes en tiempo de ejecución, como cualquier otro código de nivel de aplicación dentro del entorno de programación.

Bee es un sistema orientado a objetos vivo con las siguientes características:

i) es de tipo dinámico, como se define en la próxima sección,

ii) utiliza recolección de basura,

iii) facilita la inspección, depuración y cambio de código arbitrariamente en tiempo de ejecución.

Bee/LMR es una implementación de un motor de ejecución de Bee que permite programar en vivo el propio motor. Proyectos similares en el pasado proporcionaron soporte para cambios dinámicos del motor de ejecución [USA05, VBG+10] y

fueron una inspiración para este trabajo. Sin embargo, ninguno de ellos fue capaz de crear un sistema completamente funcional. No admitían todas las características de programación viva del lenguaje que se estaba implementando, ni podían ejecutarse tan rápido como las VMs tradicionales. Específicamente, no implementaban recolectores de basura programables en vivo, y su rendimiento era deficiente.

Hasta el día de hoy, los motores de ejecución para entornos de programación vivos sólo admitían pocos o ningún cambio en vivo, incluso cuando se escribían usando lenguajes dinámicos. Abordamos esa situación en este trabajo.

1.3.1. Trabajos Publicados

Esta tesis se centra en las bibliotecas de motores de ejecución metacirculares vivos. Algunos resultados presentados aquí han sido publicados originalmente en [PBAM17, PM17, PC19, PMG24]. Durante el doctorado trabajé en otros temas, siempre relacionados con máquinas virtuales pero no necesariamente con LMRs. Detalles de esos trabajos, como [Pim22], [Pim18] y [CPVF18], no fueron incluidos para mantener la tesis concisa y estructurada.

1.4. Esquema de la Tesis

El resto de esta tesis está estructurado como se describe a continuación.

Capítulo 2 Detalla el contexto en el cual se ha llevado a cabo esta investigación, describiendo entornos de programación vivos, máquinas virtuales y Bee Smalltalk.

Capítulo 3 Muestra los problemas que enfrentan lxs programadores de aplicaciones al utilizar máquinas virtuales tradicionales diseñadas como una capa dividida. Se presentan 3 casos de estudio de la vida real que ejemplifican esos problemas.

Capítulo 4 Propone nuestro nuevo diseño, al que llamamos Live Metacircular Runtimes, y detalla su implementación. Esto incluye información sobre la arquitectura de Bee/LMR, sus compiladores JIT, funciones built-in y los trabajos relacionados.

Capítulo 5 Describe los desafíos de implementar la recolección de basura y la gestión de memoria en LMRs, y muestra el diseño e implementación de diferentes algoritmos utilizando Bee/LMR como vehículo de investigación.

Capítulo 6 Presenta el framework Metaphysics, que se utiliza para implementar un depurador out-of-process para Bee/LMR, diseñado con el propósito de depurar componentes de bajo nivel como el GC y el JIT compiler.

Capítulo 7 Incluye una evaluación cualitativa donde mostramos cómo las LMRs mejoran el estado del arte en VMs, cómo resuelven los problemas planteados en los casos de estudios presentados en el Apartado 3.2, y también se abordan las preocupaciones asociadas a la utilización de LMRs.

Capítulo 8 Evalúa cuantitativamente las LMRs, realizando una comparación de rendimiento con otras VMs, como así también comparando el tamaño de la implementación.

Capítulo 9 Presenta conclusiones y trabajo futuro.

Capítulo 2

Contexto

Programación Viva y la Arquitectura de Máquinas Virtuales

El foco de esta tesis es acortar los ciclos de *feedback* en el desarrollo de VMs. En este capítulo proporcionamos contexto sobre los conceptos relacionados y, para evitar confusiones, también definimos su significado específico cuando se hace referencia a lo largo de este texto.

2.1. Lenguajes Dinámicos

Definición 2.1.1 *Decimos* Lenguaje Dinámico *para referirnos a lenguajes que están diseñados para reducir la complejidad del software al ocultar detalles a lxs programadores de dos maneras: utilizan tipado dinámico y proporcionan gestión automática de memoria.*

Smalltalk, Self, JavaScript, Python, Ruby son ejemplos típicos de lenguajes dinámicos.

2.2. Entornos de Programación Vivos

Definición 2.2.1 *Un* Entorno de Programación Vivo (LPE, por sus siglas en inglés) *es un conjunto de herramientas de software que permiten desarrollar continuamente programas al mismo tiempo que se están ejecutando, permitiendo cambiar sus componentes*

sin reiniciarlos para que el feedback del desarrollo sea inmediato.

Los LPE están diseñados para permitir a lxs programadores recompilar código con feedback instantáneo, sin requerir un paso de compilación separado antes de la ejecución. Tales entornos tienen una larga historia [RRL+18], con el trabajo en SOAR siendo quizás uno de los primeros ejemplos [UBF+84] de entornos de programación vivos para lenguajes orientados a objetos basados en Smalltalk. Self también fue diseñado con esa misma filosofía [CUL89] y su implementación Klein [USA05] se acerca más a lo que consideramos un entorno de ejecución metacircular vivo, como discutimos más adelante en el Capítulo 4.

No todos los lenguajes dinámicos que proporcionan características reflectivas poseen entornos de programación vivos. Para esto necesitan poseer las herramientas para inspeccionar dinámicamente el estado de los programas, depurar su código fuente y cambiarlo de manera viva. Smalltalk es el ejemplo seminal de los entornos de programación vivos para lenguajes orientados a objetos. Self también fue diseñado con esa misma filosofía.

Otros lenguajes como Python, JavaScript y Ruby no fueron implementados originalmente como LPEs, requiriendo en la práctica reinicios del programa para evaluar nuevo código. Sin embargo, con el tiempo han adquirido características y herramientas que acortan los ciclos de feedback y permiten la programación exploratoria. Por ejemplo, IPython [PG07] brinda a lxs desarrolladores un *cuaderno computacional*, que es similar a un ciclo de read-eval-print con feedback mayormente inmediato. Desde que se volvió popular, se cambió el nombre a Jupyter para indicar el soporte de una amplia gama de lenguajes y se utiliza ampliamente, por ejemplo, para tareas de análisis de datos. Estas nuevas herramientas, si bien a diferencia de las utilizadas en Smalltalk y Self no fueron creadas mediante un enfoque integral, permiten parcialmente la programación en estos lenguajes dentro de entornos de programación vivos.

2.3. Máquinas Virtuales

Los lenguajes de programación dinámicos orientados a objetos se ejecutan sobre Máquinas Virtuales (VMs).

Definición 2.3.1 *Una Máquina Virtual de Proceso, o simplemente* Máquina Virtual, *es un programa que toma programas de alto nivel como entrada y los ejecuta en el hardware subyacente del ambiente* host.

Las máquinas virtuales poseen diferentes componentes que se ensamblan para abstraer a lxs programadores del hardware que ejecuta código máquina por detrás. Estos componentes usualmente son parsers, compiladores, gestores de memoria, operaciones built-in e interfaces para llamar dentro y fuera del lenguaje guest hacia el host y viceversa.

Existen múltiples técnicas para ejecutar el código fuente de programas en máquinas virtuales. Los intérpretes, los compiladores just-in-time (JIT) y ahead-of-time (AOT) cambian la eficiencia por facilidad de implementación, entre otros. Por lo general, el código fuente de los programas se traduce a una representación intermedia (IR), que puede ejecutarse de manera eficiente por medio de, o bien un intérprete o bien por un mecanismo que compila a código nativo. Para maximizar el rendimiento, una VM puede combinar la interpretación con la compilación JIT y AOT.

Definición 2.3.2 *Un* intérprete *es un programa que ejecuta otro programa objetivo sin traducir el objetivo a código nativo.*

Definición 2.3.3 *Un* compilador JIT *es un programa que transforma otro programa objetivo en código nativo dinámicamente, mientras el objetivo se está ejecutando.*

Definición 2.3.4 *Un* compilador AOT *es un programa que transforma otro programa objetivo en código nativo antes de que el objetivo comience su ejecución.*

Los compiladores JIT fueron popularizados por [DS84] en conjunto con otras técnicas para hacer los sistemas Smalltalk más eficientes. En los compiladores JIT, el proceso de nativización puede ser desencadenado de manera *lazy*, generalmente una vez que la VM detecta que una sección de código ha sido ejecutada múltiples veces. Los compiladores JIT intercambian tiempo de compilación por tiempo de ejecución, por lo que están diseñados de manera escalonada: cuando la VM detecta que una pieza de código se está ejecutando con cierta frecuencia, invoca a un compilador JIT rápido que realiza optimizaciones económicas; cuando detecta

que la pieza de código se está ejecutando aún más frecuentemente, puede invocar a un compilador JIT que realice aún más optimizaciones en el código, gastando más presupuesto de compilación para producir un tiempo de ejecución global más pequeño.

2.3.1. La Arquitectura de las Máquinas Virtuales

Esta sección brinda una breve descripción de las arquitecturas proporcionadas por las máquinas virtuales. Primero, discutimos la arquitectura tradicional en capas, donde se impone una separación entre la VM y la aplicación. Luego, examinamos variaciones y enfoques que afectan esta separación. Esta sección también presenta Bee Smalltalk, el sistema utilizado en nuestros casos de estudio.

Una Arquitectura en Capas: Separando las Capas de la VM y la Aplicación

Como se discute en el Capítulo 1, las máquinas virtuales tradicionales están diseñadas para separar claramente la capa de máquina virtual de la de aplicación, proporcionando una arquitectura en capas.

Esto brinda a lxs desarrolladores de aplicaciones un objetivo concreto y fijo, permitiéndoles, por ejemplo, portar una aplicación a VMs compatibles siempre y cuando sólo dependan de interfaces y comportamientos garantizados en una especificación, como por ejemplo cuando se utilza la JVM [LYBB14] o ECMAScript [Int15]. En esos lenguajes, lxs usuarixs son libres de elegir la implementación de VM que prefieran, ya que mientras las VMs respeten la especificación correspondiente, el código que hayan escrito lxs programadores debería ejecutarse de manera equivalente.

Características como la recolección de basura y la compilación just-in-time son automáticas y lxs desarrolladores de VM se esfuerzan mucho para optimizarlas y a la vez hacerlas inobservables. Sin embargo, una VM también puede ofrecer APIs que permiten a una aplicación interactuar con la VM y configurarla según sus necesidades. Por ejemplo, muchas VMs permiten a una aplicación activar manualmente la recolección de basura. Otras APIs comunes permiten el acceso por ejemplo a estadísticas en tiempo de ejecución, como el uso de memoria, el tiempo dedicado a la recolección de basura o la compilación just-in-time. A menudo, estas

APIs estarán limitadas para evitar revelar detalles de implementación. Esto crea un límite claro entre la aplicación y la VM, asegurando una separación clara de preocupaciones y protegiendo a lxs desarrolladores de aplicaciones de los detalles de implementación de la VM, minimizando así el riesgo de que una aplicación dependa de ellas.

Las VMs populares que utilizan esta arquitectura en capas incluyen CPython (la principal VM de Python), la JVM, el Common Language Runtime de .NET, y las máquinas virtuales JavaScript compatibles con ECMAScript, por ejemplo, V8, SpiderMonkey y JavaScriptCore. Las implementaciones comunes de estas VMs utilizan una combinación de código C/C++ y ensamblador. Esto brinda a lxs desarrolladores de VM el control para alcanzar el rendimiento deseado, a cambio de un alto esfuerzo de desarrollo.

Sin embargo, este diseño de dos capas también tiene sus problemas, ya que las características se proporcionan intencionalmente como una caja negra para lxs desarrolladores de aplicaciones. Esto oculta conexiones causales y evita que lxs desarrolladores de aplicaciones entiendan y confíen en los detalles de implementación.

Para dar un ejemplo, en lenguajes como Smalltalk y Self, generalmente se dice que todo es un objeto. Estos sistemas permiten a lxs programadores de aplicaciones manipular metaobjetos como métodos, diccionarios de métodos y clases o prototipos. Al proporcionar estos componentes como objetos debidamente abstraídos, lxs programadores pueden comprender más fácilmente cómo funciona el sistema, y cómo resolver problemas relacionados con la interacción de aplicaciones y metaobjetos.

En cambio, el compilador JIT, el recolector de basura y las funciones built-in no forman parte de las bibliotecas de tiempo de ejecución, por lo tanto, *no son objetos*. Así, lxs desarrolladores de aplicaciones no pueden simplemente inspeccionar el recolector de basura para saber bajo qué condiciones se activa, ni tampoco observar el compilador JIT para saber cuándo se considera un método para la compilación.

Si bien muchas de estas VMs son de código abierto, típicamente no están escritas en el mismo lenguaje que la aplicación, y no son parte de los repositorios de código directamente visibles por lxs desarrolladores de aplicaciones. Esto les impide usar sus herramientas de desarrollo habituales para observar, analizar, reutilizar y cambiar estos componentes. El feedback inmediato que reciben de sus herramientas, con

pocas excepciones, termina en el punto de entrada a la capa de la VM.

2.4. Bee Smalltalk

Bee Smalltalk, nuestro vehículo de investigación, inicialmente se ejecutaba sobre una VM tradicional implementada en C++ y ensamblador. A lo largo de este trabajo, nos referimos a Bee/SVM para hacer referencia a Bee Smalltalk ejecutándose sobre esa VM estática. Bee/LMR fue diseñado como un reemplazo directo de la VM tradicional, y cuenta con un compilador JIT, recolector de basura (GC) y numerosas funciones built-in. La principal diferencia es que Bee/LMR está implementado en Smalltalk utilizando una capa de aplicación/VM unificada, mientras que en Bee/SVM la VM está separada de la aplicación y su código C++ es prácticamente invisible para lxs programadores de aplicaciones.

Los métodos de Smalltalk se compilan a un formato de bytecode, y cuando se ejecutan por primera vez, se compilan a través de un compilador JIT de tipo template. No hay modo intérprete.

2.4.1. Gestión de Memoria

Tanto Bee/SVM como Bee/LMR poseen dos recolectores de basura: uno generacional para el espacio de objetos nuevos y otro para el espacio de objetos antiguos. Los algoritmos implementados para cada uno de ellos en Bee/LMR son distintos de los implementados en Bee/SVM. Mientras que las características de estos recolectores se explican en mayor detalle en el Capítulo 5, aquí presentamos brevemente los rasgos principales de su diseño.

En Bee/LMR los GCs son metacirculares y modificables en tiempo de ejecución. Debieron ser diseñados para que, durante la recolección de basura, se mantenga la memoria de los objetos siempre en estado válido, como se describe en [PBAM17]. Para garantizar que los objetos sean válidos durante el proceso de recolección, los GCs realizan una copia de objetos a medida que son movidos, almacenando punteros *forwarding* en tablas externas. De esta manera, durante el proceso de recolección el GC solo modifica los bits de marca en los headers de los objetos, en vez de sobreescribirlos completamente. El recolector del espacio de objetos

antiguos de Bee/LMR es de tipo G1 [DFHP04].

El espacio joven de memoria consta de un gran **eden** y dos espacios más pequeños, **from** y **to**. Cuando el **eden** se queda sin espacio, se ejecuta el recolector generacional. El mismo mueve los objetos sobrevivientes en **eden** y **from** mayormente a **to**, y luego intercambia **from** y **to**. Los objetos maduros se mueven a la zona antigua, donde permanecen hasta que se ejecuta el recolector G1. Esta ejecución se desencadena por una heurística que considera el crecimiento del heap desde la recolección G1 anterior.

La zona antigua se divide en *espacios* de igual tamaño, y además existe otra zona para objetos grandes, aquellos que exceden un umbral de tamaño. Los objetos maduros se crean mediante *bump-allocation* en los espacios antiguos. A medida que el cómputo evoluciona, algunos objetos antiguos se vuelven inalcanzables. El recolector G1 lleva un seguimiento del uso alcanzable de cada espacio antiguo, donde un uso bajo implica una mayor fragmentación. Antes de comenzar la etapa de *tracing*, el GC selecciona los espacios a ser evacuados, eligiendo aquellos que tienen las proporciones de uso más bajas. La evacuación hace que los objetos asignados en espacios fragmentados se compacten en otros espacios libres.

2.4.2. Otras Características

El sistema Smalltalk en sí cuenta con las características típicas de un entorno de desarrollo basado en imagen. Utiliza un navegador de clases clásico con inspectores de objetos y espacios de trabajo para la evaluación de código arbitrario. Como en la mayoría de los sistemas Smalltalk, el depurador permite a lxs desarrolladores inspeccionar, explorar y modificar el sistema en tiempo de ejecución cambiando el estado y el código a voluntad.

Bee es la plataforma para una aplicación de simulación en la industria del petróleo y el gas desarrollada por un pequeño equipo. Con el tiempo, el equipo de desarrollo se dio cuenta de que la estricta separación entre la VM y la aplicación causaba un costo demasiado alto, porque la VM debía mantenerse como un repositorio de código separado. Por ejemplo, investigar crashes, ya sea que fueran causados por la aplicación o por un error en la VM, requería demasiado esfuerzo. Como un equipo pequeño con una aplicación de producción de 1.1 millones de líneas de código

que dependía de Bee, se decidió que se debían tomar medidas importantes para facilitar el mantenimiento de Bee para todos los miembros del equipo. Describimos el resultado en el Capítulo 4.

Capítulo 3

Motivación y Casos de Estudio

Para motivar nuestra exploración de un diseño de motor de ejecución que elimine las barreras entre la aplicación y la VM, ejemplificamos los tradeoffs basándonos en tres casos de estudio originados en el desarrollo de nuestra aplicación, PetroVR, sobre Bee.

Los mismos fueron tomados de escenarios de desarrollo reales de un producto implementado dentro de un LPE. Cada caso de estudio describe un escenario y los pasos de solución necesarios en una VM de tipo tradicional.

Para cada caso de estudio, identificamos los obstáculos específicos con estas VMs y las características necesarias para permitir a lxs desarrolladores de aplicaciones beneficiarse de sus herramientas normales y de ciclos de feedback cortos. Examinamos cómo estas VMs hacen difícil tratar con los casos presentados. Finalmente, distinguimos tres características que se necesitan para crear entornos de programación exploratoria con ciclos de feedback cortos, donde lxs desarrolladores pueden comprender y mejorar más fácilmente la VM. En el Capítulo 4, proponemos el uso de entornos de ejecución metacirculares vivos (LMRs) como un tipo de VM que incluye esas tres características.

3.1. Casos de Estudio

Para esta tesis, seleccionamos un caso de estudio sobre el impacto en el rendimiento de la recolección de basura, uno sobre cómo evitar recompilaciones por

el compilador JIT, y finalmente uno sobre agregar soporte para instrucciones vectoriales del procesador para un mejor rendimiento.

Estos casos fueron recopilados a partir de experiencias del mundo real con un producto desarrollado en un LPE que utilizaba una VM tradicional. Todos ellos exponen una categoría compartida de problemas al trabajar con VMs de última generación: son difíciles de depurar dentro del LPE, requieren aprovechar componentes de la VM de formas no anticipadas, configurar el sistema para poder trabajar con la VM representa un desafío para lxs desarrolladores de aplicaciones, dificultando en la práctica resolver estos problemas mediante programación viva.

3.1.1. Caso de Estudio 1: Ajuste de la Recolección de Basura (GCT)

La forma en que determinadas aplicaciones asignan memoria puede desencadenar un comportamiento no deseado en su recolector de basura. Los cuellos de botella de GC no son infrecuentes en aplicaciones que asignan objetos libremente, donde la memoria es recuperada automáticamente con una intervención mínima del código de la aplicación. Los administradores de memoria no son infalibles. Por ejemplo, demasiadas asignaciones en un heap demasiado pequeño pueden causar recolecciones demasiado frecuentes, lo que resulta en problemas de rendimiento.

Ciertos patrones de asignación y estructuras del heap también pueden causar largas pausas de GC. En otros casos, las recolecciones más frecuentes pueden eliminar más rápido grandes cantidades de objetos temporales, lo que puede hacer en cambio que un tamaño de heap más pequeño tenga un rendimiento general mejor.

