Plataformas de ejecución de software reflexivas

Chari, Guido Martín

2017-12-13

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Plataformas de ejecución de software reflexivas

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

Lic. Guido Martín Chari

Director de tesis: Dr. Diego Garbervetsky
Consejero de estudios: Dr. Víctor Braberman
Lugar de trabajo: Departamento de Computación, Facultad de Ciencias Exactas y Naturales.

Buenos Aires, 2017

Fecha de defensa: 13 de Diciembre de 2017

Plataformas de ejecución de software reflexivas

Resumen: Las Máquinas Virtuales (MV) son artefactos de software complejos. Sus responsabilidades abarcan desde realizar la semántica de algún lenguaje de programación en particular hasta garantizar propiedades tales como la eficiencia, la portabilidad y la seguridad de los programas. Actualmente, las MV son construidas como “cajas negras”, lo cual reduce significativamente la posibilidad de observar o modificar su comportamiento mientras están siendo ejecutadas. En este trabajo pregonamos que la falta de interacción entre las aplicaciones y las MV impone un límite a las posibilidades de adaptación de los programas, mientras están siendo ejecutados, ante nuevos requerimientos.

Para solucionar esta limitación presentamos la noción de plataformas de ejecución reflexivas: un tipo especial de MV que promueve su propia inspección y modificación en tiempo de ejecución permitiendo de este modo a las aplicaciones reconfigurar el comportamiento de la MV cuando sus requerimientos cambian. Proponemos una arquitectura de referencia para construir plataformas de ejecución reflexivas e introducimos una serie de optimizaciones específicamente diseñadas para este tipo de plataformas. En particular proponemos aplicar técnicas de optimización especulativa, técnicas estándar en el contexto de los lenguajes dinámicos, a nivel de la MV misma.

Para evaluar nuestro enfoque construimos dos plataformas de ejecución reflexivas, una basada en un compilador de métodos y la otra en un optimizador de trazas. Luego, analizamos una serie de casos de estudio que nos permitieron evaluar sus propiedades distintivas para lidiar con escenarios adaptativos. Comparamos nuestras implementaciones con soluciones alternativas de nivel de lenguaje y argumentamos porqué una plataforma de ejecución reflexiva potencialmente las subsume a todas. Por otra parte, mostramos empíricamente que las MV reflexivas pueden ejecutarse con un desempeño asintótico similar al de las MV estándar (no reflexivas) cuando las capacidades reflexivas no se usan. También que la degradación del desempeño es bajo (comparado con las soluciones alternativas) cuando estos mecanismos sí son utilizados. Aprovechando nuestras dos implementaciones, estudiamos cómo impactan las diferentes familias de compiladores (por método vs. por trazas) en los resultados finales.

Por último, realizamos una serie de experimentos con el objetivo de estudiar los efectos de exponer el comportamiento de los módulos de compilación a las aplicaciones. Los resultados preliminares muestran que este es un enfoque plausible para mejorar el desempeño de aplicaciones sobre las cuales las heurísticas de los compiladores dinámicos producen resultados subóptimos.

palabras clave: maquinas virtuales, reflexión, compiladores dinámicos, compiladores de trazas, evaluación parcial, especulación, adaptación de software, evolución de software.

Fully Reflective Execution Environments

Abstract: Many programming languages run on top of a Virtual Machine (VM). VMs are complex pieces of software because they realize the language semantics and provide efficiency, portability, and security. Unfortunately, mainstream VMs are engineered as "black boxes" and provide only minimal means to expose their state and behavior at run time. In this thesis we argue that the lack of interaction between applications and VMs put a limit on the adaptation capabilities of (running) applications.

To overcome this situation we introduce the notion of fully reflective VM: a new kind of VM providing reflection not only at the application but also at the VM level. In other words, a fully reflective VM provides means to support its own observability and modifiability at run time and enables programming languages to interact with and adapt the underlying VM to changing requirements.

We propose a reference architecture for such VMs and discuss some challenges in terms of performance degradation that these systems may induce. We then introduce a series of optimizations targeted specially to this kind of platforms. They are based on a key assumption: that the variability of the VM behavior tend to be low at run time. Accordingly, we apply standard dynamic compilation techniques such as specialization, speculation, and deoptimization on the VM code itself as a means to mitigate the overheads.

To validate our claims we built two reflective VMs, one featuring a method-based just in time (JIT) compiler and the other running on top of a trace-based optimizer. We start our evaluation by analyzing a series of case studies to understand how a reflective VM could deal with unanticipated adaptation scenarios on the fly. Furthermore, we compare our approach with the existing language-level alternatives and provide an elaborated discussion on why a reflective VM would subsume all of them. Then, we empirically show that our implementations can feature similar peak performance of that of standard VMs when their reflective mechanisms are not activated. Moreover, they present low overheads (in comparison to existing alternatives) when VM's reflective capabilities are used. We also analyzed how the different compilation strategies (per-method vs. tracing) impact on the overall results. Finally, we conduct a series of experiments in order to study the effects of opening up the compilation module of a reflective VM to the applications. We conclude that it is a plausible approach that brings new opportunities for optimizing algorithms in which the compiler heuristics fail to give optimal results.

keywords: virtual machines, reflection, just in time compilation, tracing compiler, partial evaluation, speculation, softare adaptation, software evolution, dynamic adaptation.

No están quienes hubieran propagado, altivas y hasta el hartazgo, el título que a partir de este trabajo precede a mi nombre. Para ellas: Anita y Olguita.

Agradecimientos: Mucha gente colaboró conmigo, de muy diversos modos, en el transcurso de tiempo que me tomó desarrollar mi tesis doctoral. Algunos aportando ideas y/o enfoques fundamentales, otros facilitando las herramientas y/o el apoyo necesario. La mayoría simplemente con su valiosa compañía, ya sea en los momentos de trabajo (zozobra) como así también en los de ocio y regocijo. Aún corriendo el riesgo de, injustamente, olvidarme de algún nombre propio, me tomaré el trabajo de nombrar a todos en los párrafos que siguen.

En primer lugar debo mencionar a los que tomaron el rol de directores de este trabajo. Stéphane Ducasse y Marcus Denker, del RMoD participaron de la génesis de muchas de las ideas que esta tesis desarrolla. Los aportes técnicos de Stefan Marr son inestimables: sin su cooperación y entera predisposición hubiera sido imposible llegar a la meta. Finalmente mi gratitud para con Diego Garbervetsky, el único que me acompañó, en su rol de director principal, los 5 años que llevó el desarrollo total de la tesis. Diego me transmitió fielmente los conceptos metodológicos esenciales para llevar adelante un trabajo científico con rigor técnico y una absoluta honestidad intelectual.

LaFHIS, el laboratorio de investigación en ingeniería de software me apadrinó durante estos años. A pesar de no trabajar en sus temas de interés, desde el primer día y desinteresadamente, quienes dirigen el grupo me han brindado todas las herramientas necesarias para desarrollar mi trabajo. Además, sus consejos han sido siempre de una lucidez inusitada. Por eso mi sincero agradecimiento tanto para Sebastián Uchitel como para Víctor Braberman. No quiero dejar de mencionar investigadores del grupo que admiro y quienes me han brindado su tiempo y sabios consejos como Hernán M. (jurado de esta tesis) y JP.

En la oficina 12 he pasado muchos momentos inolvidables. Por allí pasaron: Shewu, Ferto, Estebán, Dipi, Hernán C., Dani B., Naty R., Casuterano, el Turco, Zoppi, Dani C., Vero, Christian, Vir, Iván y Naty D. En su mayor parte, hemos compartido alegrías, nos hemos divertido y nos sumimos en inagotables discusiones. El último período en la infoteca lo compartí con personas extraordinarias con quienes entablamos una amistad que estoy seguro perdurará en el tiempo: Rodri, Mariano y Ferni.

Luego están mis amigos de la vida. El grupo de Lajas, entre quienes se destaca Guille por siempre intentesarse y procurar entender de qué iba esta aventura. Luego esta *la banda del cafe con leche*, gente de exactas con quienes compartí innumerables veladas y viajes por el mundo en este tiempo: Pachi, Pocho, Fran, Marino, Quique. A ellos se suman Ger, Arti y el Manishe desde diversas latitutes. También Alfa quien me hospedó más de una vez en París.

Mis padres han sido un sostén de una calidad imponderable. Es plausible conjeturar que sin su apoyo incondicional el recorrer este sendero jamás hubiera sido siquiera concebido. Mi hermano es un ser extraordinario y por ende una influencia positiva en todo lo que llevo a cabo. También quiero agradecer a mis tíos Liliana y Horacio que siempre estuvieron y siempre estarán. Mis abuelas, Anita y Olguita, hoy andarían orondas con su nieto doctor. Anita aún me acompañaba cuando comenzó este camino.

Por último debo agradecer a Jan Vitek y Andrés Diaz Pace por haberse tomado el trabajo de leer y comentar este manuscrito.

Contents

List of Figures

List of Tables

Part I

Prelude

CHAPTER 1

Introduction

Most software systems evolve during their lifetime [MD08]. In many cases, the required modifications must be performed without interrupting their execution [PDF+15]. The problem of adapting systems while they are running can be approached at different abstraction levels. Language-level solutions are appealing for scenarios in which fine-grained adaptations are needed, i.e, whenever the granularity of the modifications is that of individual objects or methods. To support this kind of adaptations, a new class of dynamic languages was suggested a decade ago [NBD+05]. Essentially, these languages must include special software abstractions and tools supporting run-time evolution. One of the key elements proposed is that dynamic languages provide extensive and on-demand reflective capabilities.

Reflective capabilities are usually provided by managed languages that run on top of software artifacts known as virtual machines (VMs). VMs perform various tasks such as realizing the language's semantics, providing dynamic compilation, adaptive optimizations, automatic memory management and enforcing application's security. It was already identified that these low-level features conforms an inherent complex domain that tends to expose strong dependencies between its different submodules [WLB+15, HGA+09].

Beyond the inherent complexities, VMs must cope with demanding performance requirements. Consequently, today's industrial-strength VMs for mainstream languages are usually developed using languages and tools oriented to favor more performance than flexibility. This results in VMs that, after compilation, become optimized binaries, limiting the ability of applications to observe and adapt them at run time. Thus, while modern languages supporting run-time adaptation continuously demand more flexibility, VMs still tend to remain as black-box components, not accessible to the language and not adaptable at run time.

We believe the state-of-the-art regarding VMs negatively impacts the reflective capabilities provided by managed programming languages. For instance, languages such as Java, JavaScript, Python, and Smalltalk already provide language-level mechanisms allowing developers to partially observe and interact with the application at run time. These usually take the form of *reflective* APIs. However, they all still present limited reflective capabilities. Especially, when considering low-level concerns of a language such as its memory organization, optimization aspects, or the way its statements are executed. Even advanced (research-oriented) reflective solutions such as CLOS [KR91] and Partial Behavioral Reflection [TNCC03] expose this limitation.

To workaround some of these issues, multiple research projects explored the use of high-level languages for VM construction [AAB+05, WHVDV+13, USA05, CBLFD11]. This approach is appealing because developers can leverage modern programming techniques and principles to better deal with the complexity of VM's development. Some of these approaches are even metacircular, *e.g.*, implementing a Java Virtual Machine in Java. In addition to the use of high-level languages, aspects such as modularity, observability, and extensibility have been in the focus of the VM research community as well [WBLF12, GTL+10]. However, even the current metacircular approaches produce VMs that do not enable significant observability and interactivity at run time.

Summarizing, since VMs execute application code, they interact with application entities such as objects, methods, and statements. Furthermore, VMs must be aware of their most inner aspects to realize its features. However, this relation is unidirectional: applications have very limited control over how the VMs manage themselves. As a consequence, applications are usually designed to be unaware of the existence of the VM. Based on the fact that reflection has already been identified as a fundamental technique for software evolution [NDG+08], in this thesis we advocate that:

the new generation of VMs should promote a ***bidirectional*** *communication between themselves and the applications they execute based on a reflective architecture.*

Our main hypothesis is:

Hypothesis 1 (H1) *Software development would be improved in terms of adaptability if virtual machines enable an interaction with the applications they run. Especially when the applications need to deal with dynamic adaptations involving low-level features.*

To validate our hypothesis we propose the idea of fully reflective virtual machines: a kind of virtual machine exposing its whole structure and behavior to the applications at run time. Fully reflective VMs allow developers to observe and adapt the VM *on-the-fly* enabling from simple adaptations up to fine-grained tuning of applications. This provides developers with two ways of adapting a running application to changing requirements: by modifying the application code or by customizing an aspect regarding the application semantics, *i.e.*, any component/feature provided by the VM. At the same time, fully reflective VMs benefits VM developers by bringing them the possibility to program low-level components with instantaneous feedback, something rarely available nowadays.

This work explores fully reflective virtual machines taking into consideration feasibility, usability, and performance aspects. What follows is a summary of the main contributions of this thesis guided by the corresponding question we intend to answer.