Al diagnosticar estos problemas en una aplicación, lxs desarrolladores que enfrenten este caso de estudio primero necesitarán identificar cuándo ocurre el GC y recopilar estadísticas sobre el tamaño del heap, las tasas de supervivencia de objetos y la fragmentación del heap. Para aplicaciones de larga duración, también pueden necesitar filtrar estas estadísticas para inspeccionar momentos específicos, aislando el cómputo de interés.

Una VM de última generación puede tener una variedad de recolectores diferentes. Para simplificar, sólo discutiremos un escenario de procesos *single-threaded*. Aquí,

una VM podría tener un GC generacional que se activa cuando no hay espacio en la guardería para asignar nuevos objetos. Las VMs tienen varias heurísticas para iniciar recolecciones menores o completas. Una aplicación puede ser capaz de usar una API de la VM que proporcione acceso de lectura a estadísticas básicas, o permita iniciar manualmente una pasada de recolección de basura.

Sin embargo, cambiar el GC o ajustar finamente sus parámetros, a menudo requiere iniciar la VM con parámetros especiales. Los detalles completos del GC pueden estar disponibles sólo a través de logs de consola, normalmente habilitados por parámetros de línea de comandos, o APIs de monitoreo que se pueden usar a través de herramientas externas para observar detalles de GC y asignación, como es el caso de la JVM. Por lo tanto, en la mayoría de los casos, lxs desarrolladores de aplicaciones tienen que salir del entorno de desarrollo de la aplicación y posiblemente aprender a utilizar herramientas diferentes.

Todas las soluciones actuales tienen en común que mantienen la estricta separación entre la aplicación y la VM y dependen de mecanismos y herramientas externas para proporcionar las averiguaciones necesarias.

Para el problema de rendimiento específico en cuestión, después de recopilar los datos, estos deberán analizarse con nuevas herramientas cuya forma de uso deberá ser aprendida, o mediante el uso de código personalizado. Podría descubrirse entonces que una solución al problema es establecer un tamaño de heap específico mientras el código causante del problema se está ejecutando, equilibrando mejor el costo del GC con la frecuencia con la que se realiza la recolección de basura. Sin embargo, ajustar el tamaño del heap durante la ejecución no es algo generalmente admitido por la mayoría de las VMs. Es más, modificar la VM es impensable para lxs desarrolladores de aplicaciones en el caso de VMs tales como la JVM, de JavaScript y muchos otros lenguajes. No solo requiere que lxs desarrolladores aprendan a compilar estos sistemas, sino también que naveguen a través de posiblemente cientos de miles de líneas de código.

En resumen, para lxs desarrolladores de aplicaciones, los sistemas de GC disponibles proporcionan interfaces opacas que dificultan determinar qué sucede detrás de escena y recopilar estadísticas del GC. Modificar una sola línea de código del GC requiere un costo inicial significativo de configuración y compilación, y es una tarea desafiante debido al cambio a un repositorio externo de código de una

VM opaca, especialmente cuando está escrita en un lenguaje no interactivo. Los bucles de feedback son largos, y los detalles sobre cómo se gasta el tiempo en las rutinas de manejo de memoria son escasos. Es impráctico analizar los eventos de activación del GC, y el código del GC no puede ser fácilmente inspeccionado o cambiado de la misma manera en que se desarrollaría una aplicación.

3.1.2. Caso de Estudio 2: Recompilaciones Recurrentes por el Compilador JIT (JITC)

El problema de generar el código objeto óptimo a partir del código fuente de un programa es indecidible en general [ASU20]. Los motores de tiempo de ejecución implementan heurísticas que intentan optimizar al máximo los escenarios más frecuentes. Cuando se encuentran con casos inusuales, simplemente tratan de asegurarse de que el rendimiento se degrade de la manera más gradual posible. Los *inline caches* monomórficos y polimórficos [DS84, HCU91], y los procedimientos de *on-stack replacement* [HU96, FQ03], son sólo algunas técnicas que permiten tratar con la adaptación del entorno de ejecución para mejorar el rendimiento dinámicamente. Sin embargo, no existe un enfoque único que pueda cubrir todos los casos. Además, algunas de estas técnicas son complejas de implementar y pueden no estar disponibles en el sistema.

Dado que la compilación en tiempo de ejecución conlleva un costo, las VMs deben determinar cuándo la compilación JIT vale la pena. Sin embargo, como con cualquier heurística, los casos inusuales pueden resultar en un rendimiento no deseado.

En el caso de estudio, una actualización del producto se envió a un cliente, quien informó que la nueva versión era mucho más lenta que la anterior. El problema subyacente era que en la nueva versión, el código de la aplicación agregaba y eliminaba dinámicamente métodos a un objeto, lo cual invalidaba su código nativo en tiempo de ejecución, causando recompilaciones frecuentes.

En una VM de última generación, para detectar la causa de los problemas de rendimiento lxs desarrolladores comenzarían a hacer *profiling* la aplicación. Sin embargo, dado que la compilación JIT es transparente, un profiler normalmente no muestra el tiempo de compilación. También es poco probable que desarrolladores

de aplicaciones piensen en buscar estadísticas de compilación, o que se den cuenta de que el thread del compilador pueda mostrarse como ocupado durante más tiempo de lo habitual.

De hecho, nuestrxs desarrolladores de aplicaciones no pudieron encontrar el problema usando el profiler. Sin embargo, construyeron un mecanismo que les permitió usar una búsqueda binaria a través de los cambios al código de la actualización para identificar la causa, lo que lxs llevó al cambio en el código que agregó las modificaciones dinámicas de métodos. Así, debido a la estricta separación entre la VM y la aplicación, las herramientas estándar fallaron a nuestrxs desarrolladores, y tuvieron que depender de otras herramientas usando un enfoque de desarrollo diferente para encontrar el lugar donde se originaba el problema.

Después de identificar la causa raíz, nuestrxs desarrolladores necesitarían una buena comprensión del compilador JIT para poder idear una solución y evitar las recompilaciones. Aunque el problema se puede resolver con una heurística simple de cacheo de código nativo, cambiar el código del compilador JIT que está fallando probablemente sea demasiado complejo para lxs desarrolladores de la aplicación usando el enfoque tradicional de separación entre aplicación y VM. Por lo tanto, necesitarán reescribir el código de la aplicación para evitar agregar y eliminar métodos dinámicamente.

3.1.3. Caso de Estudio 3: Optimizaciones SIMD (CompO)

Aplicaciones como la nuestra, con mucha aritmética de punto flotante, pueden ganar rendimiento mediante el uso de operaciones vectorizadas en procesadores modernos x86 y ARM. Sin embargo, muchos lenguajes de alto nivel no proporcionan optimizaciones del compilador o APIs para usarlas. [1]

Para optimizar los métodos que realizan el cálculo, nuestros desarrolladores de aplicaciones necesitan usar directa o indirectamente las operaciones de punto flotante vectorizadas (SIMD: single instruction, multiple data).

En las VMs de última generación, se podría esperar o solicitar al proveedor de la VM que agregue soporte para las operaciones vectorizadas. Una posibilidad es simplemente esperar a que los fabricantes de la VM implementen las operaciones

[1] Incluso Java solo tiene soporte experimental: https://openjdk.org/jeps/438.

vectorizadas ellxs mismxs. Sin embargo, esto puede ser impráctico. Por ejemplo, para Java, es probable que aún falten algunos años para que se finalice el soporte. Para dar otro ejemplo, aunque las instrucciones MMX se agregaron a los procesadores Intel en 1997, solo fue después de 2014 que algunas VM de JavaScript comenzaron a agregar extensiones SIMD, ¡17 años después!

Una alternativa es usar una extensión a la VM, basada en código de bajo nivel, lo que hace que las operaciones sean accesibles. JVM proporciona la *Java Native Interface*, y otras VM suelen tener mecanismos similares.

Otro enfoque, la exocompilación [IBR+22], permite abstraer algoritmos de procesamiento de datos de la *planificación*. Esto simplifica la generación de código máquina eficiente que apunta a distintas instrucciones y arquitecturas de hardware.

Sin embargo, estos enfoques requieren que lxs desarrolladores de la aplicación cambien el lenguaje y las herramientas, lo que conlleva un esfuerzo adicional y probablemente una cadena de errores más larga de lo habitual. En el peor de los casos, es posible que ni siquiera haya disponibles instrucciones intrínsecas del compilador y nuestrxs desarrolladores necesiten usar inline assembly. Dependiendo de cómo funcione la interfaz nativa de la VM, también puede venir con ineficiencias adicionales debidas a los cambios de contexto entre código normal y de extensión, quizás porque se necesita convertir argumentos de alto nivel en argumentos de bajo nivel antes del cómputo, y posiblemente también para guardar el estado de la VM antes de llamar al código nativo.

3.2. Problemas con las VMs de Última Generación

El patrón común en nuestros casos de estudio es que la estricta separación entre la aplicación y la máquina virtual, a pesar de todos sus beneficios de ingeniería, también tiene un costo.

La serie de acciones requeridas para abordar cada problema en cada caso de estudio presentado, con las VMs de última generación, se complementa con otra serie de acciones no triviales, que aumentan la complejidad accidental del sistema.

3.2.1. Observabilidad Limitada de la VM (PG1)

Dado que las VMs están diseñadas para abstraer y ocultar detalles de implementación, las API disponibles suelen ser mínimas. Incluso las interfaces de herramientas disponibles, como las disponibles para la JVM, pueden estar limitadas por el deseo de minimizar el overhead en tiempo de ejecución al recopilar datos.

En nuestros casos de estudio, esto significaba que recopilar la información deseada sobre la recolección de basura requería o bien aprender sobre herramientas externas o bien procesar el output del log de la VM. Para comprender el comportamiento del compilador JIT, las JVM sí proporcionan los datos como parte de las interfaces de herramientas y las *Java Management Extensions*.[2] Sin embargo, dado que es transparente para lxs desarrolladores, no visible en profiles, y menos conocido como fuente de problemas de rendimiento que el recolector de basura, es poco probable que lxs desarrolladores de aplicaciones lo consideren como una posible fuente del problema.

3.2.2. Modo Separado de Desarrollo de la VM (PG2)

Dado que, como el autor de este trabajo, lxs lectores pueden ser desarrolladores de VM, su primer instinto puede ser *sí, arreglemos la VM*. Sin embargo, en el caso común, trabajar en la VM es demasiado diferente de trabajar en la aplicación como para transferir fácilmente conocimientos de lenguaje y herramientas. En la mayoría de los casos, la VM se implementa en un lenguaje de más bajo nivel, tiene pasos de construcción complejos que deben entenderse, y requiere el uso de diferentes herramientas de desarrollo. Por lo tanto, hacer un cambio en una VM requiere mucho aprendizaje adicional antes de poder comenzar a explorar los típicos códigos fuente de las VMs de cientos de miles de líneas de código. Por lo tanto, la solución estándar para lxs desarrolladores de aplicaciones será encontrar soluciones alternativas en el código de la aplicación usando las herramientas que ya conocen.

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

3.2.3. Feedback Loops de Editar, Compilar y Ejecutar Más Largos en los Componentes de VMs (PG3)

En situaciones como nuestro caso de estudio donde queremos utilizar operaciones vectorizadas en nuestro código de simulación, bajar al nivel de desarrollo de VMs es la única opción. Si bien la construcción de extensiones a menudo es facilitado con documentación y herramientas, lo cual requiere menos aprendizaje que cambiar la VM en sí, todavía es un cambio de lenguaje de programación y herramientas, y viene con la carga de feedback loops de editar-compilar-ejecutar más largos de lo que lxs desarrolladores de aplicaciones están acostumbrados a partir de sus lenguajes de alto nivel.

Con todo este aprendizaje requerido, idealmente se pueden cambiar cosas, explorar consecuencias y tener feedback inmediato, como lxs desarrolladores de aplicaciones están acostumbrados con sus entornos de programación viva. Sin embargo, debido a que las extensiones requieren lenguajes de más bajo nivel, los cambios pueden requerir reiniciar la VM y la aplicación antes de que tengan efecto, lo cual es un cambio drástico en el flujo de desarrollo.

Esta falta de feedback instantáneo aumenta el tiempo desde las ideas hasta los experimentos y el costo de construir tales extensiones.

3.3. Resumen

Demostramos que el uso de una estricta arquitectura de dos capas que separa el código de la VM y de la aplicación en entornos de programación vivos no invita a lxs desarrolladores a lidiar con los tres casos de estudio.

Por el contrario, tal diseño plantea barreras de conocimiento y practicidad que son artificiales, no relacionadas con los problemas que se están abordando en cada caso. La barreras impuestas por esta arquitectura impiden a lxs desarrolladores detectar y solucionar los problemas que se les presentan en su trabajo real.

La división entre la aplicación y la VM corta la conexión causal que asocia los problemas con la solución, aumentando el tiempo necesario para encontrar el camino a través de repositorios de código que pueden tener cientos de miles de líneas. Estas tareas adicionales a realizar pueden considerarse demasiado difíciles

para lxs programadores de aplicaciones.

Si bien la arquitectura presenta beneficios como asegurar, por ejemplo, la portabilidad, también limita la observabilidad de la VM (PG1), normalmente conduce a un modo de desarrollo separado para la VM (PG2), y provoca un feedback loop de editar-compilar-ejecutar más largo con los componentes de la VM que para el código de la aplicación (PG3).

Capítulo 4

Entornos de Ejecución Metacirculares Vivos

Para superar las limitaciones presentadas en el Apartado 3.2, proponemos la implementación de entornos de ejecución metacirculares vivos (Live Metacircular Runtimes, LMR). Sostenemos que las LMRs crean una sinergia con los entornos de programación vivos, lo cual hace práctico modificar en tiempo de ejecución los componentes de la VM, que tradicionalmente se han considerado estáticos.

En este capítulo, describimos nuestro vehículo de investigación, Bee/LMR, que es una implementación de una LMR sobre Bee Smalltalk, un entorno de programación vivo que originalmente estaba diseñado para ejecutarse utilizando una VM escrita en C++.[1]

Básicamente, una LMR consiste en una biblioteca de módulos escritos en el mismo lenguaje que las aplicaciones, que reemplazan lo que normalmente se provee como una capa separada y oculta, la VM. Estos módulos pueden implementar intérpretes, compiladores just-in-time y ahead-of-time, recolectores de basura y operaciones built-in, todo lo necesario para permitir que el sistema ejecute código. Los módulos de la LMR forman el corazón del entorno de programación vivo. La

[1] Si bien esta tesis describe el trabajo realizado en Bee/LMR, que no está disponible libremente, también se encuentra en desarrollo una implementación de una LMR sobre un dialecto de Smalltalk de código totalmente abierto, conocido como Egg. La implementación de la LMR sobre Egg está disponible de forma gratuita.

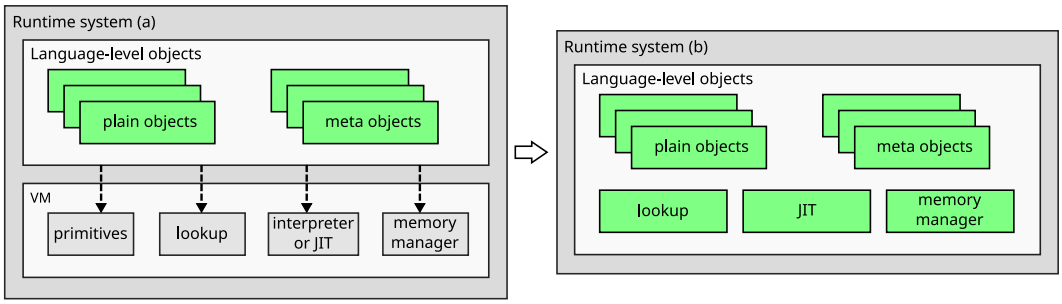


Figura 4.1: Transformación de Bee/SVM (izq.) en Bee/LMR (der.). Mientras que Bee/SVM originalmente utilizaba una VM tradicional, implementada en C++ y assembly en una capa separada y compilada estáticamente, Bee/LMR se implementa como una serie de bibliotecas comunes escritas en Smalltalk, entre la cuales están el JIT, el GC y un kernel con funciones built-in tales como el lookup de métodos.

Figura 4.1 muestra la transformación de diseño necesaria para convertir Bee/SVM en Bee/LMR.

Antes de adentrarnos en los detalles de implementación del módulo LMR, describimos los aspectos clave de Bee Smalltalk para ayudar a contextualizar todo el panorama del sistema.

4.1. Bee Smalltalk

Bee es un dialecto de Smalltalk que sigue vagamente la especificación de Smalltalk-80 [GR83], así como la jerarquía de clases principales de Squeak Smalltalk [IKM+97]. Su VM original, ahora obsoleta, fue implementada en C++ y ensamblador, y Bee tenía un diseño de VM/aplicación de dos capas. A esta VM, la cual es compilada estáticamente y no puede ser modificada fácilmente en tiempo de ejecución, la llamamos Bee/SVM durante este trabajo.

Para contextualizar los cambios requeridos para la migración desde Bee/SVM hacia Bee/LMR, comenzamos dando una visión general de la arquitectura de Bee y sus VMs.

4.1.1. Modelo de Ejecución

En Bee, el código Smalltalk se precompila desde el código fuente a métodos compilados en forma de objetos, los cuales contienen instrucciones en forma de bytecode. Los bytecodes utilizan una arquitectura de CPU virtual, por lo que no pueden ser ejecutados directamente. Bee no implementa un intérprete. En su lugar, los compiladores JIT y ahead-of-time traducen los bytecodes de los métodos a código nativo cuando es necesario. Generalmente llamamos a estos compiladores *nativizadores*, y esta última traducción *nativización*.

4.1.2. Arquitectura de CPU Virtual

En Bee, tanto los bytecodes como los nativizadores están diseñados en torno a una arquitectura de CPU virtual. Esta arquitectura consta de una pila y registros de propósito especial descritos en el Cuadro 4.1. Diferentes backends de ensamblador asignan los registros abstractos a registros concretos. Por ejemplo, el backend AMD64 asigna el registro **RAX** a **R**, **RSI** a **S** y así sucesivamente. Este diseño permite que los generadores de código de alto nivel trabajen con una única arquitectura virtual, por lo que cambiar la ISA de destino se convierte mayormente en un cambio del backend del ensamblador.

4.1.3. Código Nativo en Métodos Compilados

Cuando un método es nativizado, se le asigna un objeto **NativeCode** que servirá como contexto de ejecución para el método. Este objeto contiene un puntero a un byte array con el código máquina que resultó de la traducción de los bytecodes, y la lista de literales que son referenciados desde ese código máquina. El propio código máquina no contiene ningún puntero a objetos Smalltalk, lo que facilita la recolección de basura, como se explicará en el Capítulo 5. El acceso a los literales del método desde el código máquina se realiza indirectamente a través del registro M, que se inicializa justo antes de entrar al método. No hay distinción entre los byte arrays de código máquina y otros tipos de byte arrays, todos son objetos que se almacenan en el mismo heap.[2]

[2]Lo cual implica que todo el heap se marque como memoria ejecutable, una decisión de diseño que podríamos cambiar en el futuro para mejorar la seguridad.

Nombre	Contenido	Guardado por
R	receptor del mensaje y valor de retorno	scratch
S	**self** dentro de un método	callee
M	contexto de ejecución del método o bloque actual	callee
E	variables de entorno del método o bloque actual	callee
A	primer argumento temporal	scratch
T	temporal o segundo argumento temporal	scratch
G	array de variables globales	fijo
nil	objeto nil	fijo
true	objeto true	fijo
false	objeto false	fijo
SP	puntero de pila	callee
FP	puntero de frame	callee
PC	contador de programa	–

Cuadro 4.1: Nombres de los registros de la CPU virtual de Bee

4.1.4. Interacciones entre la VM y las Aplicaciones

Originalmente, Bee se ejecutaba sobre una VM compilada estáticamente, que incluía el compilador JIT, las primitivas built-in y los recolectores de basura. La VM estaba compilada estáticamente y contenía la funcionalidad básica para permitir la ejecución del código de la aplicación. La VM cargaba una imagen de Smalltalk y comenzaba a ejecutar código Smalltalk desde un método de inicio, el cual nativizaba just-in-time para luego saltar a su código nativo. El código Smalltalk se ejecutaba a partir de ese momento, llamando ocasionalmente de vuelta a la VM de dos maneras: explícita o implícitamente. Explícitamente a través de la ejecución de métodos compilados *primitivos*. Implícitamente cuando fuera invocado un método que no tuviera aún código nativo (desencadenando el compilador JIT).

4.2. Migración desde el Diseño de Capas Separadas al de Módulos de Bee/LMR

Los módulos de la LMR funcionan como un reemplazo directo de la funcionalidad original de la VM. En la implementación de la LMR, simplemente no hay distinción entre las capas de la VM y de la aplicación. El código de la LMR se integra en el sistema en los lugares donde solían ocurrir las interacciones entre ambas capas, con la diferencia de que dentro de Bee/LMR no se necesita ningún mecanismo especializado para activar el código de una capa separada de la VM, dado que esta capa *no existe*. Los objetos y los mensajes son todo lo que se necesita.

La migración desde un diseño tradicional de capas separadas requirió tres componentes principales: un traductor de bytecodes a código nativo (que llamamos nativizador), las funciones built-in y el recolector de basura. Las funciones built-in y el recolector de basura requieren acceso eficiente a partes de la VM que normalmente no son directamente accesibles desde la semántica del lenguaje (es decir, *headers* de objetos y acceso irrestricto a memoria). Esto se hace posible en Bee/LMR mediante la adición de características al nativizador, así que lo explicamos primero.

4.2.1. Filosofía

La idea principal que impulsa la implementación de Bee/LMR es que nada que pueda ser implementado en alto nivel debería permanecer implementado en bajo nivel. Conceptos que típicamente se programan como partes de bajo nivel de un sistema, por ejemplo, primitivas, compiladores y depuradores, deberían realizarse en el lenguaje de alto nivel como el resto del sistema. Algunxs investigadores abogan por la programación en alto nivel del bajo nivel, argumentando que la productividad mejora mientras que los impactos en el rendimiento pueden superarse y reducirse a niveles insignificantes [FBC+09].

4.3. Nativizadores Personalizables

El primer componente que fue necesario reemplazar para implementar Bee/LMR fue el compilador JIT. Mientras que el JIT de Bee/SVM estaba escrito en C++,

en Bee/LMR se pasó a un nativizador escrito en Smalltalk. El diseño actual de Bee/LMR incluye dos nativizadores, principalmente por razones de rendimiento. Un compilador JIT de tipo template de primera etapa, que itera los bytecodes de los métodos traduciéndolos a código nativo muy rápidamente, pero realizando sólo pequeñas optimizaciones. Un compilador optimizador de segunda etapa es capaz de realizar más optimizaciones a costa de una menor velocidad de nativización.

Con el propósito de permitir un acceso eficiente a partes de la VM normalmente ocultas, en comparación con lo que hacía el JIT de Bee/SVM, sólo se necesitó una extensión: el nativizador de Bee/LMR permite personalizar la traducción de los bytecodes de envío de mensajes, para realizar acciones personalizadas con un conjunto de nombres de mensaje específicos, que se distinguen por comenzar con un carácter de underscore (_).

El mapeo de nombres de mensajes especiales a nativizaciones especiales es determinado por la configuración de un objeto de tipo **NativizationEnvironment**, que también conoce la arquitectura de destino y otros detalles importantes como se explica a continuación. Este entorno de nativización es el punto de entrada para la traducción de bytecodes a código nativo. Este objeto hace referencia a los componentes del sistema que deben ser conocidos para producir código nativo: la arquitectura de la CPU del sistema de destino, la configuración de envío de mensajes especiales, las barreras de escritura, las rutinas de *safe-point*, las globales, la ABI del sistema, etc.

4.3.1. Personalización de la Traducción y Semántica del Envío de Mensajes

En los sistemas Smalltalk tradicionales, el envío de mensajes se implementa como un componente del entorno de ejecución cuyo código es invisible. Por lo general, los envíos de mensajes se codifican con un bytecode de envío, y la VM se encarga de implementar un mecanismo de ejecución que efectúa esos envíos de mensajes de acuerdo con la semántica de Smalltalk. La VM buscará un método que implemente el selector que se está enviando en la jerarquía de clases del receptor y, cuando lo encuentre, invocará el método correspondiente.

Si bien existen maneras de alterar el mecanismo de dispatch en Smalltalks tradi-

Listado 4.1: Selección del **MessageLinker** por el **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]
```

cionales[3], estos mecanismos son de alto nivel y no permiten cambiar realmente el algoritmo interno de dispatch de mensajes proporcionado por la VM. Para poder realizar operaciones de bajo nivel utilizando Smalltalk, como acceder a los headers de los objetos y hacer aritmética de punteros, fue necesario implementar, en alto nivel, mecanismos de dispatch de más bajo nivel.

El diseño de Bee/LMR permite cambiar el mecanismo de envío de mensajes de manera arbitraria en tiempo de ejecución, en función de cada message-send en particular. La base de esta flexibilidad es el uso de una serie de objetos linker, conocidos como **MessageLinker**s, que trabajan en colaboración con el **NativizationEnvironment**.

Actualmente, el mecanismo de envío de mensajes se determina por el selector enviado: la generación de código nativo para un envío de mensaje se delega a un linker de mensaje compuesto, que elige un subtipo concreto de linker según el selector que se envía, como se muestra en el Listado 4.1. Cuando se envía **emitSend:using:** al objeto **NativizationEnvironment** para nativizar un envío de mensaje, el entorno busca entre sus linkers configurados, para determinar cuál utilizar para generar código para el selector pasado como argumento. Finalmente, le delega al subtipo la generación de código nativo para el envío de ese mensaje. Diferentes subtipos de linkers de mensaje implementan distintas semánticas de envío de mensajes.

Cuando un método se nativiza, para cada envío de mensajes que no se inlinea,

[3] como con los mecanismos **doesNotUnderstand:** o **perform:**

Listado 4.2: Ejemplo de implementación y uso del underprimitive **_isSmallInteger**.

```
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]
```

se adjunta un objeto **SendSite** al código nativo del método. El send-site codifica toda la información necesaria para ejecutar el envío de mensajes cuando llegue el momento, para cachear información del tipo del receptor para las inline-caches y también para descartar esa información si el código de la aplicación se actualiza.

Linker de Mensajes Inline y Underprimitives

Este linker permite generar código assembly inline en un send-site en vez del código nativo que generaría un envío de mensaje normal. Los mensajes linkeados de esta manera se conocen como *underprimitives*. El Listado 4.2 muestra la implementación del underprimitive **_isSmallInteger**. El método **assembleTestSmallInteger** interactúa con un objeto ensamblador que generará el código máquina requerido.

Las underprimitives se implementan siguiendo la ABI presentada en el Cuadro 4.1. En el caso de **_isSmallInteger**, la suposición es que el receptor del mensaje estará en el registro R, y el resultado se devolverá en ese mismo registro. Esta suposición está garantizada por el compilador JIT y debe ser mantenida por lxs programadores en los métodos que implementan underprimitives.

Desde el punto de vista del parser de Smalltalk, el underscore al principio de un nombre de mensaje no tiene ninguna semántica especial. Simplemente es la nomenclatura utilizada para advertir a lxs programadores de que el mensaje tiene algún significado particular, como ser de bajo nivel o una underprimitive. Tampoco hay una semántica especial en la compilación a bytecodes. El único cambio de semántica se produce en la ejecución del mensaje, a través del mecanismo de nativización.

Los selectores implementados como underprimitives se definen dinámicamente a través de metaprogramación. Los métodos implementados en la clase **InlineMessageLinker** que pertenecen a la categoría **underprimitive** se agregan automáticamente a la lista de selectores underprimitive del linker de mensajes. En el ejemplo de Listado 4.2, **#_isSmallInteger** se configura automáticamente como un underprimitive, y el JIT emitirá el código generado por **assembleTestSmallInteger** para cualquier mensaje **#_isSmallInteger** que vea.

En Bee/LMR, las underprimitives son la herramienta base utilizada para acceder eficientemente a los headers de los objetos. El Listado 4.3 muestra la implementación de un método **_size**. Al recibir este mensaje, el receptor devuelve el tamaño almacenado en el header del objeto. Tanto **_smallSize** como **_largeSize** están implementados como underprimitives. Por otro lado, **_size** como **_isSmall** son métodos Smalltalk linkeados estáticamente, como se describe a continuación.

4.3.2. Linkeo Estático de Métodos

En algunas situaciones particulares se vuelve útil implementar un envío de mensaje como si fuera una llamada a una función linkeada estáticamente. En lugar de realizar un lookup completo del método correspondiente al tipo del receiver en tiempo de ejecución, este linker causa la ejecución de un método pre-designado estáticamente para el selector correspondiente.

Los mensajes linkeados estáticamente pueden ser utilizados a modo de optimización, para ahorrar costo de ejecutar el lookup. Esto es especialmente útil con métodos de bajo nivel implementados en la raíz de la jerarquía de clases, en la clase **ProtoObject**. Los métodos implementados en **ProtoObject** son propensos a enviar mensajes cuya implementación también se encuentra en **ProtoObject**. Pero

Listado 4.3: Uso e implementación de underprimitives que acceden a los headers de los objetos.

```
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
```

como los receptores concretos de esos mensajes pueden ser objetos de cualquier tipo, los envíos de esos mensajes tienden a volverse megamórficos. Los mensajes linkeados estáticamente se vinculan en tiempo de nativización, y no involucran comprobaciones de clase, lo que hace que este tipo de código se ejecute más rápido que un mensaje normal.

Para entender mejor el problema recien mencionado, consideremos el método **ProtoObject»_size** presentado en el Listado 4.3, que envía el mensaje **_isSmall**, que también está implementado en **ProtoObject**. Supongamos que el mensaje **_size** se envía a un objeto de tipo Array. Durante la ejecución, el mensaje **_isSmall** se enviará al mismo receptor, de tipo Array, y se creará una inline cache que mapea **ProtoObject»_isSmall** a la clase Array. Si ahora se enviara **_size** a un objeto de otro tipo, el inline cache fallaría y se crearía una caché polimórfica, para agregar este segundo tipo, pero mapeando la misma implementación. Así, por cada tipo concreto de receptor de **_size**, la inline-cache de **_isSmall** agregaría una entrada, degenerando en una cache megamórfica y degradando consecuentemente la performance.

4.3.3. Nativizador de Métodos de tipo Template-JIT

El nativizador de métodos predeterminado decodifica los bytecodes del método iterándolos uno por uno hasta que haya generado código nativo para todo el método. Este nativizador evita hacer optimizaciones como transformaciones de código o asignación de registros. En cambio, aprovecha las optimizaciones realizadas a los bytecodes por etapas previas de precompilación, que pueden, por ejemplo, reordenar bytecodes para ahorrar operaciones de pila y copia de registros.

El Listado 4.4 muestra el algoritmo de nativización. Comienza en **translateMethod**, que genera código nativo para el método y sus bloques. Este proceso de traducción implica iterar sobre el flujo de bytecodes, traduciendo cada bytecode de a uno, con un nativizador de bytecodes especializado para cada tipo de bytecode. Un ejemplo de diferentes nativizadores de bytecodes se muestra en el Listado 4.5.

El caso de los bytecodes de send es el único que requiere un cuidado especial en comparación con Bee/SVM, debido a las semánticas de envío de mensajes añadidas en Bee/LMR. Como se muestra en el Listado 4.6, el nativizador para los bytecodes

Listado 4.4: Implementación del traductor de métodos bytecode-to-assembly de tipo template

```
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
```

Listado 4.5: Traductores para dos bytecodes diferentes: **LoadFalse** y **LoadInstanceVariable**.

```
LoadFalseBytecodeNativizer >> assemble
    assembler loadRwithFalse


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

Listado 4.6: Traducción del bytecode de send a código nativo.

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


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

de send delega la tarea de nativización al linker de mensajes correspondiente. La última etapa de este proceso se mostró en el Apartado 4.3.1.

Para mejorar el rendimiento, los bytecodes de envío para operaciones aritméticas y lógicas se optimizan generando inline-assembly para casos comunes como comparaciones de igualdad o suma de enteros. El Listado 4.7 muestra la traducción del envío de mensaje #= y las optimizaciones realizadas en esa traducción.

4.3.4. Nativizador de Métodos con Optimizaciones

Si bien el compilador JIT de tipo template genera código nativo de calidad suficiente para la mayoría de los escenarios, había casos en los que se deseaba un compilador más potente. En particular, el código Smalltalk que reemplazó a las funciones built-in de Bee/SVM se ejecuta intensivamente, por lo que pequeñas optimizaciones proporcionan grandes beneficios. Por esa razón, se implementó un compilador optimizador en Bee/LMR.

A diferencia del JIT de tipo template, el compilador optimizador genera una representación intermedia basada en grafos del código Smalltalk, siguiendo las mismas direcciones que [Cli93, BBZ11, DWS+13, LKH15]. Esta representación intermedia luego se optimiza y finalmente se genera código nativo a partir de ella.

La principal fortaleza de este optimizador es su capacidad para el inlining de métodos y block closures, realizando *value numbering* y llevando a cabo optimizaciones *peephole*. El compilador está diseñado para permitir más optimizaciones en el futuro. Particularmente relevantes serán aquellas relacionadas con la compilación dinámica, como recompilación adaptativa [HU94], escape analysis [PG92, SWM14],

Listado 4.7: El código nativo del bytecode de send para el mensaje #= está optimizado de dos maneras. Si tanto el receptor como el argumento apuntan a lo mismo, se devuelve **true** sin hacer ninguna llamada. Si esto no sucede, pero el receptor es un **SmallInteger**, se devuelve **false**, sin hacer una llamada. De lo contrario, se realiza una llamada normal al mecanismo de lookup (que incluye la utilización de 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.
```

y también otras relacionadas con compiladores en general, como loop invariant code motion.

El backend del compilador implementa un asignador de registros de tipo linear-scan basado en SSA [PS99, WF10].

Este compilador optimizador se utiliza solo de forma ahead-of-time, durante el bootstrapping, para optimizar un conjunto prefijado de métodos, la mayoría de ellos siendo los que reemplazaron las funciones built-in de Bee/SVM.

Sistema de Tipos del Compilador Optimizador

Los nodos en el grafo IR del compilador optimizador no almacenan información de tipos concretos, sino que asumen del tipo genérico **ProtoObject**. Esto significa que los cálculos numéricos intermedios no se realizan usando enteros nativos de la arquitectura de la CPU, sino usando objetos Smalltalk que representan enteros. Esto produce código menos eficiente, porque cada operación aritmética y lógica emitida incluye una conversión de **SmallInteger** a valor nativo y viceversa. Sin embargo, este diseño permite una conexión transparente con el recolector de basura. El código generado por el compilador siempre es seguro para el GC: no importa dónde interrumpa el GC al código optimizado, para el GC todos los valores en la pila y los registros son punteros a objetos que pueden atravesarse de manera segura.

Como el código Smalltalk no contiene anotaciones de tipo y el compilador optimizador se ejecuta a modo ahead-of-time, el mecanismo de inlining necesita un poco de ayuda para poder detectar qué métodos encontraría el algoritmo de lookup para cada envío de mensaje en el código. Dado que el conjunto de métodos a optimizar está prefijado y consiste en código muy estructurado (al final de cuentas, reemplaza código C++ de tipo estáticamente tipado), la información de tipos requerida por el compilador se proporciona manualmente mediante anotaciones de tipo.

4.3.5. Evitando la Invocación Recursiva del Nativizador

Como Bee/LMR no incluye ningún intérprete, todo el código Smalltalk que se ejecuta debe ser nativizado antes de su ejecución. El compilador JIT está escrito en Smalltalk y utiliza instancias de las clases estándar de Bee. Por lo tanto, para

realizar compilación JIT de código Smalltalk, el sistema necesita ejecutar código estándar Smalltalk, lo cual podría requerir invocar recursivamente al compilador JIT, causando un bucle infinito.

Para evitar que esa situación ocurra, los métodos del kernel de Bee/LMR se compilan de antemano durante el arranque (como se describe en el Apartado 4.5.1). El nativizador de Bee/LMR está escrito de manera que sólo necesite ejecutar métodos en el módulo del kernel, por lo que nunca desencadena una llamada recursiva a sí mismo mientras trabaja en un método. Cuando lxs desarrolladores modifican métodos del módulo del kernel, éstos se nativizan justo antes de ser instalados. El resto de los métodos en el sistema se compilan just-in-time. Esto sucede cada vez que el runtime detecta que un método que está a punto de ejecutarse aún no ha sido nativizado.

4.4. Implementación de las Funciones Built-in en Bee/LMR

Bee/SVM incluye alrededor de 200 primitivas que permiten acceder a las funciones built-in de la VM. En Bee/LMR no existen tales primitivas, sino que hay alrededor de 150 selectores especiales (que comienzan con _). Estos mensajes suelen tener implementaciones más cortas que las funciones built-in, están escritos en Smalltalk y pueden cambiarse dinámicamente.

Solo para dar un ejemplo, el Listado 4.8 muestra el cambio desde la implementación en Bee/SVM a la de Bee/LMR. Se observa cómo una primitiva que accede a la memoria directa puede implementarse en términos de los metamensajes **_byteAt:** y **_size**.

4.4.1. Lookup de Mensajes

Bee/LMR implementa el algoritmo típico de *mensaje lookup* de cualquier Smalltalk, recorriendo los diccionarios de métodos en la cadena de superclases hasta encontrar un implementador o enviando **#doesNotUnderstand:** en caso de que no se encuentre ninguno.

La implementación de este algoritmo es bastante simple, y su código se muestra

Listado 4.8: La implementación de **String»#byteAt:** en Bee/SVM (arriba) y en Bee/LMR (abajo).

```
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
```

en el Listado 4.9. El método **_lookup:** se invoca al despachar un mensaje. El resultado de la búsqueda puede ser cacheado, y el método **_lookup:** se invoca solo si la caché no contiene una entrada coincidente. El punto de entrada al algoritmo de dispatch se muestra en el Listado 4.10. Las estrategias de almacenamiento en caché se describen en el Apartado 4.4.2.

Después de que se completa la búsqueda[4], el algoritmo prepara el método encontrado para su ejecución, asegurando que contenga código nativo, generándolo si es necesario.

Las líneas al principio de **_dispatchSend:** actúan como un pragma para el nativizador, que genera código que espera que el argumento de este método esté en un registro especial en lugar de en la pila. Esto sólo es necesario para el código del lookup, que el nativizador linkea de una manera especial.

Como el algoritmo está escrito en Smalltalk, una ejecución naive del algoritmo de búsqueda requeriría llamar recursivamente al lookup para cada uno de los mensajes enviados, lo cual causaría un bucle de lookup infinito.

Para resolver este problema, el método **_dispatchSend:** se nativiza utilizando un **NativizationEnvironment** especial que está configurado para reemplazar los envíos de mensajes dinámicos por invocaciones estáticas de métodos. Este linkeo estático

[4]Si no se encuentra ningún método, entonces se devuelve **nil**, lo cual causa el envío del mensaje **doesNotUnderstand:**.

Listado 4.9: El algoritmo de lookup consiste en buscar un método que implemente el selector enviado en la cadena de diccionarios de métodos del objeto que recibe el mensaje. Esa cadena se conoce como *behavior*, y sigue la jerarquía de clases correspondiente.

```
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
```

Listado 4.10: El punto de entrada al mecanismo de dispach. El argumento es recibido en un registro en vez del stack. La implementación de **_cachedLookup:** y **when:use:** se muestra en el Apartado 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
```

es posible porque los métodos que deben ser invocados en cada envío de mensaje del algoritmo de lookup pueden ser precalculados. La clausura de los métodos involucrados en el algoritmo de lookup es un conjunto pequeño de métodos, por lo que no hay problema en calcular manualmente qué método se ejecutará en cada envío de mensaje.