1.1 Are reflective VMs better than alternative language-level approaches for software adaptation?

As an example for the potential of reflective VMs consider a server application that has to run without interruption. In case a security issue is found, one might want to use a custom security analysis to determine its impact with respect to application and user data. For safety measures, such an analysis on a live system needs to ensure that it does not modify any data, *i.e.*, that the security analysis is free from side-effects. Thus, the analysis should have only read access to the application data.

In this thesis we show that solutions based on a reflective VM provide the mechanisms to approach such scenario at the language level and without interrupting the whole system in a simple manner. Chapter 4 describes precisely the main characteristics that makes a reflective system to be considered a fully reflective virtual machine. Afterwards, we introduce MATE, a reference architecture we propose for building and exploring a particular kind of reflective VM for object-oriented languages. MATE's most salient aspect is a metaobject protocol (MOP) [KR91] as a means to operate, programmatically and at run time, on the structural and behavioral aspects of the VM. This MOP is responsible for providing the aforementioned bidirectional communication. By enforcing the interaction through a predefined API (MOP), we ensure that the VM reflective capabilities are used in a controlled fashion.

We then validate our hypothesis regarding adaptation capabilities. To perform the corresponding experimentation, we first present MATE$_{PE}$ and MATE$_{MT}$, two Smalltalk reflective VMs implemented on top of the Truffle [WWS^{+}12] and PyPy [BT15, BCFR09] frameworks respectively. Both implementations provide reflective capabilities for a subset of VM components considering both, structural and behavioral aspects. Their only difference is their compilation approach, discussed below. We then analyze how they handle a series of unanticipated adaptive scenarios requiring changes to both the application and its runtime environment. We consider security, optimization, and profiling scenarios. For instance adopting readonly references for protecting a module or customizing how the fields of objects are actually stored in memory to improve performance.

1.1.1 How reflective VMs compare with state-of-the-art language-level adaptation solutions?

We compare a reflective VM with the most established language-level approaches that support run-time software evolution. To do so, we started from a recent survey on self-adaptive systems by Salvaneschi et al. [SGP13] which asks: is it possible to adopt a single paradigm providing all required abstractions to implement adaptive systems? To answer this question, they evaluate contemporary reflective systems (RS), aspect-oriented programming (AOP) and context-oriented programming (COP). The authors identified strengths and weaknesses for all the approaches and conclude that there is no consensus on which of the existing language-level approaches to adopt.

To compare a reflective VM with the three approaches we analyze three nonfunctional generic adaptation scenarios regarding performance, security, and profiling aspects. All require complex behavioral adaptations orthogonal to the application logic. Despite paradigms like AOP and COP may enable to adapt a low-level aspect of the system in several cases, their main implementations promote mechanisms (pointcut languages, layers, etc) for intercepting execution points and redirecting their execution flow. We show that this presents drawbacks in terms of maintainability, debugging, and performance aspects of adaptations. On the other hand, most contemporary RSs present limited reflective capabilities regarding the runtime elements and thus the adaptations concerning these elements are not expressible, or can be achieved only indirectly.

We show that a fully reflective VM is the only solution that manages to approach all the scenarios. Furthermore, the adaptations does not present the aforementioned drawbacks. As such, we conjecture that reflective VMs could take the role of a uniform solution for adapting systems at the language level. To support our conjecture, we provide an elaborated discussion regarding why fully reflective VMs subsume the three alternative approaches.

1.2 What are the performance overheads of reflective VMs?

The most salient aspect of reflective VMs is that they provide reflective capabilities in every component and aspect of the VM. In MATE this flexibility implies a significant proliferation of indirections and conditions (guards) that need to be checked at run time to realize the semantics of the system. This suggests that in a reflective VM, the cost for flexibility would be payed in terms of performance overheads. That is the reason why after demonstrating their usefulness, we focus on the performance of reflective VMs.

Recently it was shown that speculative compilers can remove the run-time overhead for common usage patterns of reflection and even for simple MOPs [MSD15]. Since FREEs present a particularly challenging flexible model these performance results do not apply directly to them. However, they laid the groundwork for elaborating the following hypothesis:

Hypothesis 2 (H2) *Reflective VMs do not incur in significant overheads when its particular adaptive capabilities are not being used. When actually exploited, if the variability they introduce is low, the performance overhead is equal or better than the alternative approaches when the proper optimizations are applied.*

To validate this hypothesis we present an optimization model that mitigates the indirections caused by a reflective VM. To do so, the optimization model speculates on the fact that applications are going to present, usually, a stable behavior. This speculation is based on the analogous phenomenon observed in dynamic languages: the actual usage of their dynamic features is usually moderate or it follows fixed patterns. Our conjecture is that the variability caused by the adaptation of the VM would be low for most consecutive executions of the application's behavior. Accordingly, similar to the standard speculation techniques performed by just in time compilers for optimizing dynamic features, we rely on run-time deoptimizations for the cases when our assumptions do not hold.

We implement this model in both, Mate$_{PE}$ and Mate$_{MT}$, and assess their peak performance through an extensive validation regarding different aspects of our model. First, we analyse how our implementations stand in terms of performance in the programming language's (PL) landscape. To do so, we select an established set of benchmarks for comparing language implementations [MDM16] and run them with several PLs, including industrial-strength ones. The results show that our implementations present a mean overhead of 2.9x and 4.2x respectively with respect to Java (Hotspot VM), outperforming other Smalltalk implementations and achieving similar performance levels than mainstream language implementations such as JavaScript using V8 VM. Afterwards, with additional benchmarks we assess the inherent performance overhead of running applications in a reflective VM. We empirically demonstrate that reflective VMs can reach a similar performance to standard non-reflective VMs when no VM-level adaptations are needed. Concretely, the evaluation shows that a reflective VM implementing our optimization model has, on average, no significant peak performance overhead compared to a standard VM for the same language.

We continue by measuring the overhead of our implementations when their particular adaptive features are being exploited. Concretely we first measure the overhead incurred by our implementations when redefining the semantics of individual VM operations such as how to read a field or perform a method lookup. The results show that the peak performance overhead is negligible also for most operations. We then evaluate MATE$_{PE}$ and MATE$_{MT}$ in on-the-fly adaptation scenarios that need intervention at the VM level: introducing immutable (read-only) references [ADD+10] and profiling several aspects of the actual run-time behavior. For the more elaborated scenarios, the reflective solutions using the MOP introduce low overheads in comparison with existing language-level alternatives. Moreover, the MOP-based approach of a reflective VM is lightweight in terms of engineering effort.

We also use the fact that our implementations use different families of just in time compilers (method-base and trace-based) to analyze the impact of the compilation approach on this kind of particular systems. Finally we provide a comprehensive analysis of the performance impact of a reflective VM in the warm up of the system.

1.2.1 Can reflective VMs be used to improve the performance of applications?

Since JIT compilation overhead is included in the program's execution time, JIT compilers must resolve a tension between fast and optimal compilation. To solve this tension they rely on heuristics that decide what, when, and how to optimize. By definition, an heuristic do not always hit a maximum. As a last experiment we analyze how fully reflective VMs including reflective capabilities concerning the compilation module could provide developers with tools for improving the performance of their applications at run time.

We start by characterizing special kind of application's variability that make some scenarios challenging for the existing heuristic model. This variability exhibits the following characteristics: 1) appears as high variability to JIT compilers resulting in run-time overhead 2) application developers could know this variability follows a fixed pattern that makes optimizations suitable 3) pattern instances are application-specific.

To mitigate their overhead, compilation heuristics could be adapted whenever a new case is found. However, this requires changes to low-level and complex artifacts, such as the compiler and the VM. This is costly, and gathering the necessary information to recognize complex variability patterns might also harm the overall application performance. Especially for variability that is highly application-specific, a general solution at the compiler level might be too complex and may introduce unacceptable run-time overhead for other applications.

To overcome this situation, the context-oriented programming community [PFH16] already suggested the need of more flexible compilers. As a step further, Rompf already demostrated they improve the overall performance under dynamic scenarios [RSB+14]. The last hypothesis of this work is:

Hypothesis 3 (H3) *The reflective capabilities of a fully reflective virtual machine could be used to significantly boost application's performance, at run time, in cases where the performance is sub-optimal due to application-specific variability.*

In this thesis we partially validate this last hypothesis. We start by including compilation reflective capabilities to MATE$_{PE}$. Examples are reifications of elements such as dispatch chains, program specializations, application profiling, and code splitting. By exploiting these capabilities, developers can provide fine-tune optimizations and mitigate application-specific run-time overheads by providing additional information to the compiler about a program at run time. This has the advantage of enabling custom optimizations with a generic API, accessible at the language level, and applicable to different scenarios and applications dynamically. Finally we empirically demonstrate how to workaround some of the performance issues raised by application-specific variability scenarios. Concretely, we present running time reductions in two scenarios exhibiting application-specific variability, provide some preliminary conclusions, and discuss our plans to continue this line of research.

1.3 Contributions

Below we outline a summary of the main contributions of this thesis:

- A characterization of fully reflective virtual machines.
- A reference architecture for building reflective VMs featuring a MOP as a means for handling VM-level reflection from the application level.
- An empirical validation through a series of case studies that demostrates the feasibility and usefulness of a reflective VM as an alternative for handling dynamic software adaptation.
- An approach to optimize MOP-based reflective VMs based on the assumption that the actually observed local meta-variability at run time is low.
- Empirical evidence demonstrating that under several heterogeneous scenarios reflective VMs can run efficiently in terms of peak performance.
- The proposal of opening-up JIT compilers for a fine-tuning of performance under application-specific variability scenarios along with an API for managing several aspects of the compilation process from the application.
- Empirical evidence showing that an open JIT compiler could produce significant overhead reductions in scenarios including application-specific variability.

1.4 Artifacts

As a result of this thesis we also built a series of artifacts that we make available to the community:

MATE$_{PE}$: a fully functional reflective VM following the MATE architecture and implemented using the Truffle framework. Accordingly, it features a method-based JIT compiler. It supports the Smalltalk programming language and provides advanced reflective capabilities at both, the language and the VM levels. Additionally, it implements the optimization model we proposed in this thesis and runs efficiently both, when using its particular reflective capabilities and when not. MATE$_{PE}$ is an open source language implementation that can be found at: https://github.com/charig/truffleMate/tree/papers/phdThesis.

MATE$_{MT}$: another Smalltalk reflective VM featuring the same characteristics as MATE$_{PE}$ but implemented in top of the PyPy framework. Accordingly, it features a tracing JIT compiler. It is also an open source language implementation that can be found at: `https://github.com/charig/RTruffleMate/tree/papers/phdThesis`.

Reproducibility Kit: We provide a repository with instructions for reproducing all the experiments used to produce the empirical evidence that support this work. They can be found at `http://github.com/charig/MatePerformance/tree/papers/phdThesis`. It also includes the fresh data, and the R code for all the reports that produce the empirical content presented in this work.

1.5 Roadmap

Chapter 2 presents the necessary background required to understand and properly frame the contributions proposed in this thesis. This content is mainly related to reflective systems, dynamic adaptation, and dynamic languages.

Chapter 3 motivates the need of novel language-level abstractions for properly handling software evolution.

Chapter 4 describes the fundamental features a system must expose to be considered fully reflective VMs. In this chapter we also propose a reference arquitecture for building this kind of VMs along with its most salient properties. Finally, we present a VM-level MOP aiming at handling unforeseen dynamic adaptations.

Chapter 5 presents the most important characteristics concerning the design and implementation of two reflective virtual machines. Within this chapter we also depict an optimization model specifically targeted to this kind of platforms.

Chapter 6 presents an extension for the MOP introduced in chapter 3 so that reflective VMs provide the means to improve the perfomance reached by the JIT compiler.

Chapter 7 provides a comprehensive evaluation of the adaptive capabilities of a concrete reflective VM. The chapter compares our implementations with existing alternatives in a series of unanticipated scenarios regarding optimization, security, and profiling aspects. It also presents an overall discussion of our results. We discuss aspects such as the generality of our claims and several issues that raise in the process of performing the evaluation.

Chapter 8 evaluates empirically the optimization model we propose for boosting the performance of MATE$_{PE}$ and MATE$_{MT}$. It also presents an analogous discussion to the one in the previous chapter.

Chapter 9 presents an analysis of a comprehensive corpus of work related to ours.

Chapter 10 provides the general conclusions and discusses the future work.

1.6 Dissemination of Results

Some results presented in this thesis have been originally published by the authors in [CGMD15, CGM16, CGM17b, CGM17a]. At the time of this writing, the results presented in Chapters 3, 4 and 7 have been submitted to the Transactions on Software Engineering Journal (TSE). Also, a subset of the results presented in Chapters 5 and 8 are about to be submitted for consideration to the Transactions on Programming Languages and Systems Journal (TOPLAS).