4.4.2. Lookup Caching

En Bee/LMR, cada sitio de envío de mensaje está asociado a una instancia de **SendSite**. Los objetos **SendSite** están diseñados para permitir la implementación de mecanismos de lookup caching [CPL83, Ung83, DS84, Ung86]. El primer slot de los send sites se llama **dispatch**, y apunta al código nativo de un rutina de dispatch. La ejecución de un envío de mensaje se realiza apilando los argumentos en el stack, cargando la dirección del objeto send site en un registro, y finalmente llamando a la rutina almacenada en el slot de **dispach**. Esto se muestra en el Listado 4.11.

Cuando se instancia un send site, su slot **dispatch** se inicializa apuntándolo a la rutina de lookup genérica, **ProtoObject»#_dispatchSend:**. Durante la ejecución, la primera vez que se alcanza el send site, se invoca el código de **_dispatchSend:**. En lugar de realizar todo el proceso de lookup directamente, este método primero busca

Listado 4.11: El código máquina para un message send en la arquitectura AMD64. El registro RBX contiene el objeto **NativeCode** correspondiente al método actualmente en ejecución. Este a su vez contiene todos los objetos send site asociados al método.

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

en una dispatch-caché global, la cual es básicamente un arreglo de pares <**Símbolo, Clase**> que almacena el resultado del lookup para ese par. Si se encuentra, el lookup en los diccionarios de métodos se puede omitir, lo que resulta en una mejora de rendimiento. Si no se encuentra, el algoritmo realiza la búsqueda y almacena el resultado de la misma en la caché global. El algoritmo de almacenamiento en la caché global se muestra en el Listado 4.12.

El método **_dispatchSend:** también crea una inline-caché con el resultado de la búsqueda. El send site tiene una variable de instancia **cache** para almacenar esta información, que consiste en un arreglo de pares <**Símbolo, Clase**>. Después de agregar una entrada para el método encontrado para el envío actual, el algoritmo cambia el contenido del slot **dispatch**, para que ahora apunte a un código que busca en la inline cache antes de hacer el lookup en la cache global. Esto agiliza el lookup la siguiente vez que se utilice el mismo send site. La primera vez que se utiliza el send site, se crea una inline cache monomórfica, como se muestra en el Listado 4.13. El **monomorphicStub** asignado a la variable de instancia **dispatch** del send site se describe en el Listado 4.14.

Como se mencionó anteriormente, el mecanismo de envío de mensajes se compone de objetos comunes y corrientes: **SendSite** es una clase normal del sistema, la variable de instancia **dispatch** apunta simplemente a un objeto de tipo **ByteArray**, y **cache** a un **Array**. Cuando un caché monomórfico falla, el mecanismo de envío reemplaza el array del caché monomórfico con otro mas grande, polimórfico. El algoritmo de inline caching polimórfico se muestra en el Listado 4.15. El código

Listado 4.12: Algoritmo de lookup caching global de 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
```

Listado 4.13: Algoritmos de inline caching monomórfico de 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
```

Listado 4.14: El código ensamblador del stub monomórfico. Inicialmente, RAX, RDX y R15 contienen al receptor, send site y arreglo de globales, respectivamente. El stub carga la caché del send site en RCX, luego el behavior del objeto en RSI, luego realiza la prueba y, en caso de éxito, salta al código nativo del método que está siendo despachado. En caso de falla, el stub de dispatch carga el código nativo de **_dispatchSend:** desde el arreglo de globales (R15) y salta a su código nativo.

```
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]
```

Listado 4.15: Algoritmos de inline caching polimórfico de 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]
```

assembly correspondiente es similar al del cache monomórfico, pero con más casos.

4.5. Integrando los componentes de la LMR en Bee

4.5.1. Bootstrapping

El proceso de crear un entorno de ejecución que a su vez pueda ejecutarse, a partir de código fuente escrito en el mismo lenguaje que el entorno de ejecución, se conoce como *bootstrapping*. Puede representar un problema cuando el lenguaje carece de un sistema de ejecución ya funcionando[5], ya que no sería posible ejecutar el código que genera el ejecutable. Afortunadamente, en el caso de Bee/LMR contamos con Bee/SVM, que por supuesto cuenta con compiladores y ensambladores C++ ya bootstrappeados. Por lo tanto, fue posible generar la imagen del kernel de Bee/LMR desde el sistema en ejecución sobre Bee/SVM, colocando en su interior el código nativo de todos los métodos que pertenecen al módulo del kernel.

Durante la creación de esta imagen, se eliminan las cosas que hacen referencia a Bee/SVM. Los métodos que hacen referencia a primitivas, o que acceden al estado

[5]Esto suele suceder cuando el lenguaje está en proceso de implementación por primera vez.

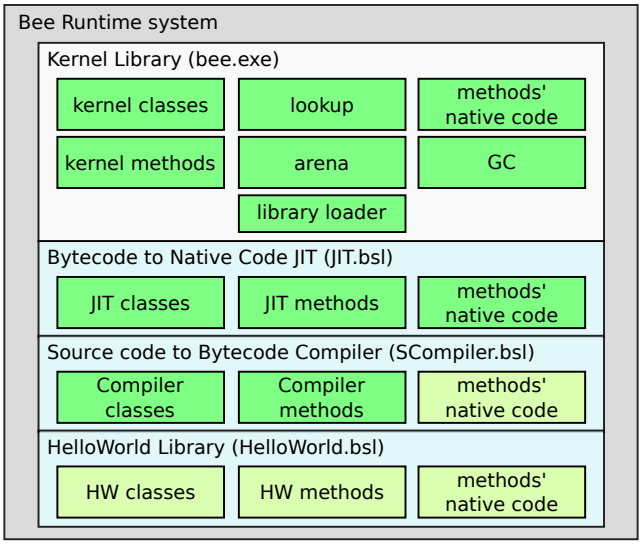


Figura 4.2: Módulos de Bee. La biblioteca del kernel contiene funcionalidad mínima para soportarse a sí misma y cargar otras bibliotecas. La carga del nativizador JIT y compiladores de bytecode es opcional. Excepto por el kernel y el JIT, las bibliotecas pueden ser enviadas con sólo código fuente, con bytecodes precompilados o con código nativo.

y código de la VM host como espacios de memoria, lookup, GC y el cargador de librerías se reemplazan con sus versiones complementarias implementadas en Smalltalk. El JIT se implementa como un módulo separado, que se escribe utilizando los mismos mecanismos que el módulo del kernel, y que se puede cargar cuando sea necesario. Se muestra una vista de alto nivel del ecosistema en la Figura 4.2.

4.5.2. Entorno de Desarrollo de Bee/LMR

El entorno de desarrollo utilizado para implementar Bee/LMR es el mismo que se usa para aplicaciones normales en Bee, que es un Smalltalk tradicional. Antes de que Bee/LMR estuviera completamente funcional, se utilizaba Bee/SVM para ejecutar el ambiente. Bee/SVM también es útil para implementar cambios en partes críticas del sistema, si lxs desarrolladores desean evitar fallas del sistema en esas partes en que están trabajando.

4.6. Trabajo Relacionado

Varios proyectos han experimentado con cambios y variaciones de la arquitectura de dos capas.

4.6.1. Máquinas Virtuales Self-Hosted

La torre de intérpretes de Lisp [Smi84] y el Smalltalk self-hosted Squeak [IKM+97] son ejemplos clásicos, que en cierto grado logran el objetivo de implementar un lenguaje en sí mismo. Ejemplos más recientes de este enfoque incluyen PyPy [RP06], Rubinius [FM08], JikesRVM [AAB+05], Squawk [SC05], Maxine [WHVDV+13], y SubstrateVM [WSH+19].

Las VMs self-hosted están escritas mayormente en el mismo lenguaje que soportan o en un subconjunto de este. Mientras que la torre de intérpretes de Lisp tiene la capacidad de cambiar el lenguaje en cada nivel, el enfoque más ampliamente utilizado para las VMs puede observarse con OpenSmalltalkVM para Squeak y PyPy, que se traducen a C y luego se compilan estáticamente con un compilador C estándar. JikesRVM, Maxine y SubstrateVM utilizan una VM de bootstrap para generar directamente una imagen ejecutable nativa en lugar de código C.

Este enfoque de generar código C da cierta flexibilidad y, por ejemplo, permite a Squeak proporcionar herramientas de simulación que pueden ejecutar el código de la VM de la misma forma que el código Smalltalk normal, lo que a su vez hace posible usar las mismas herramientas de desarrollo para programar la VM como si fuera cualquier otra aplicación. Sin embargo, esta simulación a menudo es alrededor de 1000 veces más lenta que la VM real [BKL+08]. De manera similar, la implementación de PyPy puede ejecutarse como código Python normal y lxs desarrolladores pueden usar sus herramientas estándar de Python para trabajar con ella.

El mayor inconveniente es el bajo rendimiento de ejecutar una VM metacircular sobre sí misma, que no está optimizada y no está destinada a ser utilizada en producción por programadores de aplicaciones. Los intérpretes de AST autooptimizantes [HWW+15, WWS+12] proponen reutilizar la infraestructura de VMs existente, combinando un compilador JIT de Java, que está escrito en Java, e intérpretes de AST, también escritos en Java. Muestran cómo el código de soporte

para nuevos lenguajes puede reutilizar fácilmente los componentes de optimización de otra VM host para ser eficiente.

Además, estos enfoques aún mantienen la arquitectura de dos capas discutida anteriormente, que separa claramente las aplicaciones de la VM.

Ilustramos estos diseños en la Figura 4.3. En la parte superior, Figura 4.3a muestra la arquitectura de dos capas, mientras que a continuación la Figura 4.3b representa las VMs self-hosted que pueden ejecutar el código de la VM como si fueran otra aplicación.

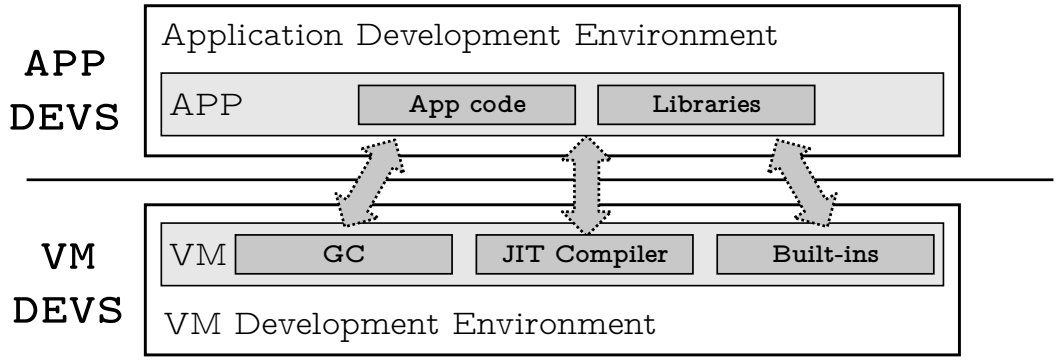
4.6.2. Diseños con Protocolos de Metaobjeto Extensibles

Aunque aún mantienen una arquitectura de dos capas, los Entornos de Ejecución Totalmente Reflexivos (FREE) [CGMD15, CGM16] buscan ampliar las APIs proporcionadas por las VMs para permitir la personalización de idealmente todos los aspectos de una VM. Esto se logra mediante el diseño de protocolos de metaobjeto que pueden cambiar, por ejemplo, cómo se realiza la búsqueda de métodos, el acceso a los campos, e incluso cómo se realiza la recolección de basura y la compilación JIT. Sin embargo, como se ilustra en la Figura 4.3c, esto aún mantiene la distinción arquitectónica y no brinda a lxs desarrolladores acceso a la implementación de la VM en sí misma, solo proporciona más capacidades de personalización. Pinocchio [VBG+10] es otro ejemplo de una VM de Smalltalk donde el propio intérprete puede adaptarse mediante un protocolo de metaobjeto.

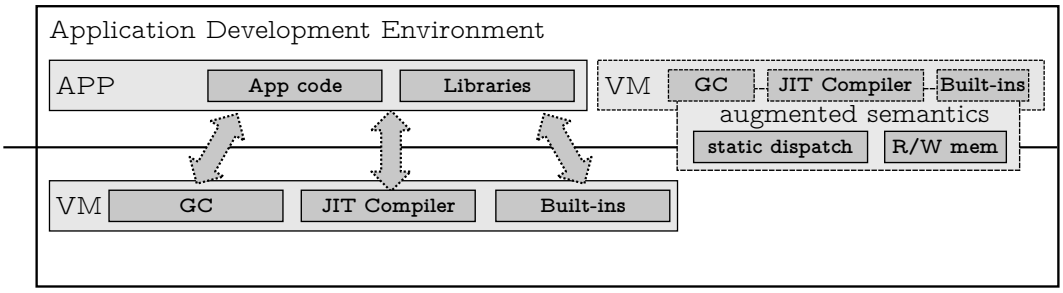
En los diseños de dos capas, aunque teóricamente sería posible actualizar partes del código de la VM mientras se ejecuta, en la práctica ningún sistema está realmente destinado a ser programable en vivo en entornos de producción.

4.6.3. Más allá de las Máquinas Virtuales Orientadas a Objetos

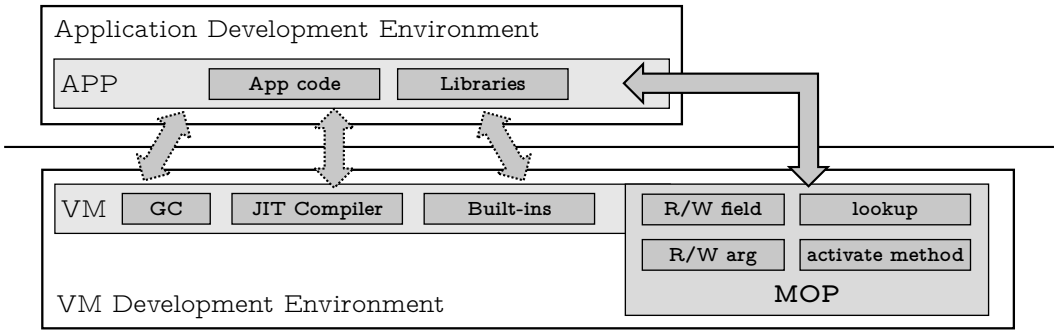
Fuera del mundo de las máquinas virtuales orientadas a objetos, proyectos como SML# [OUH+14] tienen como objetivo combinar la aplicación y el entorno de ejecución mediante la compilación de ambos en un binario estático. Sin embargo, aunque el producto final sea una combinación, SML# todavía separa la máquina virtual de la aplicación.



(a) Arquitectura de dos capas de las VMs tradicionales, separando claramente la aplicación y la VM, con APIs que limitan la interacción.



(b) Las VMs self-hosted mantienen la arquitectura de capas, pero pueden permitir a lxs desarrolladores ejecutar la VM como si fuera una aplicación.



(c) VMs con Protocolos de Metaobjeto para personalizar el acceso a campos, la búsqueda de métodos y otros aspectos desde una aplicación.

Figura 4.3: Enfoques de implementación de VMs con el diseño tradicional de dos capas.

Para aplicaciones generales, Kiczales [Kic96] propuso la noción de *Open Implementations*. Además de una interfaz estándar, se espera que un módulo proporcione una meta-interfaz. Esta meta-interfaz luego se puede usar para controlar y adaptar la implementación de un módulo, quizás para seleccionar un algoritmo que tenga un mejor rendimiento en un contexto específico. Esta idea se basa en la noción de protocolos de metaobjeto [KDRB91] y metaarquitecturas, en el contexto del cual la investigación exploró una amplia gama de problemas y diseños [Tan09], incluyendo también protocolos de metaobjeto en tiempo de compilación [Chi95].

Sin embargo, estos enfoques son sobre los que se basan los Entornos de Ejecución Totalmente Reflectivos [CGMD15], que aún dependen de una meta-interfaz fija, y por lo tanto, imponen un límite fuerte entre módulos, o en nuestro caso, entre la VM y la aplicación.

4.6.4. Diseños con Límites Reducidos entre la VM y la Aplicación

Klein [USA05] empujó un poco más el límite. Era una VM self-hosted escrita en Self para Self, que era orientada a objetos, metacircular y reactiva, lo que significa que podía cambiarse mientras se ejecutaba. En nuestra comprensión, difuminó la línea entre la VM y la aplicación mucho más que cualquier otro enfoque. Por lo tanto, es una inspiración para nuestro trabajo, un primer intento de crear una VM que no separa estrictamente la aplicación y que puede evolucionar al igual que una aplicación. Desafortunadamente, Klein no implementó un GC self-hosted ni logró la funcionalidad y el rendimiento completos de Self.

Otro punto de inspiración es el proyecto Mist, [6], que se explica como *un Smalltalk sin una VM*. Según nuestro entendimiento, también tenía como objetivo ser una implementación de Smalltalk self-hosted que se compila a sí misma en código nativo, pero aún está en una etapa inicial de ideas.

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

Capítulo 5

Recolección de Basura y Gestión de Memoria

En Bee/SVM el administrador de memoria es invisible para la aplicación. La interfaz para accederlo desde la aplicación es mínima, permitiendo sólo activar pasadas de GC mayores y preguntar sobre el tamaño del heap.

Por otro lado, en Bee/LMR el modelo del recolector de basura se coloca a nivel del lenguaje y se modela como cualquier otra entidad del sistema que tenga forma y comportamiento: con objetos.

5.1. Desafíos de Implementación

En las LMRs, el heap de objetos es gestionado por un conjunto de objetos programables en vivo, y no hay una interfaz de bajo nivel para las funcionalidades del recolector de basura, sino sólo un modelo unificado de alto nivel donde *todo* es un objeto. Al momento de implementación esta unificación planteó una serie de desafíos que tuvieron que ser abordados para crear una solución funcional.

5.1.1. Recolección de Basura de Objetos del Entorno de Ejecución

Un primer desafío es que las LMRs añaden las entidades del entorno de ejecución al conjunto de objetos ya existentes en los entornos metacirculares. Esto incluye espacios de asignación, pilas, arrays, memoria e incluso el recolector de basura mismo y sus objetos, que están representados por simples instancias de clases.

Así, al rastrear punteros desde las *roots* de un programa, el GC se encontrará a sí mismo y a todos los demás objetos necesarios para su propia implementación. Ejemplos de estos objetos son la instancia del GC en sí, su clase, métodos, espacios e incluso las pilas. La clausura transitiva de estos objetos también es rastreada.[1]