1.7 Resumen en Castellano

Muchos sistemas requiren ser modificados sin que su ejecución se vea interrumpida [MD08, PDF+15]. Abordar esta problemática usando soluciones del nivel de lenguaje es beneficioso en los casos en que las modificaciones requeridas involucren una granularidad fina tal como puede ser la de objetos indivuales. Los mecanismos reflexivos fueron sugeridos para evolucionar sistemas en estos niveles [NBD+05]. Éstos son provistos, en general, por lenguajes que corren sobre máquinas virtuales (MVs).

Las MVs son responsables de realizar funcionalidades tales como la semántica del lenguaje, la compilación en tiempo de ejecución y la organización de la memoria. Estas funcionalidades son complejas y tienden a exponer dependecias fuertes entre ellas [WLB+15, HGA+09]. Además, las MV deben lidiar con requerimientos de desempeño exigentes. Por esto generalmente se terminan construyendo en forma de binarios optimizados que limitan la posibilidad de adaptación en tiempo de ejecución.

Nosotros creemos que este estado del arte de las MVs impacta negativamente en las capacidades reflexivas que proveen los lenguajes que corren sobre ellas. Sobre todo al considerar aspectos de bajo nivel como la organización de la memoria o la optimización de métodos. En resumen, las MVs ejecutan código y por ende deben interactuar y conocer los aspectos internos de las aplicaciones. Sin embargo, esta relación es unidireccional: las aplicaciones tiene un accesso muy restringido a los aspectos de la MVs. Como consequencia, las aplicaciones son diseñadas de forma tal que la MV es transparente.

La hipótesis principal de esta tesis es:

Hipótesis 1 (H1): *El software mejorará en términos de adaptibilidad si las máquinas virtuales permitiesen una interacción bidireccional con las aplicaciones que ejecutan. Sobre todo cuando las aplicaciones deben lidiar con requrimientos de adaptación que involucren aspectos de bajo nivel.*

Para validar H1 proponemos la noción de máquina virtual completamente reflexiva: un tipo de MV que expone toda su estructura y comportamiento a las aplicaciones en tiempo de ejecución. Ésto permite en tiempo de ejecución la adaptación de la MV a diversas granularidades. En el capítulo 4 describimos las características que conforman una MV completamente reflexiva. Luego presentamos MATE, una arquitectura de referencia y para demostrar factibildad presentamos dos prototipos: MATE$_{PE}$ y MATE$_{MT}$. Cada uno implementado con diversos mecanismos de meta compilación. Para validar la hipótesis estudiamos cómo estas implementaciones abordan una serie de escenarios de adaptibildad no anticipada que conciernen aspectos de seguridad, optimización, y monitoreo.

También comparamos las MV reflexivas con los sistemas reflexivos contemporános, la programación orientada a aspectos (AOP) y la programación orientada a contextos (COP). Exponemos como, a pesar de que en ciertos casos estas alternativas pueden lidiar con los escenarios de adaptabilidad, por naturaleza (y por la naturalza de sus principales implementaciones) AOP y COP promueven mecanismos the intercepción y redirección del flujo de ejecución. Esto presenta limitaciones en términos de la mantenibilidad, depuración y el desempeño de las adaptaciones. Finalmente discutimos por qué las MV reflexivas subsumen estas tres alternativas.

Respecto al desempeño, todo indica que la flexibildad de las MV reflexivas implica una penalidad significativa en los tiempos de ejecución. Sin emabargo, recientemente fue demostrado que los compiladores especulativos pueden mitigar las penalidades de algunos mecanismos reflexivos [MSD15]. Este resultado sentó las bases para la segunda hipótesis que se desarrolla en este trabajo:

Hipótesis 2 (H2): *Las MVs reflexivas no deben incurrir en penalidades significativas cuando sus capacidades reflexivas no son utilizadas. Cuando sí lo sean, si la variabilidad que introducen es moderada el desempeño general debe ser igual o mejor que el que presentan las soluciones alternativas siempre y cuando las optimizaciones adecuadas esten implementadas.*

Para validar H2 presentamos un modelo de optimización adaptativo basada en la conjetura de que las aplicaciones presentarán una baja variabilidad de sus mecanismos adaptativos. Lo implementamos y demostramos usando benchmarks establecidos [MDM16] que una MV reflexiva puede tener, en promedio, una penalidad no significativa comparada con las MVs estándar al analizar el desempeño asintótico. También demostramos que la penalidad de redefinir operaciones individuales de la MV es despreciable. Para el caso de las adaptaciones que requiren un uso extensivo de las capacidades reflexivas de la MV una MV reflexiva introduce penalidades razonablemente bajas en comparasión con las alternativas existentes. Finalmente, presentamos un análisis del impacto de una MV reflexiva en los tiempos de compilación iniciales.

Finalmente analizamos si las MV reflexivas pueden mejorar la performance de aplicaciones mientras están siendo ejecutadas. El problem se da porque los compiladores dinámicos utilizan heurísticas para determinar qué, cómo y cuándo compilar. Cuando las heuristicas no logran aproximarse a los valores óptimos, el desempeño de las aplicaciones se resiente. Esto da lugar a la tercera hipótesis que se desentraña en este trabajo:

Hipótesis 3 (H3): *Las MVs reflexivas pueden ser usadas para mejorar significativamente el desempeño de aplicaciones en los casos en que el desempeño de estas sea suboptimo debido a variabilidad específica de la aplicación.*

H3 es validada parcialmente en este trabajo. Para ello, incluímos capacidades reflexivas sobre algunos aspectos de compilación a MATE$_{PE}$ y demostramos empíricamente como usando estos mecanismos se puede mejorar el desempeño en ciertos escenarios donde la heurística de compilación produce resultados subóptimos.

Artefactos desarrollados:

- **MATE$_{PE}$**: una MV reflexiva basada en meta-compilación por evaluación parcial. Puede ser descargada de: `https://github.com/charig/truffleMate/tree/papers/phdThesis`.
- **MATE$_{MT}$**: Otra MV reflexiva basada en meta compilación por trazas. Disponible en: `https://github.com/charig/RTruffleMate/tree/papers/phdThesis`.
- **Kit de Reproducción**: Proveemos un repositorio con instruciones para reproducir todos nuestros experimentos en: `http://github.com/charig/MatePerformance/tree/papers/phdThesis`.

CHAPTER 2

Background

This chapter introduces a number of concepts referenced throughout the rest of the thesis.

2.1 Reflection in Programming Languages

Definition 2.1 ***Reflection*** *is a mechanism for programs to express computations about themselves enabling the observation (introspection) and/or modification (intercession) of their structure and behavior [Smi84].*

Reflective capabilities have been widely exploited to build different kind of frameworks, libraries, and domain-specific languages (DSLs). Recent examples include concurrency frameworks [MD12], object-centric debugging [RBN12], dynamic proxies [VCM13, WNTD14], profiling DSLs [BNRR11], and libraries for dealing with unanticipated software adaptation [RDT08] among many others.

Depending on the programming language capabilities, a reflective computation may be done using particular idiomatic approaches. For instance, several techniques for interceding semantics with before/after behaviors are subtle variants of the general concept of a proxy [BFJR98, Duc99, WNTD14, VCM13]. Another alternative is that of aspect-oriented programming (AOP) which provides dedicated pointcut languages for dealing with reflection [KLM^{+}97]. But to promote a modular, simple, and safe usage of reflective capabilities, a reflective architecture is needed.

Definition 2.2 *A programming language is said to have a* ***reflective architecture*** *if it provides tools for reflective computations explicitly [Mae87].*

Several reflective solutions already provide a fixed set of classes describing the structure and behavior of other language-level concepts such as (base-level) objects and methods. These are called metaclasses and their instances metaobjects [Tan09]. Metaobjects and the corresponding baselevel objects they describe must be *causally connected*: changes in metaobjects must lead to a corresponding effect upon their associated base-level objects accordingly [Mae87].

From an architectural point of view, the usage of metaclasses organizes applications into abstraction levels. Each level describes and controls the level immediately below to which it is causally connected. The set of metaobjects that represents a particular object constitutes the object's meta-level. The metaobjects describing all the base-level objects in an application compose the application's meta-level.

Definition 2.3 ***Metaprogramming*** *is the act of developing software features relying on metaobjects.*

Definition 2.4 *Whenever the metalevel is open to the application developers, its set of classes as well as the conventions for its usage conforms a* ***metaobject protocol*** *(MOP) [KR91].*

MOPs provide the means to modify the language's behavior and implementation by metaprogramming in a controlled manner. They are an elegant solution for handling non-functional aspects of applications [BC89, ABB+89, Coi87, MD12]. Approaches such as Iguana/J [RC02], Object-Centric Debugging [RBN12], and Albedo [RRGN10] even adopt MOPs as a way to deal with low-level concerns.

To improve aspects of distribution, deployment, and general purpose metaprogramming Bracha and Ungar [BU04] proposed the following *Mirror* design principles: i) *Encapsulation:* MOPs should not expose implementation details ii) *Stratification:* MOPs should enforce a separation between the application behavior and the reflective code. iii) *Ontological correspondence:* the meta-level reifications[1] must map one-to-one to concepts of the base-level domain.

2.1.1 Characterizing Reflective Computations

Reflective computations are commonly distinguished based on two orthogonal categories: whether they are used for *introspection* or *intercession* and whether they reflect on *structural* or *behavioral* entities. Introspective computations are those that only gather information from an entity while interception is used whenever a modification of the entity is desired. Analogously, a reflective computation is structural whenever it concerns the structure of the program while it is behavioral if its base-level operation deals with the operational semantics of the language. Gathering the name of a class is an example of an structural introspection while modifying the method lookup algorithm for an object is a behavioral interceding example. The four combinations are possible.

However, these two standard categories does not consider reflective operations at different abstraction levels. For example, adding fields to an object as well as modifying its memory location are both characterized as structural intercession. However, they deal with different abstraction levels. The first refers to application-level (object fields) concepts while the latter to VM-level (memory) concepts. Besides, it is common in reflective languages such as Smalltalk and JavaScript to support the addition of fields at run time while most languages do not support the customization of memory positions. Since to precisely characterize reflective environments it is important to distinguish the abstraction level of reflective operations we introduce a new orthogonal category:

Definition 2.5 ***Application-level*** *reflection refers to metaprograms that work with objects, classes, methods, or object fields of the application's domain model.*

Definition 2.6 ***VM-level reflection*** *refers to metaprograms regarding the operational semantics, execution stack, layout, method lookup, memory management, or any other reification concerning low-level aspects handled by the VM.*

[1]To reify: to model as a first class entity.

2.1.2 Reflective Dimensions

Beyond characterizing reflective computations this work needs means for comparing reflective systems. To the best of our knowledge, there is currently no established approach to assess the degree of reflective capabilities supported by a system. Consequently we propose a novel terminology distinguishing dimensions of reflection usually scattered throughout literature:

Definition 2.7 ***Domain-breadth*** *measures how many entities of a domain are actually reified (inter-entity) as well as how many features are included in the reification of each entity (intra-entity).*

Definition 2.8 ***Domain-depth*** *measures the number of meta-levels reified for each entity.*

To illustrate both ideas consider the concept of a *class*, a standard entity in object-oriented languages. Intra-entity domain-breadth refers in this case to the reification of features from classes such as the instance/class variables, methods, and inheritance relationships. On the other hand, domain-depth considers the levels of *metaclasses* available for a programmatic interaction. For instance, in a language with advance reflective capabilities such as Smalltalk, the user can inspect the metalevel of the metalevel, *e.g.*, the metaclass of a metaclass.

Definition 2.9 ***Full reflection*** *refers to the complete reflective coverage of both the domain-breadth and domain-depth aspects of reflection.*

2.1.3 Reflective Challenges

Traditionally, developers implementing reflective systems have had to deal with two main problems: performance and metaregression. Below, a more detailed description of each.

Performance

Definition 2.10 *Intercession handling (IH) is the mechanism that is responsible for interceding an operation and delegate the execution to the corresponding behavior implementation.*

Although computational reflection is a useful techniques for several software scenarios, most languages provide limited reflective capabilities because it is difficult to efficiently execute reflective programs. Efficiency is actually more challenging to achieve in the case of behavioral reflection. The reason is that to make a behavioral feature reflective, an *intercession handler* must be added to the system. Whenever the corresponding feature is going to be executed, the handler tests first whether the behavior has been customized by the application [TNCC03]. If that happened, instead of executing the feature, the handler must delegate its execution to the corresponding language-level entity (the metalevel). This enforces the causal connection but has a cost in terms of performance. Furthermore, whenever a shift to the metalevel occurs the VM must take care of translating its data representations and those of the language back and forth.

Definition 2.11 *An intercession handling site refers to any place of the system where the intercession handling is actually triggered.*

This suggests that incorporating more reflection into a system, *i.e.*, making it more complete, increases the flexibility at the cost of affecting its performance. When designing a reflective system this tension must be resolved. Nevertheless, it is worth noting that recent research suggests that optimization approaches based on speculation and dispatch chains can significantly mitigate the overheads of some specific intercession handlers whenever the usage of the metalevel is moderate and exhibits stable usage patterns [MSD15].

Metaregression

The metaregression problem [CKL96] happens in any reflective architecure in which metaobjects are handled as standard objects. This essentially means that executing code on metaobjects is not different to any execution occurring at the base level of the application. Metaregressions appears whenever the code in a metaobject call a base-level behavior which is, again, redefined by itself. As a consequence, an endless recursion is triggered.

To solve this problem, Denker et al. [DSD08a] proposed to represent the level of execution as a first class notion. This provides an extra parameter for defining the semantics of the operations of the system and enables to avoid the metaregression problem. Under this model, language implementers may provide different levels of restriction, blocking thus the redefinition of base-level operations when executing the metalevel.

2.2 Software Adaptation

It has become common for software systems to require or benefit from dynamic adaptation, i.e., to modify their behavior while they are running [PDF$^+$15]. Even sustainability issues nowadays require special software treatment which includes software evolution techniques [CCR15].

Definition 2.12 *Dynamic Software Updating (DSU) [HN05] represents the overall techniques for adapting applications without interrupting them.*

Among the existing approaches to DSU, language-level solutions are appealing for scenarios in which fine-grained adaptations are needed, i.e., when the granularity of the modifications is that of individual objects. Also for small applications where an architectural solution based on complex middleware is overkill.

Ideally, a solution providing dynamic adaptation capabilities must be flexible, robust, simple to use, and should introduce as less overhead as possible [HN05]. To deal with these trade-offs DSU could be approached at different levels of granularity: parameter, method, aspect, component, application, architecture, system, and data center [SGP13].

Definition 2.13 ***Anticipated adaptations*** *are adaptations identified at designing time.*

Definition 2.14 ***Unanticipated adaptation*** *are adaptations that appear at run time in systems that were not designed for dealing with them.*

When anticipated adaptations appear, the system is ready to cope with them. On the other hand, unanticipated adaptations could require the modification of functional aspects of an application that was not designed for dealing with the adaptation. It is also common to face with non-functional adaptations after deploying applications such as those related with monitoring, property enforcement, or the improvement of performance. These adaptations usually consist of complex behavioral variations, which are orthogonal to the application logic and should be additionally embedded in fine-grained locations.

In this work, we are interested in language-level adaptions which conforms the modifications at the finer-grained levels, *e.g.*, parameters, methods, aspects, and even individual objects. Additionally, from this particular adaptations we are mainly interested in unanticipated adaptations which are usually much more challenging than those anticipated.

2.3 Dynamic Languages

Definition 2.15 *A type system is a set of rules that assigns a property called* type *to the various constructs of a computer program [Pie02].*

Type systems guarantee that a program only computes expected values. To enforce correctness, the type system can statically analyse the source code or dynamically check run-time values. Implementations honoring the first policy are known as static type systems while approaches following the latter are known as dynamic type systems.

Pros of static typing are widely known and can be summarized as formal documentation, early error detection, and the ability to leverage on type information for improving performance [Bra04]. However, for several scenarios these advantages may not pay off. The reason is that type systems may be too restrictive and reject valid programs, they may be unsound, or they may hinder software evolution while adding unnecessary complexity [Tra09]. On the other hand, dynamic typing is well-suited for prototyping and managing complex pieces of software needing continuous evolution.

In the context of this work, with the term dynamic language we refer to those programming languages providing significant reflective capabilities and automatic memory management [JL96]. Regarding reflection, at least they should allow to modify code and objects at run time. Since these capabilities are limited in statically-typed languages [NBD+05], we will stick to dynamically-type languages when using this term.

The most influential examples of this type of languages were historically Lisp [McC60] and Smalltalk [GR83a] for the functional and object-oriented domains correspondingly. More recently, Python [VRDJ95], Ruby, and JavaScript took the role of mainstream representatives.

2.3.1 Execution Environments

Definition 2.16 *An* Execution Environment *is a set of software components in charge of executing programs written on a specific programming language.*

Most dynamic languages are supported by execution environments. Particularly, in this work we are interested in execution environments for object-oriented (OO) languages. For instance, an execution environment for an OO language is responsible for executing statements, defining and managing the representation of objects in memory, collecting the objects that are no longer used, and enforcing security properties, among others. Following Simth and Nair taxonomy, EEs are *process* virtual machines [SN05]. However, the programming language community usually refer to them just as virtual machines (VMs) or managed runtimes. We prefer the term execution environment to avoid confusion with system virtual machines such as emulators. Anyway, hereinafter, we will use the term VM to follow the community.

2.4 Optimizing Dynamic Languages

The main strength of dynamic language is their advantages in terms of productivity and flexibility in the context of dynamic requirements [NBD+05]. This has a cost which is payed, in general, in terms of run-time performance. There is little doubt that, in practice, equivalent programs in dynamically typed languages are slower than in statically typed languages [Tra09]. Dedicated experiments with optionally typed languages support this performance statement [CMS+11].

The main reason for this performance degradation is that the type of values are unknown until execution. Hence, instead of operating directly on values the VM must operate on special values that *box* or *wrap* the actual values. To overcome this, several extensions exist for typing or infering the types of some dynamic languages [BAT14, BG93, AACM07].

With the advent of mainstream dynamic languages such as JavaScript, this performance gap has been significantly reduced. The seminal work promoting particular optimizations for dynamic languages such as dynamic type analyisis, specialization, speculation, and splitting, among many others, was presented by Chambers et al. many years ago [CU91]. Below, an outline of the main techniques that has historically contributed to narrow this gap.

2.4.1 Dynamic Compilation

Definition 2.17 *a Just-In-Time (JIT) compiler translates the source code of a program to another representation (usually machine code) while the application is already running.*

Since the techniques pioneered by the Self prototyping language [CUL89] most dynamic languages are supported by sophisticated JIT compilers to run efficiently [Ayc03]. Modern JIT compilers usually perform aggressive inlining, and after inlining they apply classic compiler optimizations such as constant folding, dead code and common subexpression elimination, and scalar replacement [KWM+08]. According to Tratt [Tra09], the difference in speed between dynamically typed language implementations with and without JIT compilation is typically a factor of three to five.

Even some not so fully dynamic languages, such as Java supported by the Oracle's HotSpot VM [PVC01], use JIT compilation. The reasons are twofold: portability and performance. Regarding portability, programs can be delivered in an intermediate representation such as bytecode, and later a JIT compiler translates it to the corresponding underlying platform format at run time. On the other hand, JIT compilers based their optimization decisions on an analysis of the actual runtime behavior of the program, which provides a more accurate view of the common execution paths. Hence, more speculative optimization opportunities appear in comparison to ahead-of-time (AOT) compilation, which is based on an static analyses of the code.

The main issue with JIT compilers is that since the compilation overhead is included in the program's execution time, they must resolve a tension between fast and optimal compilation. To workaround this tension, several JIT compilers adopt a multi-tier approach, resorting to a stack of compilers with different optimization levels [SYK+05]. The VM starts executing applications by using an interpreter, or code generated by a baseline compiler which does not perform aggressive optimizations. At run time, the JIT compiler collects information about the most common executed behavior and hands it to a second-level compiler. This second compiler weighs more the efficiency of the resulting native code. Since ideally the generated code should result in more benefits than the time lost in performing the optimizations, the JIT decides which part of the program to actually compile (and when) based on complex heuristics and budget models [Kul11]. Note that this two-tier model can be generalized to more than two compilers.

Method vs. Tracing. Nowadays, the mainstream approaches for JIT compilation can be separated into two main families: method-based [PVC01] and trace-based [GES+09]. There is no clear winner between these two approaches. Trace-based compilers have been proven efficient in the context of functional languages like Racket [BBH+15]. They also have been satisfactory applied to web applications [CSR+09, Spi, Hc11] and general purpose languages such as Python [BCFR09]. On the other hand, the Java HotSpot VM [PVC01] and the V8 JavaScript engine [Goo] are the most prominent examples of efficient method-based JIT compilers. More examples include fastR [SWHJ16] and JRuby+Truffle [MSD15].

Tracing compilers builds on two basic assumptions: most of the execution time of a program is spent in loops, and several iterations of the same loop are likely to take the same path through the program. Accordingly, tracing compilers identified somehow this *hot* loops and then compile (and optimize) them. The compiler add guards to the compiled code to ensure that at run time the actual conditions of the program comply with the assumptions taken for the current compiled trace. On the other hand, method-based compilers just detect and compile *hot* methods. To further optimize the most common execution paths, they exploit techniques such as specialization and method inlining.

2.4.2 Specializations

Definition 2.18 *Specialization is a form of strength reduction program transformation on generic implementations. In a JIT compiler setting it is an optimization that selectively executes an optimized subset of the code by fixing the value of a variable [CU89].*

As already mentioned, the flexibility of dynamic languages results in a richer semantics which, in turn, leads to a performance degradation caused by three main sources:

1. Operators in dynamic languages can often be applied to a wide variety of operands. Hence, without precise type information, the compiler must emit generic machine code that widely performs type checks for dealing with all the potential type combinations.

2. Analogously, every message sent could result in the execution of very different methods at run-time because of method polymorphism [Nie89].

3. Field accesses could be done to heterogeneous objects. Moreover, in languages such as JavaScript, the same object could change its fields during program execution. As a consequence, for every field access the actual memory position must be computed at run time.

Nevertheless, there exists empirical evidence exposing that, at run time, programs are usually not as dynamic as they could be and their behavior tend to stabilize [DS84, HCU91, MSD15, HH09, MDM16, GDGC94]. Concretely, the types of the objects actually observed at each access/call site tend to be, in most cases, relatively low. Based on this fact, JIT compilers for dynamic languages mitigate these sources of overhead by using specialization.

For example, in dynamic languages like JavaScript and Python operations such as + could describe numerical addition, string concatenation, or whatever semantics a user defined for its objects. However, an specialized version would directly execute an integer addition in all the cases the compiler could guess it is only executed with integer parameters. Of course, specialization is only effective if the VM can guarantee that the specialized code will only be run on values of the appropriate types.

It is worth noting that there also exist studies showing that, unlike benchmarks, it is not negligible the relying of real programs on dynamic features [RLBV10, RLZ10].

2.4.3 Speculative Compilation

Definition 2.19 ***Deoptimization*** *is a technique used by JIT compilers that enables the system to invalidate the native code while it is being executed and jump to another version of the same code [HCU92].*

In dynamic languages, JIT compilers perform specializations by speculating on the most common observed values [CU91, CUL89]. Concretely, JIT compilers collect information while the system is running. Based on these information they generate specialized code guarded by a condition that ensures that the assumptions hold at run time. If the guard fails, the code needs to deoptimize, *i.e.*, to bail out to the runtime or jump to more inefficient code. Ideally, speculative optimizations will always turn out to be right and provide performance gains that outweigh the compilation costs. But, in reality, some speculations will turn out to be wrong triggering deoptimization and further recompilations [ZBB17].

There are several possible options on which to specialize dynamic languages. The most relevant in terms of perfomance impact is the speculation on types [KRR$^+$13]. This enables to save the levels of indirection the VM must follow because it does not know the types before run time. It also enables to work with primitive types whenever possible and avoid the boxing and unboxing of this values into objects. Another common speculation is on conditional jumps. Concretely, the JIT compiler could replace low-probable branches with a deoptimization trigger to avoid compiling uncommon paths. This may provide more optimization opportunities and may result in better instruction cache performance since the resulting code is smaller.

Two main methods exist for gathering the required information to produce specializations: type inference [Age96] and type feedback [HU94]. There is no clear winner between them. But feedback-driven (or profile-driven) optimizations are necessary in the context of dynamic languages. The main reason is that run-time profiles reflect the actual behavior of the currently executing program providing key information for performing adaptive optimizations. However, evidence suggests that combining both increase the benefits while decrease the overheads [KRR$^+$13]. Accordingly, Firefox's Spidermonkey JavaScript VM adopted this hybrid approach [HG12].

Definition 2.20 ***Adaptive Optimization*** *is a compilation mechanism for recompiling application's code at run time based on profiling information [HU96].*

Adaptive optimizations recompile code whenever an assumption fails based on the new behavior the profiling has observed. This way, whenever the program enters a different execution phase, the dynamic compiler has the possibility to adapt its compiled code with a newer version specifically optimized for the new phase. As a consequence, the compiler produces higher-density code for the common code paths for each phase. It is also used for recompiling hot methods with more aggressive optimization techniques.

Finally, to avoid using resources for recompilation whenever it will not provide significant benefits, the compiler provides the VM with details on how to deal with deoptimizations. This information is typically provided in form of parameters passed to the invocation of the deoptimization trigger routine in the generated code. These parameters may suggest the VM to execute a non-optimized version of the code if the compiler knows its improbable to produce a more optimized version. It also may tell the VM if the deoptimization was caused by an insufficient profiling information so the VM can profile this part of the code again before triggering a recompilation. To avoid an endless cycle of recompilations whenever an application changes phases frequently, per-method counters are installed. After hitting a threshold they are never compiled again.