El recolector de basura necesita determinar qué objetos deben ser seguidos y cuáles no. Los objetos creados temporalmente por el recolector no deben quedar asignados consumiendo memoria después de que la recolección termine y, a diferencia de las VMs clásicas, el entorno de ejecución necesita que sus propios objetos sean recolectados de vez en cuando.

5.1.2. Indisponibilidad del Contexto de Ejecución Durante la Recolección de Basura

Un segundo problema surge en las LMRs también como consecuencia de tener un paradigma común para los objetos de nivel de VM y de aplicación. Debido al diseño de capa dividida, los recolectores de basura típicos, incluso en implementaciones self-hosted como las de Squeak o PyPy, se trabaja dentro de un entorno de ejecución separado, que está desconectado del entorno que al que se está efectuando la recolección.

En esos sistemas, los métodos y clases de esos recolectores de basura pertenecen a una capa separada, que es inaccesible mientras se recorren los roots del espacio recopilado. Dentro del heap, el algoritmo de GC puede cambiar libremente los punteros a objetos según sea necesario para ajustar las referencias. Sin embargo, por otro lado, el GC de las LMRs está compuesto por objetos estándar que viven en la misma memoria que la aplicación.

[1] Por clausura transitiva de un conjunto de objetos nos referimos a todos los objetos que son alcanzables directa o indirectamente desde los objetos root.

Estos objetos no solo viven en el mismo heap que los objetos de nivel de aplicación, sino que son indistinguibles de otros tipos de objetos. Por ejemplo, las clases de la aplicación heredan directamente de la misma clase **Object** de la que la clase **GarbageCollector** hereda. El código nativo también se representa mediante objetos simples. A menos que se trate de forma especial, el código puede ser movido por el GC a través de la memoria como cualquier otro objeto. Esto significa que en algún momento durante el GC, diferentes copias del código del recolector de basura pueden estar vivas y ser alcanzables.

Por las razones descritas anteriormente, una implementación naive de un GC para LMRs no puede pausar toda la ejecución de alto nivel, ya que su propio código está escrito a en alto nivel. Lxs implementadores de LMRs tienen que enfrentarse al problema de que los algoritmos comunes de recolección de basura dejan objetos en un estado inconsistente mientras el recolector de basura está en funcionamiento. Los headers y punteros de los objetos se manipulan para detectar objetos vivos y reorganizar referencias, lo que significa que en etapas intermedias de la recolección los objetos del heap no pueden recibir mensajes.

Entonces, el recolector de basura necesita estar completamente desconectado de alguna manera del entorno de ejecución recopilado, o ser modificado de manera que sus objetos permanezcan operativos durante su trabajo. Dos ejemplos específicos de este problema se describen a continuación: *forwarding pointers* en recolectores de copia y *pointer reversal* en algoritmos de *mark and sweeep* [JHM11].

Dado que el GC es un componente crucial que necesita funcionar en todo momento, necesitamos asegurar *objetos siempre activos* y prevenir *escrituras perdidas* y *objetos perdidos*. Esto restringe las opciones de diseño para los algoritmos de GC, pero nos permite construir un sistema confiable.

Objetos siempre activos

Dado que el código del GC consta de la interacción de objetos comunes, durante el GC no es posible aplicar algoritmos de GC que sobrescriban temporalmente el heap donde se almacenan los datos de los objetos, como aquellos algoritmos que instalan forwarding pointers en los headers de los objetos, o que realicen pointer threading.

Forwarding [FY69] es una técnica de GC que sobrescribe el header original de

los objetos movidos con forwarding pointers a sus nuevas ubicaciones. *Pointer reversal* [SW67] es una técnica utilizada para eliminar la necesidad de memoria adicional para una pila de trazado, utilizando los slots los objetos para implementar una estructura de lista enlazada funcionalmente similar a una pila. En el algoritmo de mark and sweep, se realiza *pointer threading* [Mor78, Jon79] para encontrar eficientemente todas las referencias a cada objeto movido.

Con tales algoritmos, si el GC intentara enviar un mensaje a un objeto sobrescrito temporalmente, eso podría causar comportamiento indefinido porque no se encontraría la información esperada del objeto en su header, como su clase o su tamaño.

Pérdida de Escrituras

El recolector de basura debe evitar mover objetos que puedan cambiar durante la recolección de basura. Durante el GC, los objetos suelen moverse (copiarse) a otra área cuando se alcanzan por primera vez al rastrear el grafo de objetos. A medida que el GC continúa el tracing, puede encontrar más referencias a los objetos ya movidos y actualizarlos con las nuevas direcciones.

Si el GC modificara un objeto en movimiento, podría causar inconsistencias porque puede ser imposible determinar si un objeto es la nueva copia o la antigua, y durante el GC algunos objetos aún podrían ver la versión original en lugar de la modificada, cuando aún no han sido rastreados.

Objetos Perdidos

El recolector de basura cambia el área de asignación al comenzar cada recolección, de modo que los objetos nuevos necesarios para esa recolección de basura no se mezclen con el resto y se descarten tan pronto como la recolección de basura termine. Esto significa que ningún objeto nuevo creado durante la recolección de basura sobrevive después de que la recolección de basura finalice.

En particular, no se permite nativizar mediante el JIT durante la recolección de basura por esta razón. El sistema debe asegurar que cualquier código que pueda ejecutarse durante la recolección de basura haya sido nativizado previamente a que la recolección de basura comience.

5.1.3. Depuración del Recolector de Basura

Como parte de un sistema metacircular vivo, es deseable que el recolector de basura pueda ser depurado, al menos parcialmente, dentro del entorno de programación estándar con herramientas estándar. Pero esto plantea otra situación de "huevo y gallina": en entornos metacirculares vivos, las herramientas estándar se implementan *dentro* del lenguaje y requieren una interfaz de usuario completamente funcional que maneje eventos; pero los recolectores de basura de tipo stop-the-world requieren pausar la ejecución del programa de alto nivel, incluida la interfaz de usuario. Una solución a este problema se presenta en el Capítulo 6, mediante el uso de un depurador out-of-process.

5.2. Direcciones de Diseño

Uno de los desafíos más difíciles al implementar Bee/LMR fue el recolector de basura. Como se explicó anteriormente, el recolector de basura necesita ser capaz de recorrerse a sí mismo, dejar objetos operativos mientras se ejecuta y ser depurado con herramientas estándar.

5.2.1. Dos Enfoques Posibles

Descubrimos dos enfoques posibles para alcanzar nuestro objetivo. El primero es utilizar un algoritmo estándar de recolección de basura y desconectar su entorno de ejecución del espacio que será recolectado. Esto evita el problema de que el GC se recolecte a sí mismo, es decir, de recorrer las estructuras de datos que utiliza, y también el problema de exponer un estado inconsistente de objetos. Este enfoque requiere generar una clausura de código de todos los métodos y objetos involucrados en el GC, y compilarlos en un heap separado. Como el sistema es de tipado dinámico, este es un paso engorroso, porque la falta de anotaciones de tipo dificulta encontrar dicha clausura.

Un segundo enfoque es modificar los algoritmos existentes de GC para que los objetos estén en un estado consistente durante todo el proceso de recolección de basura. Esto significa que se restringe el diseño del GC más que con el primer enfoque, lo cual podría tener implicaciones de rendimiento, pero evita tener que

determinar una clausura de código para el GC.

Implementamos ambos enfoques. Describimos brevemente el primero a continuación, para luego explicar por qué decidimos mantener sólo el segundo, que se describe en detalle en la siguiente sección.

Un Recolector de Basura en una Botella

Utilizando el primer enfoque, implementamos un recolector de basura de tipo mark-and-sweep para el espacio antiguo y un scavenger generacional [Che70] para los objetos nuevos. Generamos estáticamente una clausura del código del GC y los objetos requeridos para su funcionamiento, seleccionando manualmente los métodos que van en la clausura. Esta clausura se coloca dentro de una biblioteca de enlace dinámico (DLL) que puede regenerarse sobre la marcha y reemplazarse tantas veces como sea necesario en tiempo de ejecución.

La clausura se llena con una copia de todos los objetos relacionados con el recolector de basura: clases, métodos, código nativo, etc. Además, el recolector utiliza un espacio separado para los nuevos objetos que crea, el cual se descarta después de que finaliza la recolección. Esto hace que el entorno de ejecución del GC sea independiente de los objetos originales. El beneficio es que objetos importantes, como por ejemplo la clase **Object** del heap que se recolectará, pueden dejarse en un estado inconsistente durante la recolección, ya que la biblioteca de GC utiliza su propia copia de ese mismo objeto.

Posteriormente, se implementaron otras variaciones de algoritmos de GC ajustando las estructuras internas de los recolectores generacionales y de marcado y compactación, e incluso se implementó una variación multithreaded del de mark-and-sweep.

Para el desarrollo, pudimos utilizar nuestras herramientas estándar de alto nivel para analizar y depurar los algoritmos. Por ejemplo, implementamos un conjunto de alrededor de cien tests de GC con herramientas de alto nivel, donde se generan dinámicamente espacios externos con diferentes objetos en su interior. Los recolectores se pueden depurar mientras atraviesan estos espacios para verificar qué objetos están marcados, recolectados y copiados. Para depurar la integración una vez creada una DLL, primero tuvimos que recurrir a herramientas de bajo nivel, sin embargo, ese problema se resolvió mediante el diseño de las herramientas

descritas en la Capítulo 6.

El mantenimiento de la clausura de métodos y objetos involucrados en el GC resultó ser una tarea manual que consume mucho tiempo, porque esta clausura consiste en cientos de métodos. Si bien el primer enfoque de GC es potente y permite algoritmos más eficientes que el segundo pues no tiene las limitaciones antes mencionadas, decidimos dejar de utilizarlo hasta que implementemos un sistema de inferencia de tipos capaz de encontrar mayormente de forma automática la clausura de código del GC. Esto fue dejado como trabajo futuro.

5.3. Diseño General de los Recolectores de Basura de Bee/LMR

Esta sección detalla la implementación del segundo enfoque de GC, un recolector de basura que puede recorrerse a sí mismo.

El punto de acceso principal a los objetos encargados del manejo de la memoria es una instancia de la clase **Memory**. Este objeto proporciona una API para asignar memoria para nuevos objetos, para iterar a través de los objetos de las diferentes partes del heap de objetos, para recopilar datos de uso del heap y para liberar memoria a través de un par de recolectores de basura generacionales y del old space.

El objeto que reifica la memoria también implementa operaciones para encontrar referencias a objetos en el heap y el stack, y para convertir referencias a un objeto en referencias a otro, es decir, para implementar *become*.

Para lograr que el entorno de ejecución se mantuviera funcional durante todo el proceso de recolección de basura, cambiamos del algoritmo de mark-and-sweep con pointer threading del espacio antiguo que había sido implementado en el primer enfoque, a un recolector de copia estándar. Esto tiene un impacto en el uso de memoria, pero por sus características los recolectores de copia facilitan mantener los objetos en estado operativo durante la recolección. Específicamente, el principal cambio respecto a las implementaciones tradicionales fue la utilización de tablas de forwarding externas, para que durante la recolección la única modificación a los objetos en el heap sea el bit de marcado. Esta técnica de forwarding tables externas también se utilizó para el GC generacional.

5.3.1. Distribución del Heap de Objetos

El espacio de direcciones del proceso se divide en fragmentos de memoria llamados **GCSpaces**. Cada espacio representa un rango de direcciones virtuales asignadas por el sistema operativo. La memoria de cada espacio puede estar o bien reservada o bien entregada al proceso.

Una pequeña porción del heap se define como un área joven, delimitada por **youngBase** y **youngLimit**; el resto se considera espacio antiguo. El área joven consiste de un espacio *eden* y dos espacios más pequeños *from* y *to*. Dentro del área antigua, coexisten espacios para objetos en direcciones fijas, objetos grandes (también fijados), segmentos de imagen y una zona de asignación para objetos de edad madura, que es gestionada por el recolector descrito en el Apartado 5.5.

La zona de asignación antigua se divide en fragmentos de igual tamaño que pueden asignarse y desasignarse libremente a medida que se crean, mueven y liberan objetos.

5.3.2. Asignación de Memoria para Objetos

Los objetos se crean utilizando *bump-pointer allocation*, como se muestra en el Listado 5.1. El *fast-path* de este asignador aumenta el puntero **nextFree** del espacio eden y retorna la dirección del buffer obtenido, si no se excedió los límites del espacio.

Cuando un intento de asignación falla, puede significar que el espacio eden está lleno o que el objeto que se está asignando es demasiado grande. En este último caso, el asignador crea un nuevo espacio grande para asignar sólo ese objeto; en el primer caso, puede activar un paso de recolección de basura joven (si el GC está habilitado) y luego volver a intentarlo utilizando un *slow-path allocator*. Este segundo asignador, como último recurso, entrega la memoria de la zona de asignación antigua.

5.4. Recolector de Basura Generacional

Para los objetos recién creados, Bee/LMR implementa un recolector de basura generacional de copia [Che70, Ung84, Ung86]. Los nuevos objetos se crean en el espacio *eden*. Cuando se intenta asignar memoria y no queda espacio, se realiza una

Listado 5.1: El mecanismo de asignación intenta simplemente aumentar el puntero **nextFree** del espacio **edenSpace**. Si eso falla, la vía lenta verifica los objetos grandes, activa el GC si es necesario y, finalmente, intenta asignar nuevamente.