To conclude, combining all these techniques and incorporating also key compiler optimizations such as aggressive inlining, a JIT compiler can produce native code that get rid of most of the abstractions of dynamic languages resulting in performance overheads.

2.4.4 Object Representation in Dynamic Languages

Definition 2.21 ***Dynamic Objects*** *are objects that enable the addition and removal of fields while the application is running.*

Dynamic objects exist in dynamic languages such as JavaScript, PHP, or Python. Beyond the customization of fields, the type of their values is not known statically. To optimize the representation of such a dynamic structure of unknown size and shape, the notions of maps [CUL89] and object *shapes* [WWB+14] have been proposed. Essentially, they keep track and cache object structures and field types at run time. This information enables a record-like in memory representation, where field accesses can be mapped to direct memory accesses, performing an speculative type specialization. It has already been shown that the design of the object model impacts significantly on the speculations the compiler can perform on types [ACS+14].

2.4.5 Dispatch Chains

Definition 2.22 *Polymorphic inline caches (PICs) [HCU91] are a technique for optimizing dynamic languages by recording type information and caching method lookup results to minimize their run-time overhead.*

Dispatch Chains generalize PICs to generic operations, such as object field accesses (for dynamic objects) and metaprogramming, caching arbitrary values [WWB+14, MSD15]. Dispatch Chains rely on the stability and low variability of run-time behavior. Each element caches a value (*specialization*) and has a *guard* (speculation) to test its validity in the current run-time context. If not valid, the following element in the chain is tested. Finally, dispatch chains are structured so that the last element implements the fallback behavior, i.e., the behavior for the most general case.

2.5 Interpreters

Interpretation is one of the simplest approaches for implementing programming languages. For instance, in bytecode-based languages, the behavior of the language can be implemented with a loop that first decodes each bytecode, and then calls a procedure defining its corresponding behavior. In an abstract syntax tree (AST) interpreter, it is only needed to add an execute method to every AST node generated by the parser. But this simplicity comes with a cost: interpreters exhibit a significant performance overhead (between 1-2 orders of magnitude) in comparison with highly-optimized VMs based on JIT compilation.

Definition 2.23 *Self-optimizing interpreters rewrite themselves into more specialized version at run time based on observed types and values [WWS+12].*

To mitigate these overheads, interpreters can also specialize themselves with type inference or profiling information. In self-optimizing interpreters, analogous to speculative compilation, the specialized versions make assumptions about different values and if the assumptions do not hold at run time, they are replace again with versions that can handle a more generic case.

2.5.1 Meta-Compilation

Definition 2.24 *Meta-tracing is the mechanism that performs JIT compilation of traces of the execution of an interpreter for a particular user program instead of traces of the user program itself [BCFR09].*

Definition 2.25 *Partial evaluation (PE) of an interpreter is the process of deriving executable code from partially evaluating the guest language interpreter (code) combined with the interpreted program (data) [WWH+17].*

The main idea behind meta-compilation is to run a self-optimizing interpreter enough times until it reaches an stable state. Concretely until it already specialized itself as much as possible. At that point, optimized compiled code is derived from the current version of the interpreter. Essentially, the compiler compiles an already optimized interpreter for the current application (including its input and observed values) instead of the application itself. This makes the compiler of the interpreter a reusable meta-compiler that can be used for different language implementations.

In recent years, meta-tracing and partial evaluation became suitable meta-compilation techniques exhibiting very good performance results for a wide variety of language implementations [MSD15, BBH+15, SWHJ16, WWH+17, RP06, BKL+08]. This evidence suggests that meta-compilation overcomes some issues found when trying to target a dynamic language to an existing efficient static compiler [CEI+12]. Finally, note that empirical evidence also suggests that both approaches, meta-tracing and partial evaluation, present similar performance results [MD15].

2.5.2 Meta-tracing

Meta-tracing generates a trace of execution after running the interpreter of a program for a while. The resulting traces are the units of compilation. Based on frequently executed loops on the application level, the interpreter records a concrete path through the program, which then can be optimized and compiled to native code. Since traces span across many interpreter operations, the interpreter overhead can be eliminated completely and only the relevant operations of the application remain.

The fundamental difference between meta-tracing and non-meta-tracing JITs is that the latter must be manually written [BT15]. With meta-tracing, a JIT compiler becomes a reusable meta-compiler that can be used for different language interpreters. PyPy was the first practical solution for building efficient interpreters based on meta-tracing techniques [BR07, BT15, BCFR09].

2.5.3 Partial Evaluation

The overall techniques for making PE practical in the context of self-optimizing interpreters are fully descibed in [WWH+17]. The paper also exhibits several languages implemented with this approach reaching the performance of state-of-the-art VMs. For instance, implementations of JavaScript reach (on average) the same peak performance as V8 [WWH+17] for a set of language benchmarks. This implies an overhead of aprox. 2x compared to Java HotSpot. Furthermore, implementations of Ruby and R outperform the faster existing implementations, JRuby and GNU R respectively.

Truffle [WWS+12, WWW+13] is a platform for implementing self-optimizing AST interpreters. To realize self-optimizing AST interpreters [WWS+12], Truffle comes with a DSL to specify specializations. When used along with the graal JIT compiler [DWS+13, SWU+15], the approach derives optimized compiled code from the interpreters using partial evaluation [Fut99]. Analogous to PyPy, the partial evaluation approach can be reused with different interpreters. Conceptually both approaches follows the first Futamura projection for realizing compilers by partially evaluating an interpreter [Fut83].

Several type specializations can be defined for every operation along with a fall back mechanism for executing the generic behavior. For instance, the DSL enables to perform a native primitive addition when both arguments are integers instead of relying on a general language level method that would consider an arbitrary type for each argument. If in subsequent executions the parameters observed at run time are no longer integers, then the interpreter rewrites itself to handle the more generic but unoptimized version of the operation. This optimizes the interpreter code executed for a specific program and minimizes run-time checks because it only needs to test whether the run-time values comply with the already observed types.

2.6 Resumen en Castellano

El capítulo introduce una seria de conceptos que son necesarios para una cabal comprensión de la tesis. Comienza con la noción de reflexión en los languajes de programación, un mecanismo para que los programas puedan expresar computos sobre ellos mismos [Smi84]. Para un uso modular, simple y seguro de estos mecanismos se require de una arquitectura reflexiva [Mae87]. Las arquitecturas más referenciadas están basadas en metaclases y metaobjetos [Tan09]. Los metaobjetos describen el comportamiento de otros objetos con los cuales están enlazados: cambios en los metaobjetos provocan efectos inmediatos en los correspondientes objetos de nivel base [Mae87]. El conjunto de metaclases y las interfaces para su uso conforman lo que se conoce como un protocolo de metaobjetos (MOP) [KR91].

Los computos reflexivos suelen distinguirse en base a dos categorías ortogonales: si son introspectivos o intercesivos y si *reflexionan* sobre la estructura o el comporamiento del sistema. Sin embargo, estas dos categorías no consideran el nivel de abstracción del computo. Dado que para esta tesis esa noción es relevante, introducimos una tercer categoria: los computos reflexivos del nivel de aplicación son aquellos que trabajan con objetos, clases y métodos. En cambio los de nivel de la MV trabajan con la semántica operacional, la pila de ejecución, el layout de objetos, el *lookup* de métodos, o el manejo de memoria.

También proponemos una nueva terminología para distinguir diversas dimensiones de las soluciones reflexivas. El *ancho de dominio* estima tanto la cantidad de entidades de un dominio que son reificadas así como también la reificación de conceptos internos de cada entidad. Por otro lado, la *profundidad de dominio* considera la cantidad de meta niveles que se reifican por cada entidad. La reflexividad completa se da cuando una solución cubre todos los aspectos de ambas dimensiones. Lamentablemente la completitud está en tensión con el desempeño: cuantas más capacidades reflexivas incluye una solución, más penalidades en términos de desempeño emergen.

Respecto a la adaptabilidad se analizan las alternativas para adaptar software de manera dinámica, es decir sin ser interrumpido [HN05]. Dentro de este grupo, en esta tesis nos interesan las soluciones a nivel de lenguaje, adecuadas para adaptaciones de trazo fino. Además, nos interesan las adaptaciones que no hayan sido identificadas al momento de diseñar el software. Este tipo de adaptaciones afloran en tiempo de ejecución y son generalmente más complejas de abordar.

Luego se introducen los lenguajes dinámicos. En el contexto de este trabajo, lenguajes dinámicos son aquellos que proveen considerables capacidades reflexivas y menajo automático de memoria [JL96]. Esto generalmente es provisto por lenguajes dinámicamente tipados [NBD+05] que corren sobre MVs.

Los lenguajes dinámicos acarrean serias dificultades para ser implementados de manera eficiente. La semilla para mitigar este problema la introdujeron Chambers y otros [CU91] hace muchos años. El elemento principal son los compiladores de tiempo de ejecución, los cuales traducen los programas a código nativo mientras la aplicación ya está siendo ejecutada [Ayc03]. La principal limitación de este tipo de compiladores es que el tiempo que tardan en hacer la traducción está incluido en el tiempo de ejecución del programa. Por eso, deben resolver una tensión entre compilación rápida y compilación óptima.

Existen dos grandes familias de compiladores en tiempo de ejecución: compiladores por métodos [PVC01] y compiladores por trazas [GES+09]. Los compiladores por trazas asumen que la mayor cantidad del tiempo el programa corre en un ciclo y que el computo dentro del ciclo recorre siempre las mismas ramas del flujo de control. Por eso, estos compiladores indentifican esos ciclos y optimizan agresivamente solo el código de las trazas más usadas. Por otra parte, los compiladores por método solo detectan y compilan los métodos más usados.

Para mejorar la eficiencia de los programas los compiladores dinámicos suelen *especializar* el código para ciertos parámetros. Cuando no se puede probar que los parametros siempre serán los adecuados la especialización se considera especualtiva [CU91, CUL89]. En el caso de que en tiempo de ejecución los parametros no sean del tipo de la especialización correspondiente, el compilador debe deoptimizar [HCU92] el código. Idealmente esto nunca sucede, pero los resultados empíricos muestran que esto pasa con frecuencia [ZBB17].

Las optimizaciones adaptativas recompilan el código cada vez que alguna especulación o presunción hecha no se cumple en tiempo de ejecución [HU96]. En resumen, combinando todas las técnicas presentadas, e incorporando las optimizaciones más relevantes del dominio de compilación tal como el *inlining* de métodos, un compilador dinámico puede eliminar del código nativo que produce la mayoría de las fuentes de penalidad que los lenguajes dinámicos introducen.

Otra alternativa para implementar lenguajes dinámicos es por medio de intérpretes. Estos son mucho más simples pero exhiben una penalidad en terminos de desempeño de entre uno y dos órdenes de magintud en comparación con los compiladores dinámicos más optimizados. Para disminuir esta distancia, los intérpretes auto-optimizables reescriben el código intermedio del programa a interpretar basado en los valores observados en tiempo de ejecución [WWS+12] suscitando una especialización a nivel de interpretación.

Un paso más adelante, la meta-compilación corre un interprete auto-optimizable suficientes veces hasta que alcance un estado estable. En ese momento se compila el interprete que se deriva de la versión actual del programa. De esta manera el compilador dinámico es reutilizable para diferentes interpretes y corridas del programa.

En los últimos años meta-trazas y evaluación parcial se convirtieron en las técnicas de meta-compilación principales exhibiendo muy buenos desempeños [MSD15, BBH+15, SWHJ16, WWH+17, RP06, BKL+08]. La diferencia fundamental entre compilación de trazas y de meta-trazas es que en el primero el compilador debe ser escrito manualmente [BT15] mientras que en el segundo caso el compilador es reutilizable. PyPy es la primer solución práctica y eficiente propuesta para este tipo de abordaje [BR07, BT15, BCFR09]. Por su parte, las técnicas para realizar de manera práctica la evaluación parcial de un interprete [Fut99] se detallan en [WWH+17]. Truffle [WWS+12, WWW+13] es la plataforma saliente de este enfoque [WWS+12]. Cuando se usa junto con el compilador Graal [DWS+13, SWU+15], el enfoque produce código nativo optimizado a partir del código del intérprete combinado con el programa actual.

CHAPTER 3

Motivation

In this chapter we motivate our approach by illustrating the difficulties of handling adaptive scenarios using the existing language-level solutions.

3.1 Motivational Example

Let us consider an application that should not be stopped to avoid affecting customers. The application has accumulated a significant amount of data, which we would like to analyze to better understand our customers. We also need to improve its memory usage and the overall system performance. Concretely, we require the following adaptations:

REO: To protect the integrity of critical data accessed by analysis scripts, data must only be accessible for reading.

OPT: Many instances have several fields but only few are actually used. This results in a significant amount of unused memory that must be released.

PRO: To understand the performance of a critical subsystem, we need to gather information on the behavior of several classes. We want to monitor their most activated methods along with its activation arguments, return values, local variable accesses and object creations.

3.2 Current Status of Unanticipated Software Adaptation

We start by characterizing the differences between direct and indirect adaptations. Afterwards, we use REO, OPT, and PRO adaptation scenarios to pinpoint the limitations of the existing language-level approaches for handling dynamic software adaptation.

3.2.1 Direct vs. Indirect Adaptations

When distinguishing between direct and indirect adaptation we mean the following:

Definition 3.1 ***Direct adaptation*** *is the redefinition of the semantics of exactly the required operations for the needed scope.*

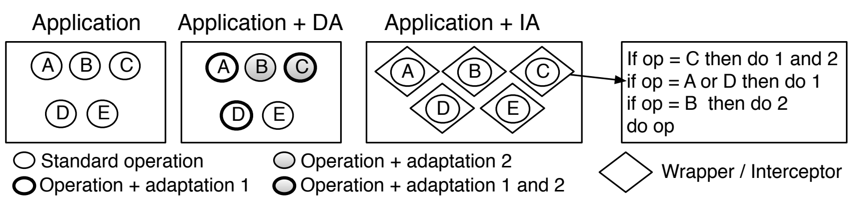

Figure 3.1: Five operations and three combination of adaptations approached with both, direct adaptations (DA) and indirect adaptations (IA)

Definition 3.2 ***Indirect adaptation*** *is the redefinition of the semantics that enables to wrap around (intercept) the required operations and redirect the flow. When the adaptation scope overpasses the requirement, it is also considered indirect.*

To clarify the difference, Figure 3.1 depicts a high-level example. Adaptations in real applications may require changes in a combination of operations for individual instances. For instance, in the context of a scenario such as REO, a direct adaptation would install a metaobject only to the references that need to be write-protected. The metaobject will redefine the semantics of the write operation. In contrast, an indirect adaptation would intercept, potentially, all the possible writes in the whole application and redirect them to an ad-hoc method. This method, depending on the receiver object, would determine whether the write operation is allowed or not.

We identified the following issues with indirect adaptations:

1. The interception of operations is usually implemented as an over-approximation of the points in the program that need an adaptation.

2. Maintainability is hard because the interception points must be updated according to the application changes.

3. Debugging gets cumbersome because intercepted methods could be polluted with instrumented code.

4. Composing adaptations may lead to complex conditions at each interception point jeopardizing performance.

5. If an operation is not interceptable, then the adaption could not be performed. For instance, language primitives might not be interceptable.

3.2.2 Direct Adaptations

To illustrate direct adaptations, we show below a direct adaptation approach for REO, OPT, and PRO:

- REO: An option would be to make all the instances immutable. However, this would imply the interruption of the whole system. To allow the system to continue working while the analysis is being executed, we need to make immutable only the references to the data from the analysis script. The problem is that dynamic languages do not usually include the concept of reference.

  Arnaud et al. proposed Handles [ADD+10] as a reification of read-only references. Handles are proxies to objects that delegate every operation to their targets but forbid mutable operations. Handles must be transparent: no one should be able to distinguish whether is accessing an object directly or through a handle. Moreover, any object accessed through a handle is wrapped into another handle. In this way the immutability is propagated through the complete chain of objects accessed from a handle. Accordingly, one option for handling REO directly is by introducing handles. To provide their semantics we need to customize the method activation, method lookup, and read operations.

- OPT: Recall from Section 2.4.4 that dynamic languages use *shapes* to provide an in memory record-like representation of a dynamic objects. Beyond optimization issues, shapes are responsible for the mapping between the field and its position in memory.

  A direct alternative to reduce the memory consumption caused by unused fields is to introduce shapes describing a much smaller amount of fields. We can then assign these shapes to sparse objects. In contrast to standard shapes, we also need to provide these shapes with a dictionary-like semantics.

- PRO: A direct way of profiling methods is to redefine how the language activates methods (only for the required instances) so that the required information is logged before the activation. We also need to redefine the reading of local variables and the creation of instance within the activated methods.

Summarizing, REO, OPT, and PRO are examples of adaptive scenarios which require behavioral adaptations, ideally, orthogonal to the application logic. For all of them we just described direct solutions that need to adapt VM-level features such as the way in which the system reads a field or organizes objects in memory. Furthermore, these adaptations are scoped at fine-grained levels, such as that of individual instances. For a detailed description of all the scenarios, as well as reference implementations using a fully reflective VM, we refer the reader to Sections 7.2.2, 7.3.2, and 7.2.3 respectively.

3.2.3 State-of-the-Art on Language-Level Adaptations

We start by analyzing the three most relevant paradigms for adapting software at the language-level: aspect-oriented programming, context-oriented programming, and metaprogramming. Afterwards, we discuss the means by which state-of-the-art runtimes support this kind of adaptation.

3.2.4 AOP

Most AOP [KLM$^+$97] implementations typically provide a domain specific language (DSL) to specify a set of points in the program (*join points*) at which a feature orthogonal to the application logic such as logging, caching, and persistence must be executed. An *aspect weaver* embeds the so-called *cross-cutting concerns* (*advices*) at the corresponding join points either statically or at run time into the program. *Pointcut* languages facilitate the specification of fine-grained locations such as before or after the execution of a method or the reading of a field. At any of these locations the execution could be redirected to ad-hoc user-defined behaviors.

Limitations. Most AOP implementations mostly provide means for indirect adaptations. For adapting a single operation for a single instance the weaver must, at least, intercept all the occurrences of such operation in the whole system where the instance may eventually be accessed. At run time, the interception code test whether the actual subject (receiver) of the operation is the required instance. This eventually leads to the problems that indirect adaptations expose (cf. Section 3.2.1).

Solving Adaptive Scenarios. Pointcuts are not about internal VM concepts like object layouts and thus, AOP cannot support the OPT scenario directly. For implementing handles, while AOP could intercept calls to the read-only reference and redirect the flow to the target, after the redirection the receiver is not anymore the handle and thus the read-only property is lost. Concretely, there is no simple way of expressing the lookup of a method call guided by one object (the target of the handle in this case), but then perform the activation in another object (the handle itself).

3.2.5 COP

COP is a paradigm specially designed for applications with behavioral variations depending on contextual information [HCN08]. In COP terms, context means any computationally accessible information. Consequently, COP provides abstractions for expressing contextual conditions. In the absence of such constructs, the application's logic would become tangled with the needed adaptations. COP could be considered as a specialized form of AOP introducing *context-aware aspects*.

Limitations. Most COP DSLs enable to express behavioral variations at the method level. As a consequence, COP implementations essentially introduce a multi-dimensional method dispatching mechanism. In contrast to most object-oriented languages, the dispatching in COP depends not only on the receiver but also on the contextual information. Adaptations concerning other aspects of execution such as statement's semantics or object's structure can not be handled directly. As such, COP is more suitable for dealing with anticipated rather than unanticipated adaptation scenarios.

Solving Adaptive Scenarios. Fundamentally, COP does not reify VM concepts other than the contextual information. Thus, it can not directly support any of the scenarios. In the case of OPT, because it does not support the customization of layouts. Regarding REO and PRO because a direct redefinition of the required operations is not possible. Furthermore, these scenarios need the support of adaptations scoped at finer-grain entities than methods.

3.2.6 Metaprogramming

Recall from definitions 2.1 and 2.3 that metaprogramming is used within reflective systems and usuallly takes the form of a metaobject protocol. Metaprogramming with MOPs provides two ways for adapting an application's behavior, both fulfilling the direct adaptation definition: 1) by modifying reified objects, *i.e.*, classes 2) by attaching metaobjects to individual objects.

Limitations. Conceptually, reflective systems are able to approach any adaptive scenario directly. However, even the approaches with the most extensive MOPs in literature suffer from a common limitation: their reifications cover only a limited subset of the VM entities [RC02, RDT08]. As a consequence, they fail to handle directly adaptations demanding changes to low-level entities.

Solving Adaptive Scenarios. Existing metaprogramming approaches can not support the OPT scenario directly because they do not reify object layouts. Moreover, for the REO scenario, advanced reflective solutions such as unanticipated partial behavioral reflection [RC02] can not express the transparency property. An alternative such as Iguana/J [RC02] does not reify the return operation nor local variable access and thus it is not able to monitor them with a direct adaptation. Section 9.3 provides a more detailed description of these reflective solutions.

3.2.7 Runtimes

The overall performance of the VM directly affects the throughput of the programs they execute. Consequently, mainstream VMs are usually built without using abstractions that could affect their performance. In addition, a common characteristic of VMs is that they are made of several intertwined components each of them coping with complex responsibilities [HGA+09]. All this makes them complex artifacts, difficult to observe and adapt at run time.

In most mainstream VMs this means that it is not possible to experiment with, for instance, an alternative algorithm for the *method lookup* mechanism or to dynamically enforce immutability for a predetermined set of objects. Such changes require developers to implement the new behavior, recompile the VM, and restart the application. Summarizing, they do not provide means for directly approaching any of the scenarios in the way we proposed.

On the other hand, some research projects provide interaction mechanisms at run time. We focus on Pinocchio [VBG+10] which is, to the best of our knowledge, the most complete solution in terms of reflective capabilities. For a more detailed analysis, including also a comprehensive list of runtimes see Section 9.2.

Pinocchio is a first-class interpreter extensible from the language level, which implements Smith's tower of interpreters [Smi84]. Since it provides reflective capabilities only for the operational semantics of the language (interpreter), Pinocchio is not able to handle adaptations for structural entities such as the layout of objects. As a consequence, Pinocchio does not support the OPT scenario.

3.2.8 Problem Statement

The existing solutions have limitations for handling each of our fine-grained unanticipated adaptations directly. For instance, solutions based on AOP and COP frequently fall into indirect mechanisms of adaptation. The reasons are two-fold. Firstly, they were not conceived for dealing with unanticipated software adaptation. AOP main goal is handling cross-cutting concerns while COP aims at facilitating the implementation of context-dependent behaviors. Secondly, despite in theory they may approach several adaptations directly, their main implementations promote mechanisms (pointcut languages, layers, etc) that biased the user to think in terms of intercepting execution points and redirecting their execution flow.

On the other hand existing runtimes and reflective systems do not reify all the elements of a language and its implementation. Especially, those regarding VM internals such as object layouts, operational semantics, etc. As a consequence, the adaptations concerning these features are not expressible or can be achieved only indirectly.

Based on these observations we argue that:

handling unanticipated adaptations at the language-level in a direct fashion requires a solution including abstractions to mold the semantics of the system considering both, the application and the runtime level.

3.3 Resumen en Castellano

En este capítulo motivamos nuestro enfoque ilustrando las dificultades que las soluciones existentes de nivel de lenguaje exponen al intentar abordar escenarios concretos de adaptación dinámica. Los escenarios son:

REO: Para protejer la integridad de datos accedidos por un análisis éstos deben ser accedidos con permisos de solo lectura.

OPT: Se require disminuir el uso de memoria causado por objetos que tienen varios campos sin inicializar resultando en una cantidad significativa de memoria desperdiciada.

PRO: Se necesita obtener información del comportamiento de varias clases mediante un monitoreo de sus métodos más usados así como también sus valores de retorno, el acceso a variables locales y la creación de nuevos objetos.

Para exponer las limitaciones del estado del arte empezamos por distinguir entre adaptaciones directas e indirectas. Las directas permiten redefinir la semántica de las operaciones exactas en el alcance preciso requerido por la adaptación. Las indirectas permiten interceptar operaciones para luego redirigir el flujo hacia código que describa semánticas alternativas. Cuando la adaptación sobrepasa el alcance exacto del requerimento también se considera una adaptación indirecta.

Las adaptaciones indirectas acarrean varios posibles problemas: 1) generalmente sobre aproximan los puntos de un programa que necesitan ser adaptados 2) la mantenibilidad puede ser compleja porque cambios en la aplicación requieren volver a instalar el código de adaptación en los puntos de intercepción 3) la depuración se puede volver engorrosa porque el código de la adaptación se instala en los métodos que definen la lógica de la aplicación 4) componer adaptaciones puede llevar a complejas condiciones de activación en los puntos de intercepción afectando así el desempeño del sistema 5) si una operación no es interceptable la adaptación no puede ser realizada.

En el capítulo demostramos que una solución para adaptar de manera directa REO, OPT, y PRO requiere la adaptación de elementos de la MV. Además, que estas adaptaciones requieren un abordaje de trazo fino. Luego de esto pasamos a describir las limitaciones que tiene el estado del arte de las soluciones de nivel de lenguaje para abordar estos escenarios de manera directa. Las tres alternativas son: la programación orientada a aspectos (AOP), la programación orientada a contexto (COP) y la metaprogramación.

En resumen las principales implementaciones de AOP [KLM$^+$97] proveen mecanismos de adaptación indirecta que llevan en general a sobre aproximar el alcance de la adaptación. AOP no permite encarar OPT de manera directa y tiene serias limitaciones para expresar ciertas propiedades inheretes a REO.