```
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
```

recolección menor. Los sobrevivientes de los espacios *eden* y *from* se mueven al espacio *to*, que se intercambia con *from* cuando la recolección finaliza. Los objetos que sobreviven a una segunda recolección menor, pasando de *from* a *to*, se marcan, para ser *tenured* si sobreviven a otra recolección menor.

Después de que la recolección finaliza, el objeto **memory** analiza las estadísticas de supervivencia y ajusta el tamaño de *eden*, como se muestra en el Listado 5.2. La heurística de ajuste intenta mantener un tamaño de *eden* pequeño mientras la tasa de supervivencia se mantenga baja (menos del 8%). Se prefiere un *eden* más pequeño porque aumenta las posibilidades de mantener la memoria utilizada para la asignación en la caché del procesador. Un aumento en la tasa de supervivencia es una señal de que la hipótesis generacional no se está cumpliendo. Por esa razón, cuando cruza el 8%, la política aumenta rápidamente el tamaño del espacio *eden* para darles a los objetos más tiempo entre recolecciones menores, aumentando las posibilidades de disminuir nuevamente la tasa de supervivencia.

Listado 5.2: Ajuste del tamaño del *eden* de acuerdo al ratio de supervivencia.

```
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]
```

5.5. Recolector de basura del espacio antiguo

Para el heap de espacio antiguo, Bee/LMR implementa un recolector de basura oportunista con evacuación de una sola pasada y basado en regiones, en la tradición de los recolectores de basura *garbage-first* G1 [DFHP04, ZB20] e Immix [BM08].

Al igual que G1 e Immix, en Bee/LMR el heap de espacio antiguo se divide en múltiples regiones de igual tamaño. El tamaño de región predeterminado en Bee/LMR es de 256 KB y el número de regiones crece a medida que aumenta el consumo de memoria del proceso. Para defragmentar el heap, el algoritmo G1 incluye un paso de compactación. Después de que se completa la fase de marcado inicial, se selecciona un grupo de regiones para evacuar según su tasa de ocupación. Los objetos vivos almacenados en regiones con menor ocupación se mueven a regiones diferentes donde se asignan uno tras otro de forma compacta.

5.5.1. Mecanismos de Compactación

Para realizar la compactación, el recolector necesita tanto mover los objetos de las regiones evacuadas como actualizar las referencias a los objetos movidos desde los slots de los objetos que los referencian. Los punteros que deben actualizarse

podrían estar en cualquier lugar del heap, por lo que un mecanismo de actualización naive requeriría un barrido completo de toda la memoria, lo que sería ineficiente.

En la actualidad existen múltiples algoritmos que ayudan a optimizar esto. Por ejemplo, para permitir una actualización eficiente de punteros, G1 mantiene un *remembered set* individual para los punteros entrantes de cada región. Por lo tanto, para actualizar las referencias a los objetos evacuados, solo tiene que recorrer la lista de punteros entrantes del conjunto correspondiente a las regiones evacuadas.

El GC Immix, en cambio, evita la fragmentación de manera diferente, utilizando dos mecanismos. Por un lado, mantiene un conteo de la tasa de ocupación de cada bloque y, cuando está por debajo de un umbral predefinido, marca el bloque como *reciclable*. Después de marcar el grafo de objetos, recorre todos los bloques reciclables para identificar sus regiones libres. Como hacerlo en un esquema de palabra por palabra sería muy costoso, en su lugar divide los bloques en líneas y mantiene bits de marcado para cada línea, para que el barrido de bloques se pueda hacer rápidamente.

Por otro lado, Immix evita la fragmentación aplicando una desfragmentación oportunista. *Antes* de comenzar la fase de marcado, Immix selecciona bloques para evacuar. Como los bloques están preseleccionados antes de que comience el marcado, el algoritmo de trazado puede determinar si un objeto se está moviendo o no en el momento en que encuentra una referencia desde otro objeto. Esto permite actualizar de manera oportunista las referencias a medida que se está trazando el heap de objetos.

El GC del heap de objetos de espacio antiguo de Bee/LMR implementa un enfoque de *garbage-first* con defragmentación oportunista únicamente, el segundo mecanismo de Immix. Antes de trazar el heap de objetos, el GC selecciona las regiones a desfragmentar en función de la tasa de ocupación calculada en el GC anterior. Durante el trazado, actualiza los punteros a los objetos evacuados al mismo tiempo que realiza la evacuación en sí.

Por simplicidad y rendimiento en tiempo de ejecución, no hay un barrido separado de líneas ni reciclaje de espacios fragmentados; los espacios son evacuados y liberados, o no liberados en absoluto. El costo de esto es un aumento en la fragmentación interna y en la basura flotante: el espacio ocupado por objetos que se encuentran inalcanzables no se recupera hasta un ciclo de GC sucesivo, en el

que la ocupación de la región cae por debajo de una relación predeterminada.

Los forwaring pointers del GC para el heap de objetos antiguos se almacenan a un offset constante desde el objeto al que se le está haciendo forwarding, en un espacio de direcciones reservado justo después de la región del objeto. Esto reduce efectivamente a la mitad el espacio de direcciones disponible para el proceso, lo cual no es un problema porque Bee/LMR está implementado para espacios de direcciones de 64 bits.

5.6. Trabajo Relacionado

El objetivo de esta investigación usando Bee/LMR no fue crear nuevos algoritmos de GC, sino de aplicar técnicas estándar, con modificaciones si fuera necesario, para implementar un sistema que funcione.

El problema de la recolección de basura de entornos de ejecución self-hosted ha sido estudiado previamente. Una de las diferencias principales entre Bee/LMR y otros sistemas es la unificación total de las capas de aplicación y del entorno de ejecución, incluído el código de la VM y los heaps de objetos, aunque este no siempre es el caso, como se muestra a continuación.

Squeak [IKM+97] y PyPy [RP06] son ejemplos de implementaciones self-hosted que no unifican el heap de aplicación con el de la VM, ya que el código de esta última es traducido a C. De esta manera, la VM maneja el heap de objetos mientras que el runtime de C maneja el heap de la VM, que utiliza **malloc** principalmente.

Truffle/Graal [WW12, WWW+13b], otro entorno de ejecución self-hosted, no implementa un GC propio. En cambio, aprovecha que corre sobre otra VM, HotSpot, la cual está escrita en C++, y utiliza el GC de esta VM host, la cual a su vez sí unifica los heaps de aplicación y entorno de ejecución de Truffle.

Otras VMs de Java escritas en Java como Maxine [WHVDV+13] y Jikes [AAB+05], si bien para lxs programadores también dividen la capa de la aplicación de la capa de la VM, internamente modelan los conceptos de la VM con objetos e implementan un GC que recorre esos objetos, los cuales desde el punto de vista del GC son en principio indistinguibles. Sin embargo, como el código de la VM no puede ser modificado en tiempo de ejecución y a su vez consta de anotaciones de tipos, los objetos de la VM pueden ser determinados automáticamente con antelación y

fijados en memoria donde no requieren moverse.

Tachyon [CBLFD11] y Klein [USA05] no proveen un GC.

Capítulo 6

Metaphysics

Un Framework para Trabajar Remotamente con Objetos y Entornos Potencialmente Dañados

Uno de los principales objetivos al diseñar el concepto de LMR fue hacer más práctico el desarrollo de VMs. Para lograrlo, en lugar de manipular archivos, se ejecuta el sistema y se modifican sus métodos, clases y, más generalmente, objetos. Como manipular objetos del kernel puede causar errores fatales, experimentamos con la adaptación de herramientas de programación viva para hacerlas más seguras para el desarrollo de los componentes principales de las LMRs. Además, las hicimos robustas para que pudieran trabajar en las LMR cuando estas sufrieran errores fatales, y para permitir la depuración de sistemas que no estaban completamente funcionales.

Una herramienta adicional fue implementada para el proyecto Bee/LMR: un depurador *out-of-process*. Este debugger permite a lxs desarrolladores inspeccionar el estado de un proceso Bee/LMR congelado, para depurar crashes y otros fallos. La herramienta reutiliza los componentes de la interfaz de usuario del resto del entorno de desarrollo. Sin embargo, requirió la implementación de un framework, llamado Metaphysics, [PM17] que proporciona acceso a objetos y memoria en el proceso remoto.

Con Metaphysics se hizo posible crear herramientas de depuración y profiling de código nativo. Estas nuevas herramientas aprovechan al máximo la metacircularidad

de Bee/LMR y habilitan un flujo de trabajo dinámico y rápido de *edición-prueba* como el que estamos acostumbrados a utilizar al desarrollar código de nivel de aplicación.

6.1. Contexto

En las LMRs, los componentes vitales del sistema se exponen a lxs programadores a través de los inspectores normales, depuradores y navegadores de código, que se ejecutan dentro del mismo entorno. Esos componentes pueden ser cambiados rápidamente, proporcionando feedback instantáneo. Sin embargo, al cambiar componentes centrales, lxs programadores se enfrentan con frecuencia al problema de que una pequeña modificación incorrecta provoca el bloqueo de todo el sistema dejando herramientas de alto nivel sin responder.

En las primeras etapas del desarrollo de Bee/LMR, utilizamos herramientas estándar de bajo nivel como GDB e IDA Pro para depurar tales situaciones. Si bien estas herramientas son personalizables y pueden funcionar con la meta información proporcionada por Bee, no proporcionan la interactividad fluida esperada por lxs programadores de Smalltalk. Además, lidiar con metadatos de estas herramientas requiere escribir scripts que duplican la lógica ya presente para herramientas dentro del sistema en ejecución.

Para evitar estos problemas, creamos un nuevo conjunto de herramientas que enriquecen el entorno de programación. A diferencia del conjunto de herramientas de Smalltalk existentes, como inspectores, navegadores y depuradores, las nuevas herramientas están diseñadas para comunicarse con una imagen de Smalltalk remota que se ejecuta en un proceso diferente del sistema operativo, lo que permite a lxs desarrolladores trabajar con objetos en un entorno remoto que puede estar congelado para su depuración. El proceso podría haber sido pausado por el sistema operativo porque ejecutó una operación no válida, o por lxs desarrolladores con fines de depuración. En cualquier caso, el proceso sigue vivo, su memoria se puede leer o escribir y, en ocasiones, luego de modificarla puede permitirse que continúe ejecutándose.

Para estas nuevas herramientas desarrollamos el framework *Metaphysics*, inspirado en ideas de [BU04]. Este framework permite acceder a objetos remotos

proporcionando *mirrors* para la reflexión estructural y proxies para la intercesión comportamental, lo que hace posible adaptar el envío de mensajes, es decir, la semántica de invocación de métodos, en función de situaciones específicas.

6.2. Usos

Este framework fue diseñado para tratar con un sistema *objetivo* que potencialmente no funciona correctamente. Esto podría ser tanto un proceso que fue pausado por el sistema operativo como un volcado de memoria de un programa que crasheó. De cualquier manera, no se está llevando a cabo ninguna ejecución en el proceso que está siendo depurado. Los sistemas operativos proporcionan diferentes APIs que permiten leer y escribir la memoria de los procesos, sus registros, los flags del procesador, etc.

Lxs programadores de la LMR acceden al proceso *target* desde un IDE *local* totalmente funcional. Este entorno local a menudo es muy similar al del target, aunque no necesita ser idéntico. Puede haber diferentes clases cargadas en cada sistema, o los métodos de las mismas clases podrían diferir. El framework permite completar la información faltante del target, como por ejemplo el *shape* de una clase en memoria, con información que lxs programadores suponen que es equivalente en el sistema local, el cual está disponible completamente. Las herramientas están diseñadas para permitir trabajar en un entorno donde los objetos del target se pueden inspeccionar con la misma libertad que los locales.

Los contextos en los que lxs programadores de LMR utilizan estas herramientas pueden variar considerablemente y de manera arbitraria, por lo que no parece existir un enfoque único que se ajuste a todas las situaciones: a veces se necesita simplemente acceder a estructuras en memoria, a veces ejecutar código de forma remota, a veces simular lo que haría una activación de un método remoto. Los requisitos pueden incluso no estar definidos de antemano, podría necesitarse cambiar de un tipo de semántica de ejecución a otro de manera dinámica a medida que evolucionan las tareas exploratorias.

El resto de esta sección proporciona ejemplos relevantes que encontramos mientras trabajábamos en Bee/LMR.

6.2.1. Descubrimiento de Objetos Remotos

Un sistema se pausa y queremos obtener información sobre sus objetos más importantes: sus direcciones en memoria, las clases presentes, el contenido de variables globales, símbolos, etc. Tenemos que descubrir dónde están los objetos y aún no tenemos acceso a los símbolos del sistema, por lo que puede que no sea posible enviar mensajes a los objetos.

En la memoria del sistema remoto podemos encontrar toda la metainformación que necesitamos, como el diccionario de globales, desde donde podemos empezar a obtener objetos conocidos como **true**, **false** y **nil**, la clase **Object**, su metaclase, la clase **CompiledMethod**, etc.

En este caso hay poca necesidad de información de depuración adicional, e incluso si existiera, sería difícil mantenerla actualizada con los objetos móviles en el heap.[1] Dada la dirección de un objeto, es posible, a través de la metainformación, acceder a sus campos y estructura interna. Esta situación es un buen candidato para ser resuelto con mirrors [BU04], porque principalmente requiere reflexión estructural.

6.2.2. Ejecución de Código Remoto en el Proceso Original

Al hacer profiling de un sistema, queremos recopilar información sobre la traza de ejecución observando la cadena de métodos compilados que se llaman unos a otros en la pila. Si bien la situación es en principio similar al ejemplo anterior, plantea algunas dificultades nuevas. Aunque atravesar una pila de Smalltalk para recopilar una cadena de llamadas puede hacerse fácilmente mediante mirrors, obtener información como el código fuente de los métodos compilados es una operación más compleja en Bee/LMR, y los mirrors de Bee no lo soportan directamente. Los objetos que representan los métodos compilados normalmente no referencian directamente a su código fuente, pues el mismo se encuentra en memoria secundaria. En cambio, esos objetos utilizan descriptores que permiten encontrar el código en archivos de sources cuando sea necesario su acceso. Los descriptores forman parte de la estructura interna de los métodos, y requieren implementar cierta lógica de programación.

[1] En caso de que se permitiera al proceso continuar su ejecución más adelante.

Acceder a la estructura interna de un método compilado para encontrar su código fuente directamente sería una mala práctica, porque significaría violar el encapsulamiento. En general, los objetos están diseñados de manera que desvinculan la vista externa de la estructura interna. En la práctica, esto significa que cuando se observa a un objeto desde el exterior, puede parecer muy diferente de la representación interna real. La mayor parte del tiempo, la vista proporcionada es de más alto nivel y preferiríamos trabajar con ella a través de la interfaz externa de los objetos, y solo entrar en los detalles internos cuando sea realmente necesario.

Volviendo a nuestro ejemplo de obtener las fuentes de un método compilado, un método como **#sourceCode** encapsula todo este proceso, que puede ser complejo. Sólo queremos ejecutarlo en el target y obtener el resultado.

6.2.3. Evaluación Simulada Local de Código Remoto

En determinadas situaciones es necesario pausar completamente una VM para su depuración, manteniéndola viva en memoria y adjuntándose externamente al proceso. Esto puede suceder porque algo falló o porque se desea analizar alguno de sus componentes en detalle. Podríamos querer inspeccionar objetos de dicho sistema, comprender el contexto de ejecución, enviar mensajes a objetos y tal vez realizar cambios para recuperarnos de una falla. Sin embargo, si ha habido una falla de bajo nivel los objetos a los que estamos accediendo podrían haber sido dañados, lo que significa que su propia memoria o la que determina su comportamiento (clase, métodos, etc.) podrían estar corrompidas.

Consideremos nuevamente el ejemplo de obtener el código fuente de un método compilado, esta vez en un sistema que ha fallado. En ese caso, es posible que no podamos ejecutar remotamente el código de **#sourceCode** porque eso modificaría el estado del entorno de ejecución y dificultaría la depuración. En su lugar, podríamos utilizar un enfoque alternativo: podríamos tomar el código remoto de **#sourceCode** y simular su ejecución en el target pero interpretándolo localmente.

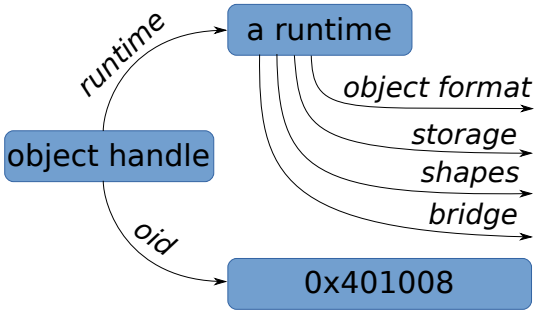


Figura 6.1: Los *handles* de objetos son referencias opacas a objetos en un entorno de ejecución remoto. Pueden responder a consultas sobre los objetos a los que apuntan solo delegándolas a su entorno de ejecución.

6.3. Diseño del Framework Metaphysics

En esta sección se describe cómo el framework Metaphysics resuelve los problemas identificados en el apartado anterior para Bee/LMR. Nuestro diseño se explica teniendo en cuenta cómo evolucionó a medida que las necesidades fueron creciendo mientras se desarrollaba Bee/LMR.

6.3.1. Conceptos Básicos de Metaphysics

La base del framework consiste en la reificación de un entorno de ejecución remoto de Smalltalk y sus objetos. Se implementó utilizando la API subyacente del sistema operativo para la depuración de procesos externos, la cual permite la comunicación con los objetos remotos principalmente vía lectura y escritura de memoria. Los principales conceptos de diseño del framework Metaphysics se muestran en la Figura 6.1.

Un **Handle** representa una entidad que vive dentro de un **runtime** dado. Hay dos tipos de handles: **ObjectHandle** para referirse a objetos comunes en el runtime, y **FrameHandle** para referirse a frames en la pila. Todos los handles conocen al **runtime** con el que están asociados, y las subclases específicas contienen diferentes slots que permiten individualizar y acceder a las entidades a las que apuntan, ya sean objetos o frames de pila, en el **runtime** asociado. Los runtimes aglomeran un conjunto de objetos que funcionan como una especie de configuración:

- Un **Storage**, que abstrae la API del sistema operativo para leer y escribir en

la memoria del proceso objetivo.

- Un **ObjectFormat**, que comprende y es capaz de leer y escribir headers de objetos remotos.

- Un **Metaspecies**, que conoce la forma de las clases en el sistema remoto, permitiendo leer y escribir slots de objetos en el target sin usar su meta-descripción remota.[2]

- Un **Bridge**, que es capaz de localizar y proporcionar handles para objetos globales en el sistema remoto.

6.3.2. Mirrors

La reflexión estructural sobre los objetos del **runtime** remoto se logra mediante mirrors. Un **Mirror** en el framework Metaphysics no es más que un contenedor de un **Handle**. Diferentes subclases proporcionan una interfaz que permite realizar consultas básicas sobre los objetos reflejados. Un **ObjectMirror** permite acceder al header y a los slots de un objeto, y a su vez puede proporcionar un mirror sobre la clase del objeto.

En Bee, los mirrors funcionan como una capa que hace una distinción entre los objetos remotos y los locales. Por defecto, los métodos que acceden a aspectos internos de un objeto devuelven nuevos mirrors que contienen las referencias directas señaladas por los slots de los objetos.

Consideremos el siguiente ejemplo: cuando un mirror de clase recibe el mensaje **name**, el resultado es un mirror para el string almacenado en el slot **name** de esa clase. Para hacer algo significativo con ese mirror, normalmente será necesario obtener una copia local de ese string, lo cual se hace a través del método **asLocalString**.

El modelo de mirrors de nuestro framework, en conjunción con el diseño presentado en el Apartado 6.3.1, es suficiente para resolver el problema de descubrimiento de objetos remotos planteado en el Apartado 6.2.1.

Listado 6.1: Uso e implementación de la semántica aumentada.

```
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
```

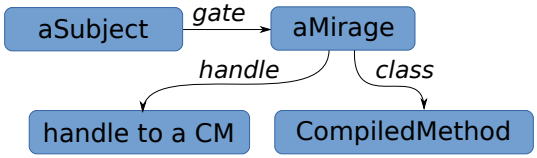


Figura 6.2: Cuando **aSubject** recibe un mensaje, éste delega la ejecución a **aMirage**, que a su vez simula la ejecución del mensaje. En este caso, el objeto proxy es un método compilado, por lo que el miraje creará un intérprete para la clase local **CompiledMethod**. El intérprete realizará una búsqueda en esa clase y recorrerá el AST del método encontrado para generar un resultado.

Listado 6.2: Delegación del mecanismo de dispatch en los subjects.

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

6.3.3. Subjects, Gates y Semánticas de Ejecución

Para comportamientos más complejos, se creó un modelo que permite variar la semántica de ejecución de mensajes. La experiencia obtenida en las situaciones descritas en los Apartados 6.2.2 y 6.2.3 mostró que los mirrors eran sólo una primera aproximación al entorno dinámico en el que deseamos trabajar.

Los mirrors permiten obtener referencias a objetos del sistema target reflejando su estructura interna. Después de eso, dada una referencia a un objeto, nos gustaría poder tratarlo como si fuera un objeto local, pero dando semánticas arbitrarias a la forma en que se ejecutan los mensajes que recibe. Eso requiere el desarrollo de otro modelo que implemente las diferentes posibles semánticas de ejecución. Para afrontar esas necesidades cambiantes modelamos los *subjects*.

Un **Subject** es un proxy a un objeto en el sistema remoto. Solo entiende el mensaje **#doesNotUnderstand:**, que sobrescribe para delegar la ejecución a distintos tipos de *gates*. Los gates de diferentes tipos producen diferentes semánticas de ejecución, trabajando como un strategy pattern. El Listado 6.2 muestra la implementación de este mecanismo.

El único slot de un subject es su **gate**, y a su vez el gate apunta al handle del objeto. Al implementar de manera diferente el método **#dispatch:**, diferentes gates pueden producir comportamientos arbitrariamente distintos en el subject cuando recibe un mensaje. La separación de subjects y gates se realiza para mantener limpia la interfaz del subject, de modo que la mayoría de los mensajes caigan en el trampolín del **#doesNotUnderstand:**.

Durante el desarrollo de Bee/LMR hemos identificado 3 tipos de gates, que podemos elegir usar dependiendo de la situación:

Trigger Gates causan la ejecución del mensaje en el proceso remoto, modificando

[2]Los metasespecies son necesarios en etapas iniciales, para proporcionar acceso inicial al meta-nivel de los objetos y para el acceso en bruto a los slots de objetos cuando se utilizan mirrors.

el estado del proceso, reanudando el proceso y devolviendo un resultado. Corresponden directamente a la semántica necesaria para abordar los problemas descritos en el Apartado 6.2.2.

Direct Gates causan la interpretación local del mensaje, obteniendo el código fuente de los métodos del objeto remoto al que se envió el mensaje.

Mirage Gates causan la interpretación local del mensaje, obteniendo el código fuente de métodos de una clase local que es equivalente a la del objeto remoto.

Los gates de tipo direct y mirage corresponden a los diferentes tipos de semántica de ejecución deseados en el Apartado 6.2.3. Requieren la implementación de un intérprete de código fuente. Este intérprete toma como entrada un AST, un receptor y un array de argumentos. Itera sobre los nodos del árbol, los evalúa y finalmente devuelve el resultado de la evaluación.

La Figura 6.2 muestra la colaboración de diferentes objetos del framework. La ejecución local de mensajes para objetos remotos puede dar lugar a handles de objetos locales. Por ejemplo, cuando se necesita crear un nuevo objeto durante la interpretación, este es instanciado localmente por el intérprete, que devuelve un handle que apunta al nuevo objeto. Los gates no necesitan saber si los handles son locales o remotos, dado que ambos respetan una misma API abstracta.

Cambiar la semántica de un subject es fácil: lxs programadores solo tienen que tomar el handle de objeto del gate y crear un nuevo subject con un nuevo tipo de gate asociado. Se permiten mensajes con argumentos, siempre y cuando cada uno de ellos sea también un subject.

6.4. Trabajo relacionado

Como se mencionó, nos basamos en las ideas de mirrors [BU04]. Trabajos similares incluyen, por ejemplo, Mirages [MVCTT07], que intentan reconciliar los mirrors con la intercesión conductual en AmbientTalk, un lenguaje orientado a objetos distribuido basado en actores.

La noción general de depuración remota de entornos orientados a objetos también ha sido estudiada. Por ejemplo, el Inspector Maxine [Mat08], Jikes RDB [MH13]

y Mercury [PBF+15] implementan la depuración remota utilizando reflexión a través de mirrors u otro tipo de middleware. Si bien Mercury proporciona funcionalidades similares a Metaphysics, sus mecanismos difieren ampliamente.

Por un lado, Mercury introduce un lenguaje modificado, *MetaTalk*, donde el meta-nivel está descompuesto estructuralmente (a través de mirrors con estado), y el target tiene que ejecutar una VM modificada que pueda explotar las características del lenguaje; proporciona un middleware adaptable, *Seamless*, para comunicar ambos sistemas, una capa de soporte para depuración en tiempo de ejecución debe ser incrustada en el entorno objetivo, y el soporte de reflexión es limitado a componentes envueltos por mirrors.

Por otro lado, Metaphysics utiliza el conocido meta-modelo de Smalltalk-80, y el entorno objetivo se ejecuta sin modificaciones, sin middleware o ninguna capa especial de soporte para depuración. La reflexión en el sistema se basa en las facilidades de reflexión ya existentes tanto en el sistema objetivo como en el anfitrión. Metaphysics fue pensado para depurar procesos remotos donde la comunicación con el entorno remoto es confiable y rápida, como un proceso remoto en la misma máquina.

A diferencia de Mercury, Metaphysics no requiere una capa de middleware porque se conecta directamente al target utilizando su metainformación ya incluida. Además, en Metaphysics, no se necesita un modelo de la aplicación depurada en la máquina del desarrollador, sino que puede descargarse de forma lazy desde el target.

Otras propuestas incluyen el uso de objetos holográficos para tratar snapshots de programas bloqueados [SK13], o infraestructuras de virtualización para entornos de ejecución de lenguajes de alto nivel orientados a objetos [PDFB13, Pol15].

Capítulo 7

Evaluación Cualitativa

Esta sección primero funadmenta cómo las LMRs mejoran el estado del arte en máquinas virtuales para resolver los problemas identificados en el Apartado 3.2. Luego detalla cómo facilitan la resolución de cada uno de los problemas de los casos de estudio establecidos en el Capítulo 3. Finalmente, discute los compromisos que introducen las LMRs.

7.1. Cómo las LMRs Mejoran el Estado del Arte en VMs

Argumentamos a favor de eliminar la estricta arquitectura de dos capas y la separación entre la VM y la aplicación. Utilizar únicamente técnicas orientadas a objetos para estructurar los programas nos permite resolver los problemas identificados en el Apartado 3.2 en las formas descriptas a continuación.

7.1.1. Observabilidad Limitada de la VM (PG1)

Como se determinó anteriormente, la estricta arquitectura de dos capas restringe deliberadamente lo que se puede observar de la VM. Sin embargo, para nuestros casos de estudio, sería beneficioso monitorear más libremente no sólo la aplicación sino también el comportamiento de la VM. Con las LMRs siendo *parte de la aplicación*, hay una conexión más directa entre el código de la aplicación y la VM,

lo que nos permite observarlas de la misma manera que la aplicación.

Los sistemas de programación implementados usando LMRs también proporcionan conexiones causales más directas entre el código de la aplicación y la VM que las VMs tradicionales, lo que también se puede aprovechar, por ejemplo, para adaptarse más fácilmente a escenarios no anticipados y ayudarnos a mejorar las aplicaciones de formas novedosas.

Lenguaje de Programación Unificado

Un beneficio implícito es que lxs desarrolladores de aplicaciones no necesariamente tienen que aprender un lenguaje diferente para trabajar com la VM. Al menos en la realización más simple de las LMRs, el lenguaje utilizado para la VM es el mismo que el utilizado para la aplicación, con las pequeñas diferencias discutidas en el Apartado 4.3, que son más bien un conjunto de convenciones que se pueden aprender sobre la marcha. Al igual que con los objetos y metaobjetos en un sistema como Smalltalk, en las LMRs no hay diferenciación entre lo que es código de VM y lo que es código de aplicación.

Herramientas de Programación Unificadas

Otro beneficio es que lxs programadores de aplicaciones no necesitan aprender un nuevo conjunto de herramientas, porque sus herramientas regulares para navegación de código, profiling, inspección de objetos y depuración son las mismas que normalmente utilizan. Sin embargo, cambiar un sistema durante su ejecución conlleva, por supuesto, el riesgo añadido de hacer cambios que resulten en fallos fatales. En nuestra experiencia, esto resulta en una experimentación más cuidadosa, pero no la desalienta.

7.1.2. Modo de Desarrollo Separado para la VM (PG2)

Dado que las LMR son esencialmente parte de la aplicación, se benefician del mismo modelo de programación que el código de la aplicación y del soporte para entornos de programación vivos. Esto también significa que no se necesita una *toolchain* o entorno de compilación separado. Además, todo el código de la VM está disponible de inmediato para lxs desarrolladores desde el momento en el que se

descarga el sistema, y no es necesario compilar partes de la VM. Beneficiarse de las mismas herramientas también significa que el código se compila automáticamente cuando lxs usuarixs aceptan cambios en el código de la VM, de la misma manera que se hace para el código de la aplicación. Por lo tanto, el modo de desarrollo de la aplicación también es el modo de desarrollo de la VM.

Aprendizaje Incremental

Lxs desarrolladores de aplicaciones que usan LMRs pueden navegar por el código de las diferentes partes de la VM e inspeccionar las piezas relevantes, como lo hacen con el código del repositorio de la aplicación. Del mismo modo, en lugar de simplemente imaginar cómo se comporta el código de la VM en tiempo de ejecución, pueden depurarlo y observarlo. Por ejemplo, en lugar de imaginar cómo o cuándo el compilador desencadena la recompilación, es posible colocar un punto de interrupción en el código mismo del compilador y depurar sus mecanismos mientras se ejecuta. Por lo tanto, es posible ejecutar el código de la VM paso a paso para comprender cómo encajan las piezas, cuándo y dónde se necesita.

7.1.3. Feedback Loops Largos entre Edición, Compilación y Ejecución para Componentes de VM (PG3)

De los beneficios previamente delineados también se sigue que tenemos el mismo feedback inmediato para el código de la VM que para el código de la aplicación. Este feedback instantáneo reduce el tiempo desde las ideas hasta los experimentos. En las LMRs, el código de la VM está disponible en el entorno de desarrollo exactamente de la misma manera que el código de la aplicación. Por lo tanto, no hay toolchains de compilación adicionales para la VM y compilar la VM no requiere ninguna acción diferente de las utilizadas para la aplicación. No es necesario que lxs desarrolladores realicen acciones diferenciadas para leer y escribir el código de la VM. Los cambios en el código se aplican al instante, lo que permite acciones similares al scripting de aplicaciones, pero durante el desarrollo de la VM con la aplicación ejecutándose al mismo tiempo.

Interfaz de Programación Unificada entre Aplicación y VM

Otro beneficio de eliminar la distinción arquitectónica entre la VM y la aplicación es que los componentes que conforman la VM en una LMR son reutilizables como bibliotecas dentro de la aplicación y viceversa. No es necesario idear una interfaz de bajo nivel para pasar objetos a la VM desde la aplicación, ni una interfaz para utilizar objetos de alto nivel dentro de la VM, ya que ambas partes utilizan una única representación uniforme.

Además, es posible realizar consultas sobre el estado del sistema en tiempo real. Por ejemplo, para saber cuánto tiempo se está gastando en la recolección de basura, no es necesario activar un modo de depuración en la VM, exportar estadísticas de GC a archivos de datos u otras herramientas para finalmente analizarlo. En cambio, lxs desarrolladores pueden acceder directamente a los objetos del recolector de basura, escribir algunas líneas de código en el lenguaje de la aplicación para recopilar y analizar los datos del sistema en vivo y, con una sola pulsación de tecla, inspeccionar el resultado de ese análisis.

7.2. Evaluación de los Casos de Estudio con una Solución Basada en LMRs

Reanudando el análisis de los casos de estudios descriptos en el Apartado 3.1, ahora mostramos cómo Bee/LMR nos permite resolver los problemas y superar las limitaciones identificadas en el Apartado 3.2.

7.2.1. Ajuste de la Recolección de Basura (GCT)

Nuestro primer caso de estudio identificó problemas de rendimiento relacionados con la recolección de basura. Específicamente, abrir la aplicación desde cero puede llevar más de un minuto y se gasta una cantidad significativa de tiempo en la asignación de objetos y recolección de basura.

El primer paso es comprender dónde exactamente se gasta el tiempo. Dado que el GC se implementa como una biblioteca, eso se puede identificar con las herramientas estándar de profiling junto con el código de la aplicación en el contexto

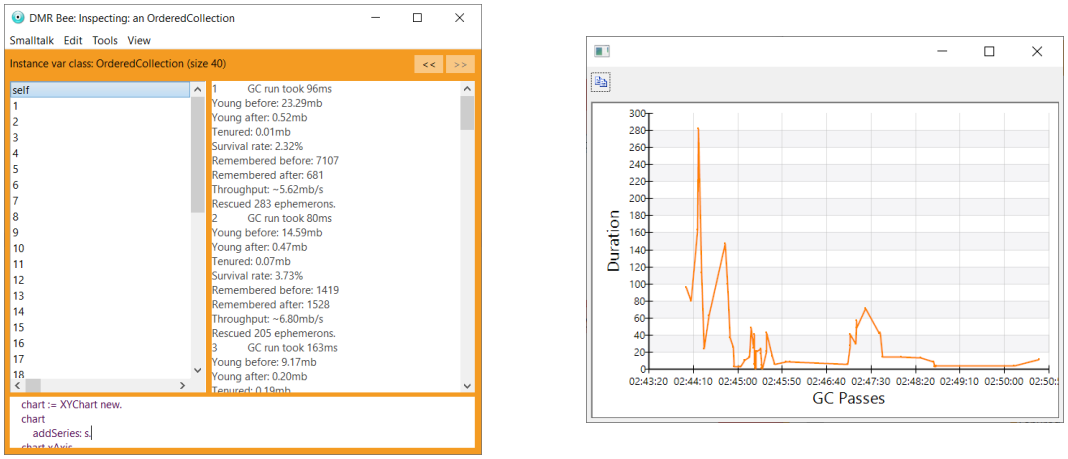


Figura 7.1: Depuración de un recolector de basura con herramientas estándar de desarrollo de aplicaciones. A la izquierda, un inspector muestra las estadísticas de recolección de basura del sistema en vivo, mientras que a la derecha, un gráfico ilustra el tiempo tomado por el proceso de recolección de basura basado en esas estadísticas.

del problema, lo cual a menudo no es posible de la misma manera en otras VM, porque intentan hacer que el GC sea transparente y no observable y las herramientas lo presentan por separado.

Dado que el profiling del GC se realiza de la misma manera que el código de la aplicación, lxs desarrolladores pueden ver desde un visualizador de resultados cuánto tiempo se ha gastado en la asignación de objetos, cuántas recolecciones de basura se activaron, y qué partes del GC, si las hay, son motivo de preocupación.

Una vez que un área es identificada como un cuello de botella de rendimiento, quisiéramos recopilar estadísticas más específicas del GC para entender mejor por qué pasa mas tiempo del que se espera allí. La Figura 7.1 muestra las herramientas utilizadas durante el caso de estudio, que ya son conocidas por lxs programadores de la aplicación.

Dado que los objetos del GC están directamente disponibles para lxs desarrolladores, se vuelve posible modificarlos directamente, mientras el sistema ejecuta, para recopilar estadísticas como los tamaños del heap a lo largo del tiempo, tasas de supervivencia de objetos y de tenuring, así como los tamaños del remembered set.

Por ejemplo, podemos agregar una variable de instancia a la clase **GenGCPass**, encargada de recopilar estadísticas del GC generacional, para almacenar el tiempo

exacto de inicio de cada paso de GC generacional, y luego modificar el código que del GC para que efectivamente se almacene ese valor cuando el GC comienza. En nuestro caso de estudio, lxs desarrolladores utilizaron los datos obtenidos para experimentar con diferentes tamaños de heap de GC generacional y heurísticas de activación mientras el sistema estaba en funcionamiento, donde era posible modificar esas cosas con el sistema en ejecución y observar y medir los cambios en tiempo real, teniendo feedback inmediato y datos para mejorar el tiempo de inicio de la aplicación. Con esta puesta a punto, el tiempo de inicio de la aplicación se redujo en aproximadamente un 25 %.

7.2.2. Recompilaciones Recurrentes por el Compilador JIT (JITC)

Similar al caso de estudio de ajustes en el recolector de basura, el profiling de nuestra aplicación en Bee/LMR también ayuda a identificar problemas de rendimiento con el compilador JIT.

Dado que con una LMR el compilador JIT es simplemente otra biblioteca, nuevas soluciones se vuelven posibles. Nuestrxs desarrolladores de aplicaciones pueden usar el profiler para navegar al código del compilador JIT, e identificar por qué tarda demasiado tiempo.

En este caso, lxs desarrolladores notan que la VM está invalidando y recompilando código repetidamente, porque para la misma clase, se ven dos métodos diferentes: el original y el método de instancia agregado a un objeto individual. Para proporcionar una solución rápida para el cliente que encontró el problema en producción, intentan modificar la VM para permitir que el compilador JIT almacene en caché el código compilado de ambos métodos. En lugar de desencadenar una recompilación al ver un método específico de instancia, agregan una búsqueda en la caché de código compilado y la compilación sólo se desencadena si no hay una entrada coincidente en la caché.

El código adaptado en el Listado 7.1 implementa este truco simple en el compilador JIT. Aunque este método pertenece al “código de la VM”, es sencillamente un método Smalltalk y por lo tanto puede cambiarse como cualquier otro método en el entorno de programación vivo. Además, lxs desarrolladores no necesitan conoci-

Listado 7.1: El método **NativizationEnvironment » nativeCodeFor:** **aBytecodeMethod** obtiene el código nativo necesario para ejecutar el método pasado como argumento. Agregar la caché mejora el soporte para métodos específicos de instancia en el compilador JIT, evitando recompilaciones repetidas.

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

mientos específicos sobre cómo funciona la compilación. Solo necesitan agregar la comprobación de un diccionario y el almacenamiento en caché del código nativo en él. Las herramientas, el lenguaje y los conceptos utilizados son bien conocidos por lxs desarrolladores. La Figura 7.2 muestra un depurador detenido en el punto de ensamblado de un byte. La vivacidad del sistema permite a lxs desarrolladores ver los resultados al instante, sin necesidad de reiniciar el sistema, incluso codificando en el depurador. Por lo tanto, esta solución podría entregarse a lxs clientes sin requerir que reinicien su aplicación.

Para resumir, cuando lxs desarrolladores utilizan una LMR para hacer profiling de sus aplicaciones, obtienen detalles adicionales sobre el funcionamiento del compilador JIT. Si el compilador JIT causa problemas de rendimiento, aparece como cualquier otra parte de la aplicación, lo que permite identificar oportunidades de mejora. Por lo tanto, las LMR eliminan barreras de complejidad accidental e invitan a lxs desarrolladores a resolver los problemas de manera más directa, con feedback instantáneo sin necesidad de reiniciar el sistema. Esas optimizaciones pueden entregarse a clientes sin siquiera requerirles que realicen un reinicio de la aplicación.

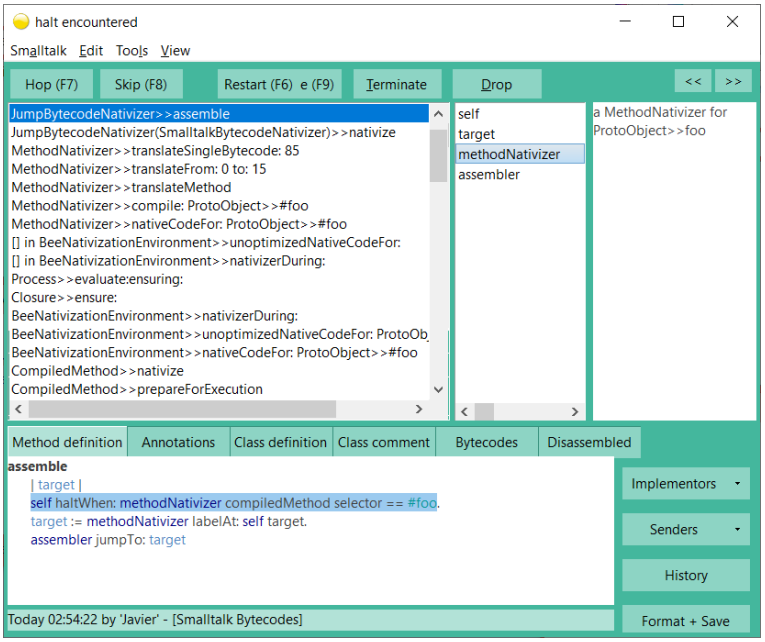


Figura 7.2: Un depurador mostrando el stack del compilador JIT ensamblando un jump

7.2.3. Optimizaciones SIMD (CompO)

El tercer caso de estudio tuvo como objetivo agregar soporte para operaciones vectorizadas para acelerar las aritméticas de punto flotante que se utilizan en el código de simulación de la aplicación. Bee/LMR nos permite aprovechar la toolchain de compilación de la VM para escenarios no anticipados. En particular, es posible adaptar el compilador para optimizar el código relevante para nosotros, lo que normalmente no es una opción con las VMs más avanzadas.

Dado que la optimización del compilador puede ser específica a la aplicación, es posible implementar el soporte mínimo indispensable para la aplicación concreta, en lugar de construir un sistema genérico que sea adecuado para una amplia gama de estructuras de código. Esto no solo brinda beneficios inmediatos, sino que también simplifica significativamente el problema y permite a lxs desarrolladores agregar soporte incrementalmente a patrones de código que se consideren importantes en la aplicación específica.

El Listado 7.2 muestra dos implementaciones de la suma de elementos de dos arreglos de punto flotante. En esta implementación, los elementos se suman uno por uno. Una versión optimizada con SIMD se muestra en el Listado 7.3. Lxs

programadores deben implementar un nodo del compilador que sea ensamblado a la instrucción **addps** deseada. Este código máquina puede depurarse con un depurador de Smalltalk convencional que incluya información sobre el código nativo. Como en los casos de estudio anteriores, no es necesario realizar tareas no relacionadas con el problema que se está resolviendo, como por ejemplo la recompilación de la VM. La implementación del soporte del compilador para ensamblar el nuevo tipo de nodo se realiza mientras el sistema está en ejecución, sin necesidad de reiniciar, y se puede cambiar y depurar según sea necesario.

7.3. Discusión

Al eliminar la separación estricta entre aplicaciones y VMs, podemos beneficiarnos de una mayor comprensión de la ejecución gracias a la utilización de herramientas estándar. Cambiar la VM y las bibliotecas de tiempo de ejecución se vuelve tan directo e inmediato como cambiar cualquier código de aplicación. A continuación, discutiremos brevemente otros beneficios y desventajas que encontramos con este enfoque.

7.3.1. Beneficios Adicionales

La flexibilidad que aportan las LMR nos brinda una gama más amplia de opciones que las VMs tradicionales cuando enfrentamos escenarios problemáticos y requerimientos de cambios no anticipados. Dado que las LMR reducen la longitud del ciclo de feedback para los cambios en la VM en órdenes de magnitud, la experimentación y la exploración requieren mucho menos tiempo y esfuerzo. Como se observó al cachear los resultados del compilador JIT en el Apartado 7.2.2, este tipo de experimentación y numerosos cambios pueden realizarse sin un conocimiento particularmente profundo del recolector de basura, compilador JIT, código nativo o conceptos similares que pueden ser nuevos para lxs desarrolladores de aplicaciones. A menudo, es tan simple como cachear algún resultado o recopilar algunos datos para comprender mejor qué está haciendo el sistema.

Por ejemplo, al observar las características del código nativo desde un alto nivel, uno puede aprender cómo funciona el compilador JIT y cómo las decisiones de

Listado 7.2: El método **FloatArray » += aFloatArray** realiza la suma de elementos uno a uno del argumento en el receptor. Invoca una implementación naive que realiza la suma uno por uno. **_floatPlus:**, **_floatAt:** y **_floatAt:put:** son metamensajes que se evalúan en tiempo de compilación y generan nodos especializados del compilador.

```
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
```

Listado 7.3: La versión SIMD es bastante similar, pero aplica la operación a múltiples datos en cada ciclo. El número de iteraciones se divide por la cantidad de sumas paralelas, el tipo de almacenamiento y carga se ajusta para que el compilador use registros SIMD apropiados y tamaños de operandos. El ensamblado para la suma se cambia a **addps**, una suma de números de punto flotante de precisión simple empaquetada.