Por su parte, describimos que COP [HCN08] puede ser considerado como una especialización de AOP para los aspectos relativos al contexto. Dado que no reifica conceptos de la MV más allá de las nociones contextuales, COP no permite encarar las adaptaciones como OPT, PRO y PRO.

Finalmente la metaprogramación conceptualmente permite un abordaje directo de cualquir escenario de adaptabilidad. Sin embargo, hasta las soluciones existentes más avanzadas no cubren la reificación de muchos aspectos de bajo nivel [RC02, RDT08].

Por último describimos que, dado que tienen requirimientos de desempeño muy exigentes, las MVs de los lenguajes industriales son construidas de forma de evitar las abstracciones que puedan afectar su desempeño. Como consecuencia, estas MVs no proveen mecanismos para adaptar directamente los escenarios introducidos. Las MVs que emergen de proyectos de investigación, sí proveen algunos de estos mecanismos. Sin embargo, hasta las soluciones más avanzadas como Pinocchio [Smi84] presentan limitaciones para abordar todos los escenarios.

La conclusión a la que llegamos es que todas las soluciones del estado del arte exponen algún tipo de limitación para afrontar el tipo de adaptaciones que presentamos. Basado en esta conclusión, propugnamos que:

para afrontar escenarios de adaptación dinámica y no anticipada de manera directa se requiren soluciones que promuevan abstracciones para alterar la semántica del sistema considerando tanto los aspectos del nivel de aplicación como también los aspectos del nivel de la MV.

Part II

Defining & Building a FREE

CHAPTER 4

Fully Reflective Execution Environments

In this chapter we present our approach for overcoming the limitations of the existing approaches for dynamically adapting systems at the language-level. Since we identified that a source of this limitation is the lack of capabilities to customize the language implementation features, our approach provides those capabilities. To do so, we propose to include reflective capabilities for all components of the VM. Ideally, reflective systems would include capabilities to inspect and intercede any entity and operation at both the application and the VM levels. We named this kind of systems fully reflective virtual machines.

This chapter discusses several aspects regarding fully reflective VMs. We first define three maxims that drive their design. In addition, we propose a reference architecture covering a subset of the design space, for building this new kind of VMs. We finally present a concrete VM-level metaobject protocol, illustrate how to use it, and discuss some particular characteristics of our approach.

4.1 Main Characteristics

Maxim 1 Universal Reflective Capabilities: *VMs must provide intercession and introspection capabilities for every component and feature at both the application and the VM level.*

VMs usually impose a rigid boundary with applications. This is beneficial in terms of security, portability, and performance, but restricts the possibility of applications to affect the VM behavior at run time. To overcome this limitation, we advocate for VMs exposing reflective capabilities that cover the Cartesian product of the dimensions introduced in section 2.1.1:

$$\begin{gathered}\{Intercession, Introspection\}\\ \times \\ \{Structure, Behavior\}\\ \times \\ \{Application, VM\}\end{gathered}$$

By complementing the traditional one-way communication from VMs to applications with reflective capabilities at the VM-level, this maxim establishes the necessary condition to enable a bidirectional interaction between the application and the VM at run time.

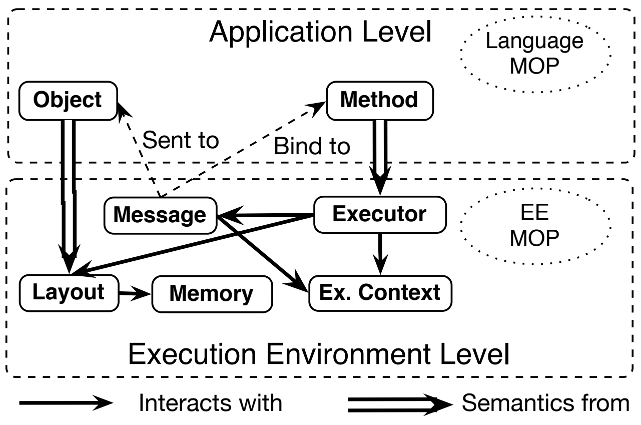

Figure 4.1: High-Level reference architecture of a FREE for an object-oriented generic

Maxim 2 Uniform Reflective Abstractions: *VMs must provide the same language tools for interacting with both the application and the VM level.*

Ideally, developers that work in different domains should be able to focus on a single tool for dealing with reflective computations at different levels. Since uniform abstractions help to improve the understandability and evolvability of the programming environment [Mey97, GR83b], we argue that VM-level reflection must use the same application-level mechanisms to avoid increasing the complexity. For instance, if the language provides reflection via a MOP, VM-level reflection must also be supported by a MOP.

Maxim 3 Separation of Application and VM: *An application should not need to be designed explicitly to support observability and adaptability. Instead, the VM should provide the necessary capabilities.*

To separate concerns, an application must focus on the problem domain, while orthogonal concerns should be handled separately. For example, similar to aspect-oriented programming, a cross-cutting low-level adaptation such as logging must not affect the application's domain model. Hence, it is important that the abstraction for dealing with reflection enables a clear separation between the application and the VM domains. However, this is rarely the case and concerns such as logging or more efficient data representations have to be realized at the application level.

4.2 Mate: a Reference Architecture

Figure 4.1 presents Mate, a high-level architecture we propose for building reflective VMs. The architecture is divided into the application and the VM layers with arrows representing different kind of interactions between components. To represent a wide range of object-oriented languages, Mate relies only on the notions of objects and methods as its core elements. This way, OO languages only need to include a MOP for application-level reflection to be compatible with MATE.

The bottom layer comprises only essential VM-level entities for executing expressions, realizing objects, and managing memory in OO languages. Further refinements of the architecture may extend those, for instance, by incorporating entities such as threads, I/O, and execution traces. To realize universal reflection (maxim 1), all of these entities must provide reflective capabilities. Besides, these capabilities are provided by a MOP, complementary to the application-level MOP. This honors the uniformity required by maxim 2 and complies with the separation of domains required by maxim 3. We briefly describe below the main responsibilities of each VM-level entity:

- `Executor` is responsible for interpreting and possibly optimizing methods. It defines the operational semantics of the language.
- `Execution context` manages the stack and the essential information that the executor uses for executing a method, including the given receiver and arguments.
- `Message` is responsible for the binding of messages to methods (method lookup) and the corresponding method activation that creates the execution *context* in which the method will be later *executed*.
- `Layouts` describe the concrete organization of the internal data of objects.
- `Memory` realizes in combination with `Layouts` the memory representation of objects. This includes defining read/write accesses as well as allocation and garbage collection.

4.3 A VM-level MOP for Handling Unanticipated Adaptations Directly

This section presents the design of a VM-level MOP for the Mate architecture. Instead of devising a design for all aspects of each component, we focus on a MOP capable of handling unanticipated dynamic adaptations such as REO, OPT, and PRO. Figure 4.2 presents a sketch of the resulting MOP. The metaclasses are grouped into two clusters: one for concepts of execution and the other for the organization of data. Compared to the Mate architecture, the resulting MOP only leaves out the memory metaclass, which we do not need for the adaptive scenarios we evaluate in Chapter 7.

The combination of the capabilities of these metaclasses represent the entire VM-level reflective capabilities of MATE. The application to new contexts may eventually bring the necessity of reifying more operations and structures. For instance, in Chapter 6 we show that new reifications are needed when using the MOP for optimizing applications. What follows is a brief description of the domain-breadth reflective capabilities per metaclass:

- `Message`: Allows developers to specialize the method lookup algorithm and the method activation mechanism. In Section 7.2.2 we show examples of its application for handling adaptation scenarios.
- `Executor`: Allows developers to redefine at the language level the behavior of *each* operation giving semantics to language constructs. For instance, in a bytecode-based implementation, it allows the redefinition of each individual bytecode. The scenario of Section 7.2.3 illustrates the usage of most of them.
- `Execution Context`: Makes it possible to observe the receiver, the arguments, the caller's context, and the stack values for each executing method. We show an example of its usage in Section 7.2.2.

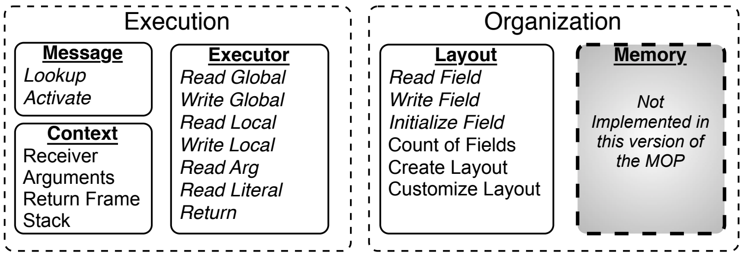

Figure 4.2: A MOP targeted to handling *unforeseen software adaptations*. The operations highlighted in italics represent behavioral, while the others structural aspects of the EE.

- `Layout`: Provides means to modify the behavior of operations interacting with object's fields. Specifically, the reading, writing, and initialization of fields. It also allows the introspection and intercession of the memory organization of objects. Usage examples can be found in Section 7.3.

4.3.1 How To Use the MOP

There are two different ways of using the MOP depending on whether one wants to interact with behavioral or structural aspects. For structural aspects, such as how many fields an object has or what the current values in the execution stack are, reflection is handled by observing or altering the corresponding metaobject directly. For instance, a layout describing the structure of each object is accessible for observation and modification. Furthermore, an execution context is accessible for every method invocation describing the contextual values. Note that although arguments and parameters may be considered behavioral, we consider them structural because their customization requires to modify the application's state.

For customizing the behavioral operations of the MOP (highlighted using italic letters in Figure 4.2) users can describe the new behaviors. To do so, the metaclasses of the MOP (`Message, Executor, and Layout`) can be subclassed to override method defining the language's behavior. Note that `ExecutionContext` is not included because its operations are accessors for structural aspects only.

Finally, the customized metaclasses must be attached to an auxiliary metaclass called `Environment`. Environments merely aggregate metaobjects into one single object, which minimizes memory usage, because each entity only needs a single pointer to an environment to customize its whole VM-level semantics.

Example To illustrate how to specialize these metaclasses consider a simplification of the REO scenario from Section 6.1. Let us assume that the analyzed module is not used in the meantime. Thereby, we can avoid introducing readonly references and implement object immutability. To do that we only need to customize the *Write field* operation from `Executor` to make it throw an exception whenever the system tries to change the value of a field.

Figure 4.3.1:a shows a configuration of metaobjects implementing the scenario. For readability, the figure distinguishes between `metaclasses` and *metaobjects*. `Immutable`, which extends `Executor`, implements the read-only semantics by specializing the *writeField* method. To make it usable an `environment` instance must be created (*aReadonlySemantics*) and the customization must be attached to its Executor field. *aReadonlySemantics* now can be attached to standard objects to make

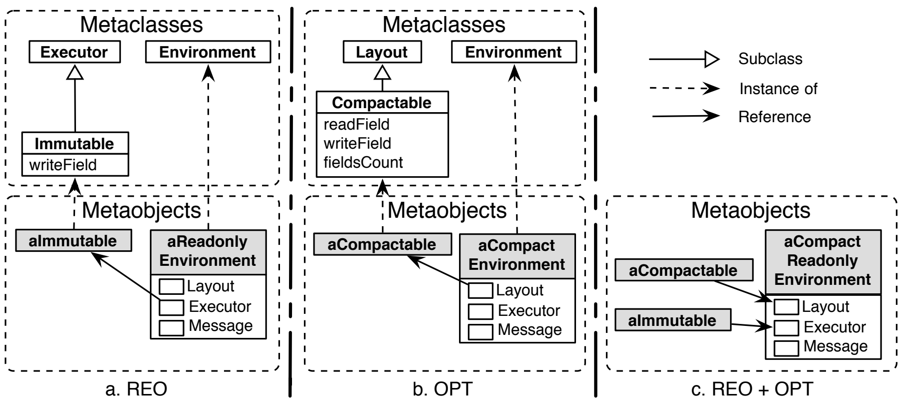

Figure 4.3: Configuration of metaobjects for the adaptation scenario. The left-hand side presents the configuration for a simplfication of the REO scenario. The middle part describes the configuration for the OPT scenario. The right-hand side shows how to combine the two behaviors in one *environment* metaobject.

them immutable. Below the corresponding concrete steps needed to generate this configuration:

1. Subclass `Executor` with `Immutable` and specialize the write field operation.
2. Instantiate `Immutable`. We named the instanciated metaobject *aImmutable*.
3. Instantiate `Environment`. We named the new instance *aReadonlySemantics*.
4. Assign *aImmutable* to the execution field of *aReadonlySemantics*.

Figure 4.3.1:b shows a possible configuration for supporting the OPT scenario. The goal is to reduce memory consumption by compressing objects contain- ing uninitialized fields. This is done by using a customized layout that implements a dictionary-like object layout, using only space for fields that are used. To this end, `Compactable` customizes the semantics of *Read field*, *write field*, and *field count* from `Layout`. Analogous to the immutable case, an instance of `Compactable` (*aCompactable*) is attached to an instance of `Environment`(*aCompactEnvironment*) but this time to the Layout field. Objects that want to use a sparse layout must then be linked to (*aCompactEnvironment*).