```
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
```

optimización del compilador JIT afectan el resultado. Esto nos permite observar las decisiones tomadas por el compilador JIT, analizarlas y modificarlas. Por ejemplo, podemos detectar qué métodos están siendo compilados inline o no. Dependiendo del contexto, muy poco o demasiado inlining puede resultar en un peor rendimiento. Dado que es un sistema vivo, uno puede ajustar las heurísticas y ver los resultados sin tener que reiniciar el sistema.

7.3.2. Importancia de la Vivacidad en las LMRs

Desde la perspectiva arquitectónica, es posible construir entornos de ejecución que eliminen la separación entre VM y aplicación, pero sin que la VM pueda programarse en vivo. Es probable que tales sistemas en también reduzcan la brecha de conocimiento, mejoren la conexión causal entre el entorno de ejecución y el código, y permitan a lxs desarrolladores de aplicaciones comprender mejor el sistema en tiempo de ejecución.

Sin embargo, el feedback instantáneo y la observabilidad de las LMRs también convierten el mero potencial de estos beneficios en encuentros fortuitos que sucederán, lo cual es clave para la comprensión de grandes repositorios de código. Sólo porque uno tenga la posibilidad de acceder al código fuente, no significa que lo vaya a examinar. Sin embargo, en una LMR, el código estará ahí mismo en el depurador o profiler que se está utilizando para diagnosticar cualquier problema, y se puede experimentar con él al instante sin necesidad de realizar ninguna acción adicional. Creemos que esta es una situación cualitativamente diferente a la que se presenta en entornos clásicos.

7.3.3. Inconvenientes y Preocupaciones Asociadas con las LMRs

Eliminar la distinción entre VM y aplicación conlleva naturalmente preocupaciones sobre abandonar los beneficios de esta arquitectura. Discutimos brevemente estas preocupaciones y otras como la seguridad del software.

Mantenibilidad y Portabilidad

Como se discutió anteriormente, el diseño de dos capas es elegido por muchas VMs a propósito, para garantizar una fuerte separación entre VMs y aplicación. Con

VMs como la Máquina Virtual de Java, esto brinda a las aplicaciones la oportunidad de alternar entre distintas JVMs con la confianza de que la aplicación seguirá funcionando. Con las LMRs, esta separación y garantía estricta ya no se dan. Sin embargo, argumentamos que en algunas situaciones este es un compromiso que vale la pena hacer.

Considerando que muchas aplicaciones grandes usarían la técnica de *vendoring* de los frameworks en los que confían en su propio código, una LMR es esencialmente la práctica de incluir la biblioteca del runtime en la aplicación. El *vendoring* se hace típicamente para ganar más estabilidad y poder controlar completamente un framework o biblioteca. Por otro lado, esto conlleva el costo de mantener esta bifurcación y enviar cambios de vuelta al original. Se podría argumentar que lo mismo ocurre con las LMR y, como con los frameworks de vendors, ofrece toda la flexibilidad y el costo de hacerlo. Sin embargo, uno siempre puede decidir no cambiar nada, lo que típicamente hará relativamente simple mantenerse actualizado con cualquier cambio realizado upstream.

Seguridad

Cuando se crea una máquina virtual para un lenguaje de programación, la seguridad del software es una preocupación crucial. Sostenemos que mover el código entre capas arquitectónicas sólo cambia la situación superficialmente y cualquier amenaza concreta es específica del lenguaje de programación de la LMR. Como tal, no hay una respuesta general y si una LMR introduce nuevas superficies de ataque depende del lenguaje en cuestión.

Para Bee/LMR, la situación no es ideal desde el principio. Siendo un Smalltalk, hay muchas oportunidades para atacar el sistema al evaluar código arbitrario en tiempo de ejecución o al usar la operación **#become:** para intercambiar objetos arbitrariamente entre sí. Lenguajes como Java y JavaScript típicamente tienen mecanismos para restringir lo que puede cambiarse.

Por ejemplo, el sistema de módulos de Java también es capaz de prevenir accesos reflexivos. JavaScript tiene el método **freeze()** que le permite hacer objetos inmutables. Tales mecanismos podrían teóricamente ser utilizados para proporcionar alguna forma de protección. Sin embargo, tener un compilador JIT, su ensamblador accesible, y la capacidad de acceder a memoria arbitraria en teoría permite la ejecu-

ción de código arbitrario y como tal requeriría un diseño cuidadoso para evitar que atacantes obtengan acceso a esta capacidad. Los sistemas de tipos y capacidades, por ejemplo utilizando mirrors [BU04], podrían facilitar un diseño de sistema adecuado.

Estabilidad

Desde un punto de vista práctico, lxs desarrolladores tienen flexibilidad para elegir su forma de trabajo preferida. Cambiar el sistema mientras está en ejecución es arriesgado, ya que incluso un error menor puede hacer que el sistema se bloquee. En Bee/LMR, no hay mecanismos especiales para evitar crashes que no sean los que ya brinda el propio lenguaje. Esto incluye la indexación segura de arrays y mecanismos como **doesNotUnderstand**, que permiten a lxs programadores detectar y corregir errores a medida que aparecen.

Es posible y es una práctica común guardar la imagen justo antes de aplicar dichos cambios. Además, si ocurre un crash, se abre un depurador remoto que permite inspeccionar la imagen congelada antes de terminar el proceso, para ayudar a entender la causa del error. Alternativamente, Bee/LMR se puede editar como código offline, es decir, desde los fuentes en archivos. Esto permite modificar el código y luego hacer un bootstrap del sistema, lo que permite aplicar cambios más complejos que podrían no ser seguros en un sistema en ejecución.

En la práctica, lxs desarrolladores a menudo comienzan con el flujo de trabajo en vivo y luego cambian algo más seguro una vez que encuentran los límites de lo que es posible en el sistema vivo. Lo mismo ocurre con el diseño del sistema en general. A menudo nos encontramos eligiendo cambios de diseño más conservadores, lo que facilita las actualizaciones en vivo y reduce el riesgo de problemas de estabilidad.

7.3.4. Dinamicidad del Metamodelo y Escalabilidad del Sistema

Nuestra implementación de la LMR no amplía el protocolo de metaobjetos para permitir cambios como la modificación del formato en memoria de los objetos sobre la marcha. Si bien en principio uno puede tratar a todos los objetos, o más bien la memoria del programa, como bits y ejecutar un script sobre ellos para hacer tales actualizaciones, Bee no proporciona ningún soporte especial o API para

facilitar tales cambios. Pequeños cambios como modificar el significado de un bit libre en los headers de los objetos pueden ser explorados mientras se ejecuta el programa. Sin embargo, cambios más grandes generalmente se realizan modificando los archivos fuente y volviendo a arrancar el sistema.

Al considerar la escalabilidad de una LMR, uno puede estar preocupado por el soporte de heaps de objetos grandes, así como por el soporte de grandes repositorios. Dado que utilizamos el diseño de GC G1, no hemos notado ningún problema de escalabilidad. G1 está diseñado para grandes heaps y para ofrecer garantías de soft real-time en las pausas de recolección de basura.

Cuando se trata de repositorios de código grandes, el enfoque de Smalltalk de un sistema basado en imágenes que contienen el código escala de manera bastante natural. Dado que el código se nativiza en tiempo de ejecución y precompila en tiempo de desarrollo de a un método a la vez y bajo demanda, la creciente base de código no ha sido una preocupación notable, incluso con nuestra aplicación de 1,1 millones de líneas de código.

7.3.5. Uso en la Vida Real

Por lo general, lxs desarrolladores de aplicaciones no intentan optimizar directamente el GC o el JIT de una LMR, sino que utilizan la información más rica obtenida del sistema vivo para modificar su aplicación, de modo que puedan tomar mejores decisiones de diseño dentro de su código para evitar producir cuellos de botella en la VM.

Lxs desarrolladores de aplicaciones que utilizan LMRs cuentan con las herramientas que usan a diario en su entorno de programación vivo para inspeccionar, navegar y depurar el código del GC si lo desean. Además, si desean modificar componentes como el JIT y el GC, pueden hacerlo con el sistema en funcionamiento, con feedback instantáneo.

En el momento del desarrollo, incluso para lxs desarrolladores de aplicaciones, puede ser útil probar pequeños cambios en la VM, realizar pequeñas mejoras y ejecutar pequeños experimentos, aunque pequeños errores puedan hacer que el sistema se bloquee durante la sesión de desarrollo. Este tipo de experimentos pueden resultar en cambios que, si se desea, luego se pueden enviar a expertos en VM para

evaluar su seguridad e inclusión en el repositorio de código principal.

7.4. Casos de Estudio Adicionales

Si bien no forman parte de los casos de estudio principales (GCT, JITC, CompO), aquí describimos otros escenarios donde el uso de una LMR mostró beneficios prácticos en contraste con el diseño de aplicación/VM de dos capas.

7.4.1. Detección de Memory Leaks

La presencia de un sistema vivo permitió usos novedosos del runtime de gestión de memoria que típicamente no son prácticos con VM estáticas. El GC puede modificarse en vivo y aprovecharse de formas no previstas.

En Bee/LMR, aprovechamos el GC para detectar memory leaks en un servidor de solicitudes HTTP. Bee incluye un GC generacional compuesto por un espacio **eden** y dos áreas también jóvenes **from** y **to** adonde los objetos van antes de moverse al área **old**. Debido a que el GC está integrado como una biblioteca, es posible rastrear objetos vivos en espacios particulares, por lo que encontrar objetos leakeados puede hacerse mediante las siguientes acciones:

1. Antes de manejar una solicitud HTTP, activar un GC generacional para limpiar el espacio **eden**.

2. Deshabilitar el GC durante el manejo de la solicitud, de modo que todos los objetos se asignen en el **eden**, haciendo que **eden** crezca si es necesario durante la solicitud.

3. Después de manejar la solicitud, activar un GC generacional para que los objetos vivos del **eden** se muevan al espacio **from**.

4. Usando la API del GC, iterar a través de los objetos en el espacio **from** agregándolos a una colección weak. El contenido de esta colección es un superconjunto de los objetos leakeados. Todavía puede retener objetos que fueron apuntados desde el *remembered set* pero que no están realmente vivos.

Listado 7.4: El método **Memory»objectsSurviving: aClosure** evalúa el bloque pasado como argumento y devuelve los objetos que se leakearon. **HttpWorker»processRequest: anHttpRequest** usa esa API para mostrar el resultado gráficamente.

```
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.
```

5. Ejecutar un GC completo para anular las entradas de la colección que son inalcanzables al rastrear el grafo completo de objetos. El contenido de este conjunto es ahora el conjunto de objetos leakeados.

El código para realizar estas acciones cabe en un método, y se agregó al gestor de memoria y se probó con feedback inmediato. La API recibe un bloque como argumento y devuelve los objetos creados que sobrevivieron a la evaluación de ese bloque. El código se muestra en el Listado 7.4.

Como en otras situaciones, esta API es principalmente útil en tiempo de desarrollo más que después de su deployment. En el escenario de las VMs tradicionales, implementar esto habría requerido atravesar todas las barreras detalladas en el Apartado 3.2: implementar el cambio en la VM, recompilarla, reiniciar la aplicación con la VM parcheada y luego probar el experimento. En la LMR, el experimento se realizó sin demoras. De hecho, el primer intento guardó los resultados en una colección estándar no weak, y nos permitió descubrir que necesitábamos un filtrado adicional de objetos a través de una recolección de basura completa. En el caso de

Listado 7.5: El método **TestSuite » coverage** limpia la caché de código, ejecuta una suite de pruebas y finalmente cuenta cuáles de los métodos en el sistema han sido ejecutados verificando cuántos han sido asignados con código nativo.

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

la VM tradicional, eso habría requerido otro paso de recompilación y reinicio que no fue necesario en la LMR.

7.4.2. Implementación de Cobertura de Código de Tests

Bee contiene herramientas para analizar la cobertura del código de tests. Estas herramientas se basaban en la instrumentación de métodos compilados. Si bien útil para la mayoría de los escenarios típicos, el enfoque de instrumentación era demasiado lento para ejecutarse en los 15000 tests del sistema que ponen a prueba las 1,1 millones de líneas de código de la aplicación.

Este problema se resolvió accediendo a la información del compilador JIT desde la aplicación, en lugar de utilizar instrumentación: como Bee/LMR sólo ejecuta código mediante la compilación JIT de métodos, si la caché de código JIT se borra antes de ejecutar las pruebas, y el tamaño de esa caché es lo suficientemente grande, entonces es posible determinar qué métodos se han ejecutado simplemente comprobando si se agregaron a la caché de código. Esta información está disponible de inmediato en el sistema y se puede recopilar con un script como el que se muestra en el Listado 7.5.

El cambio requerido fue implementado por lxs desarrolladores de la aplicación en la biblioteca de unit tests, y les permitió obtener estadísticas de cobertura sin incurrir en penalidades de rendimiento.

Listado 7.6: Un nodo **BitsAt** contiene un par de subnodos izquierdo y derecho. El izquierdo puede ser cualquier cosa, el derecho es una constante que apunta a un objeto de tipo bitfield. El método **optimized** accede a la constante y genera código optimizado (un **bit-and**, seguido por un **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
```

7.4.3. Otras Optimizaciones

En cierto momento, nuestra aplicación de simulación incorporó un tipo **BitField** que simplificaba el trabajo con bit-fields. El tipo permitía extraer bits de un entero como si fuera un bit-field en C, y también escribir bits de vuelta. El uso es simple, el cliente crea un bit-field con algo como **flags := BitField bits: 4 to: 6** y luego se pueden extraer los bits con, por ejemplo, **flags bitsAt: field** y escribir de vuelta con **flags bitsAt: field put: aValue**.

Después de algún tiempo, se descubrió que esta simple abstracción estaba causando una pequeña penalidad de rendimiento. Como el compilador no podía demostrar que los bit-fields eran inmutables, las operaciones para *shifts* y *ands* estaban utilizando código genérico con casos de fallback para tipos no enteros. Agregar soporte general para la inmutabilidad al compilador estaba fuera del alcance, pero en lugar de eliminar la abstracción, era posible ajustar manualmente el compilador de Bee/LMR para incorporar una optimización ad-hoc. Con esta optimización, cuando el compilador ve un mensaje **bitsAt:** enviado a un objeto bitfield, asume que éste es inmutable y genera código optimizado para el tamaño específico del bit-field, eliminando todas las penalidades de rendimiento.

La implementación de esta optimización involucró dos pasos: el primero fue agregar **bitsAt:** y **bitsAt:put:** a una lista de mensajes especiales en el compilador

optimizador de código nativo. Al ver tales mensajes, el compilador convierte el nodo **MessageSend** en un nodo **BitsAt** especializado, una subclase de nodos **BinaryMath**. El segundo paso fue la optimización en sí misma. Para eso, el nodo **BitsAt** implementa el método **optimized**, que a su vez se convierte en una serie de nodos *shift* y *and*, como se muestra en el Listado 7.6.

Dado que la optimización permitió bit-fields sin costo adicional, esta abstracción fue incorporada de nuevo en el resto del código del entorno de ejecución, simplificando los accesos a los bit-fields almacenados en métodos y clases compiladas.

Capítulo 8

Evaluación Cuantitativa

Para asegurar la viabilidad de nuestro enfoque, realizamos una serie de pruebas de rendimiento que proporcionan un límite inferior al rendimiento de los sistemas basados en LMRs. En la práctica el rendimiento no es un problema para las LMRs. Bee/LMR es utilizado diariamente por un equipo de cuatro desarrolladores y se emplea para ejecutar nuestro producto en producción, el cual recibe dos actualizaciones principales por año, con lanzamientos menores mensuales para clientes específicos.

8.1. Evaluación de Rendimiento

El objetivo de esta evaluación es mostrar que las LMR pueden alcanzar los niveles de rendimiento de los sistemas basados en VMs tradicionales. Comparamos Bee/LMR con los siguientes sistemas:

Pharo/Cog Pharo es un dialecto de Smalltalk que se ejecuta sobre la VM OpenSmalltalk, la VM de Smalltalk más utilizada. Ejecutamos OpenSmalltalk en modo dual JIT/interpetador (Cog) [Mir11]. OpenSmalltalk está escrito en Slang, transpilado a C y compilado con un compilador C estándar.

Python/CPython La VM estándar de Python (v3.10.7) que está escrita en C y utiliza un intérprete.

Ruby/MRI La VM estándar de Ruby (v3.0.4p208), también escrita en C.

JavaScript/V8 El motor de JavaScript detrás de Node.js (v18.16.0), escrito en C++, que incluye un intérprete y varias etapas de compilación JIT optimizadas.

Utilizamos los benchmarks *Are We Fast Yet*, que incluyen 9 micro y 5 macro benchmarks [MDM16]. Están diseñados para comparar el rendimiento entre diferentes implementaciones de lenguajes y, por lo tanto, fueron fáciles de adaptar a Bee.

Los benchmarks se ejecutaron en una máquina con un Core i7 7700HQ de 4 núcleos a 2.8 GHz con hiper-threading y 16 GB de memoria. El sistema operativo es Ubuntu 22.10 de 64 bits. Bee se ejecuta a través de la capa de compatibilidad Wine, ya que sólo es compatible con Windows. Todas las implementaciones son de 64 bits.

Medimos 100 iteraciones de cada benchmark, y en el caso de Node.js 3000 para minimizar el ruido de la compilación JIT tardía. Para cada benchmark, tomamos la mediana de las mediciones y reportamos el resumen como un diagrama boxplot estándar en la Figura 8.1. Los resultados están normalizados a Node.js, que es el sistema más rápido.

En general, Bee/LMR es ligeramente más lento que Pharo/Cog. Esto es esperado, ya que la VM de Cog ha sido optimizada durante un período mucho más largo que Bee/LMR. Por ejemplo, la copia de strings en Bee se realiza byte a byte, verificando los límites en cada carácter. Si bien Bee/LMR posee un optimizing compiler, el mismo sólo se utiliza ahead of time, durante el bootstrapping y sobre un conjunto de métodos pre-seleccionados. Bee es aproximadamente un orden de magnitud más rápido que los intérpretes de Ruby/MRI y Python/CPython. Dado que Bee/LMR utiliza un compilador JIT de tipo template [Ayc03], hay mucho espacio para mejorar su rendimiento con optimizaciones clásicas del compilador, al comparar con Node.js y Pharo.

Sin embargo, dado que CPython y MRI son intérpretes clásicos de bytecode sin compilación JIT, Bee/LMR los supera aproximadamente al nivel que se esperaría de un compilador de tipo template. Python es conocido por ser uno de los intérpretes más lentos, y las versiones más nuevas, a partir de Python 3.11, tienen como objetivo solucionar este problema[1].

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

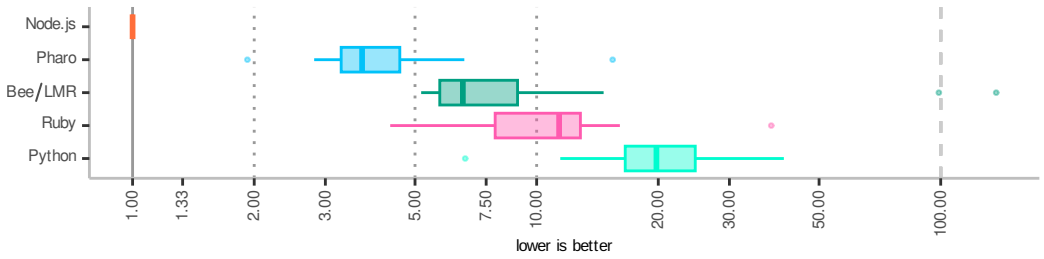


Figura 8.1: Tiempos de referencia de Bee/LMR en comparación con otros sistemas orientados a objetos dinámicos.

8.2. Tamaño de la Implementación del Entorno de Ejecución

Evaluamos el tamaño de Bee/LMR contando las líneas de código de diferentes partes del sistema asociadas con los componentes de la VM: compilador JIT, GC y funciones built-in. Comparamos esa métrica con los otros sistemas para dar una idea de la diferencia. Bee/LMR es significativamente más pequeño, porque es mucho más especializado y no soporta la misma variedad de sistemas operativos, arquitecturas de procesadores y escenarios de uso.

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) Líneas de código de los subsistemas de Bee/LMR.

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) Líneas de código de diferentes VMs para lenguajes orientados a objetos, comparadas con Bee/LMR.

Figura 8.2: Tamaño del módulo LMR en líneas de código.

Capítulo 9

Conclusiones y Trabajo Futuro

Proponemos los entornos de ejecución metacirculares vivos (LMRs), un diseño de entorno de ejecución que combina la implementación de la VM y la aplicación, reemplazando a la arquitectura tradicional que las separa en capas. Nuestro enfoque utiliza técnicas básicas de orientación a objetos para garantizar, por ejemplo, el encapsulamiento y para permitir cambios en los entornos de ejecución en tiempo de ejecución, igual que el código de aplicación normal.

En este trabajo, utilizamos casos de estudio sobre ajustes del recolector de basura, modificación del compilador JIT para evitar recompilaciones innecesarias, y agregado de soporte para instrucciones vectoriales al compilador, para argumentar a favor de los beneficios de eliminar la separación arquitectónica, lo cual permite ciclos de feedback más cortos y una mejor comprensión del entorno de ejecución por parte de lxs desarrolladores de aplicaciones.

Bee/LMR es nuestra implementación de Bee Smalltalk que se ejecuta sobre un entorno de ejecución metacircular vivo, en lugar de una VM tradicional. Se utiliza en producción para ejecutar una aplicación de 1,1 millones de líneas de código. Este sistema basado en LMR ha permitido a lxs desarrolladores de aplicaciones entender incrementalmente los componentes de la VM según sea necesario, e incluso modificarlos cuando sea necesario, como se argumenta con nuestros casos de estudio. Específicamente, mostramos que este enfoque evita la observabilidad limitada de la VM, el modo de desarrollo separado de la VM, y los largos ciclos de edición-compilación-ejecución presentes en la arquitectura tradicional de dos capas.

9.1. Trabajo Futuro

En trabajos futuros, queremos explorar cómo mejorar el rendimiento sin sacrificar la capacidad de cambiar el runtime en tiempo de ejecución. Si bien compiladores como Graal demuestran que los compiladores en lenguajes de alto nivel pueden producir rendimiento de última generación, nuestro diseño del formato de objetos y del recolector de basura requirió decisiones cuidadosas para preservar un sistema funcional en todo momento. Existen dependencias circulares similares en el compilador JIT, lo que significa que un cambio en tiempo de ejecución podría romperlo. Esto podría mitigarse mediante el soporte de múltiples versiones de tales subsistemas, donde la versión anterior y funcional permanece disponible como una alternativa.

También esperamos trabajar en un sistema de tipos que realice inferencia, tanto para proporcionar feedback de tipos al compilador JIT como para ayudar a mantener las clausuras de código del recolector de basura.

Otras direcciones de investigación futura podrían explorar cómo aplicar este diseño a otros lenguajes, o construir otros lenguajes sobre nuestra LMR. La pregunta clave de investigación aquí es cuáles son los compromisos adecuados entre permitir la programación viva y lograr un rendimiento práctico de tales sistemas, sin introducir una complejidad innecesaria.

También sería posible aplicar la infraestructura desarrollada para Bee/LMR para implementar una VM para otros lenguajes más estáticos como Java, dentro del entorno de programación vivo.

Bibliografía

[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.

[BK87] Frederick Brooks and H Kugler. *No silver bullet*. April, 1987.

[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.