Finally, Figure 4.3.1:c just shows how to combine customizations from different metaclasses in a single *Environment* (*aCompactReadonlyEnvironment*).

4.3.2 Characteristics of MATE's MOP

We discuss below the impact of some principled decisions we took in the process of designing the MOP.

Composability / Modularity

MOPs adhering to the *ontological correspondence* principle isolate the reflective capabilities into separate objects that correspond to domain-level structures. This promotes composability [McA95]. We honor that principle representing each VM-level entity by a separate metaclass.

Scoping

The `Environment` metaclass links base-level entities with the VM metalevel. For fine-grained scoping, every base-level object has a link to an`environment` metaobject which describes how the VM must operate on itself. In addition, method activations can also link to an *environment* to redefine the semantics of all the operations executed within the method. Finally, they can be set globally too. To avoid ambiguity we determine that the *environment* applied to the most general context precedes the others.

Activation/deactivation

The explicit separation between the VM-level and application-level MOPs comply with the Mirror's principle of *stratification*. Bracha and Ungar claim that adhering to this principle helps to avoid overheads when VM-level reflection is not needed [BU04]. In our case, for activating/deactivating VM-level behavior redefinitions, it is only needed to add/remove the *environment* metaobject from the corresponding base-level entity.

VM-level Behavioral Reflection

To realize the causal connection for behavioral reflection, MATE uses *intercession handling* (IH) [TNCC03]. Recall from Definition 2.10 that before executing any VM-level operation included in the MOP such as invoking a method, or accessing a local variable, the VM tests whether there is a metaobject redefining it. If not, the standard VM-level operation executes. In case the operation is redefined, the VM delegates the responsibility to the corresponding language-level method. The following algorithm defines the concrete IH for our MOP:

```
def IH(frame, operation){
  result = NOMETAOBJECT;
  if (level is Meta) return result;
  metaobject = getReceiver().getMetaobject();
  if (metaobject != null){
    result = metaobject.activateFor(operation);
  }
  if (metaobject == NOMETAOBJECT or result == null) {
    metaobject = frame.getMetaobject();
    if (metaobject != NOMETAOBJECT)
      result = metaobject.activateFor(operation);
    }
  }
  if (metaobject == NOMETAOBJECT or result == null) {
    metaobject = getGlobalMetaobject(operation);
    if (metaobject != NOMETAOBJECT)
      result = metaobject.activateFor(operation);
    }
  }
  return result;
}
```

Listing 4.1: Algorithm describing the intercession handling process.

The first two lines of the algorithm show how the metaregression (see Section 2.1.3) problem is solved. A general solution to that problem is making reflective architectures *context-aware* by reifying the execution level [DSD08b]. This provides an extra parameter for defining the semantics of the operations. We use a simplification of *meta-contexts* by providing just two different levels of execution: *meta* and *base*. Every time the VM delegates the execution to the metalevel, the depth-level is set to *meta* and no further delegations are possible until the operation returns. This plays the role of a recursion bound.

Starting with line 4, the IH checks for the existence of a metaobject associated with the *subject* of the current VM operation (*e.g.*, in a variable read operation, the concrete object owning the variable). In case there is one, the IH activates the language-level redefinition of the current operation in the metaobject. Otherwise, or if the metaobject does not redefine the VM operation being executed (returns null), the IH looks for a metaobject associated with the current *frame* of execution. Again, if there is no metaobject or it does not redefined the operation, the IH checks for a global scoping metaobject redefining the current operation. If the operation is not redefined at any level, the IH returns the `NOMETAOBJECT` constant and the VM executes the default behavior.

As already explained the domain-depth completeness of the behavioral part of our MOP is limited to two levels: meta and base. The main reason for this solution is that none of the considered scenarios requires a higher domain-depth. The domain-breadth completeness was already discussed for each metaclass of the MOP in section 4.3. Finally, note that our approach does not support adding new intercession handling points at run time, *i.e.*, the behavioral reflection capabilities of the VM cannot be increased on-the-fly. This means the MOP is fixed at compile time.

VM-level Structural Reflection

We reify the structure of base-level entities using the state (fields) of metaobjects. To guarantee the causal connection, the base level entity behaves based on the information in the corresponding metaobject and vice versa. Consequently, this mechanism enables a program to observe and change the value of metaobject fields with instantaneous effects on the base-level entity.

Analogous to the behavioral case structural reflective capabilities of the MOP cannot be increased at run time. Concretely, no new reification of structural VM level entities (for instace the trace of method activations) can be realized after compiling the FREE. Considering the domain-depth dimension, we faced a metaregression issue, analogous to the one discussed for IHs but with layouts. Since layout metaobjects are also first-class objects their layout is defined by another layout metaobject. Since our adaptation scenarios do not require further capabilities, we provide non-redefinable layouts for metaobjects limiting the domain-depth to two levels.

4.4 Resumen en Castellano

En este capítulo presentamos y discutimos nuestro enfoque para solucionar las limitaciones existentes para adaptar software dinámicamente con soluciones de nivel de lenguaje. Para esto proponemos incluir capacidades reflexivas sobre todos los componentes de las MVs. Nombramos a los sistemas que cumplen con estos requisitos máquinas virtuales reflexivas.

Sugerimos que las MV reflexivas deben satisfacer tres características principales: 1) prover capacidades reflexivas para todos los componentes tanto a nivel de lenguaje como a nivel de MV 2) proveer los mismos mecanismos para interactuar reflexivamente con las entidades de ambos niveles 3) permitir expresar aspectos adaptativos relativos a elementos de la MV de manera ortogonal a como se expresa la lógica estándar de una aplicación.

Luego proponemos una arquitectura para construir MV reflexivas que cumple con las características recientemente mencionadas. Presentamos esta arquitectura en la figura 4.1 y la llamamos MATE. Para ser compatible con un rango abarcativo de lenguajes orientadas a objetos, MATE depende solo de las nociones de objeto y método.

Respecto a la MV, MATE modela un conjunto de entidades necesarias para implementar objetos, manejar la memoria y ejecutar sentencias. Concretamente, estas entidades son: el *ejecutor*, el contexto de ejecución, la noción de mensaje, los layout de los objetos y la memoria. Todas estas entidades deben proveer capacidades reflexivas.

Además presentamos el diseño de un protocolo de metaobject (MOP) para la MATE que cubre solo los aspectos necesarios para abordar las adaptaciones REO, OPT y PRO. La Figura 4.2 presenta el MOP resultante. Comparado con la arquitectura, al MOP solo le falta la reificación de la memoria dado que no fue necesaria para los escenarios adaptativos. En el capítulo 6 presentamos algunas extensiones al MOP para poder encarar escenarios de optimización.

Existen dos maneras de usar el MOP de nivel de MV dependiendo si se predican sobre operaciones que hacen referencia a la estructura o al comportamiento de los programas. Para el caso de las operaciones estructurales, las capacidades reflexivas se proveen mediante la lectura o modificación de los campos de determinados metaobjetos. Por otra parte, para customizar el comportamiento de la MV los desarrolladores deben poder expresar los nuevos comportamientos. Para esto, el MOP requiere que se extiendan determinadas metaclases mediante los mecanismos clásicos de subclasificación y sobrecarga de métodos.

El capítulo también provee un ejemplo de cómo usar el MOP para resolver los escenarios REO y OPT. Este ejemplo se describe en la Figura 4.3.1 donde además se introduce el mecanismo de activación de las capacidades reflexivas mediante los *environments*. Los environments son metaobjetos especiales cuya única función es enlazar el nivel base de una aplicación con los metaobjetos de la MV.

Se continúa con una discusión sobre algunas características salientes que se desprenden del MOP que finalmente provee las capaciades reflexivas de MATE. Por ejemplo, se muestra que al cumplir con ciertas propiedades el MOP promueve la composicionalidad y la modularidad de los mecanismos adaptativos. También se ilustran los diversos modos de expresar el alcance de los metaobjectos: objetos invidual es, métodos enteros, o todo el sistema. La activación y desactivación de los mecanismos adaptativos se logra mediante la desinstalación del metaobjeto de los sujetos asociados. Esto simplemente requiere la escritura de un campo.

Finalmente, el capítulo presenta la forma de implementar los mecanismos reflexivos comportamentales en la MV. Se requiere de la instalación de un *manejador de intercesiones* (IH) [TNCC03] en todas las operaciones de la MV que puedan ser redefinidas por el MOP. En el listado 4.1 se describe el IH concreto de MATE y el mecanismo particular mediante el cual se resuelva el problema de la metaregresión.

CHAPTER 5

Building an Efficient Reflective VM

In this chapter we present MATE$_{PE}$ and MATE$_{MT}$, two prototypes of reflective VMs with equivalent reflective capabilities: both implement the necessary features to support the MOP presented in the previous chapter. MATE$_{PE}$ and MATE$_{MT}$ are artifacts with advanced reflective characteristics, not found in other VMs. They allow us to perform a comprehensive validation to test the hypotheses proposed in this thesis.

The main difference between MATE$_{PE}$ and MATE$_{MT}$ is that the first uses the partial evaluation meta-compilation approach while the latter uses meta-tracing. This allows us to compare the impact on performance of both meta-compilation approaches in the context of reflective VMs. Also, by comparing the performance results, we can analyze if eventual overheads are fundamental or because of the limitations of any of the implementations. To simplify the reading, from now on we will use the term MATE whenever the distinction is not relevant for that context.

Concerning language capabilities, MATE implements most of the features of the Smalltalk programming language. We chose Smalltalk as target because it already features advanced reflective capabilities at the language level that eases the implementation of the VM-level reflective API. Nevertheless, our contributions focus mainly on supporting reflection at the VM level and are applicable, in principle, to any (dynamic) language.

It is worth noting that MATE$_{PE}$ and MATE$_{MT}$ implement a MOP that does not include reflective capabilities for memory operations. As such, we refer to these implementations as reflective but not fully reflective VMs. Also note that building industrial-strength VMs is a highly resource demanding task and hence not our aim. In contrast, our goal is to provide a representative platform for understanding the effects of fully reflective VMs. This is the main reason why MATE$_{PE}$ and MATE$_{MT}$ do not provide, for instance, UI support.

This chapter also discuss performance aspects of reflective VMs. The reason is that we consider that one of the greatest obstacles hindering a wider adoption of reflective VMs would be the perception that they are inefficient. We already suggested in Section 2.1.3 that the flexibility provided by making each feature reflective comes with performance penalties. Concretely, it implies a significant proliferation of indirections and conditions (guards) that the VM needs to check at run time to realize the proper semantics of that functionality. As a consequence, it results sensible to perceive that a reflective VM would be, by definition, impractical.

To refute this belief, we present an optimization model which principal goal is to minimize the indirections caused by the incorporation of VM reflective capabilities. It is inspired by a recent work showing that speculative compilers nearly eliminate the run-time overhead for common usage patterns of reflection [MSD15]. The optimizations are also based on the assumption that the variability of actually observed metaobjects would be low for consecutive executions of the same application's code. In chapter 7 we validate this model.

5.1 SOM$_{PE}$ and SOM$_{MT}$

The Simple Object Machine (SOM) [HHP^{+}10] is a Smalltalk implementation designed to avoid inessential complexity. It includes fundamental language concepts such as objects, classes, closures, and non-local returns. Following the Smalltalk tradition, control structures such as `if` or `while` are defined as polymorphic methods on objects and rely on closures and non-local returns for realizing their behavior.

TruffleSOM (SOM$_{PE}$) is a SOM implementation on top of Truffle while RTruffleSOM (SOM$_{MT}$) is a SOM implementation using PyPy [MSD15, MD15]. They are both AST interpreters. Exploiting self-optimization techniques they both specialize generic method invocations and control structures by representing them speculatively as concrete AST node for their most common types instead of polymorphic methods. They also use storage strategies [BDT13] for arrays and vectors to achieve optimal performance for collections with homogeneously typed elements.

Since the original goal was to compare the performance results of the different meta compilation techniques, they were originally designed to be as similar as possible [MD15]. Language functionalities such as method invocation, field access, or iteration constructs are represented in the same way. Accordingly, for almost all language constructs the structure of the resulting AST is the same for both interpreters. However, it is worth mentioning that SOM$_{PE}$ extensively uses the TruffleDSL to realize the different specializations while SOM$_{MT}$ is built with ad hoc techniques.

Finally both SOM implementations heavily uses dispatch chains to implement its optimizations. Figure 5.1 shows an example: on the left-hand side is the unoptimized AST of an expression that accesses object field `foo` and invokes the method `bar` with a constant integer parameter. The right-hand side shows the optimized AST. The optimized state is reached after executing the expression at least once. The *Literal Node* is rewritten to an *Integer Literal Node* and the *Message Node* is identified as a generic message. The bottom compound nodes are an example of a dispatch chain for optimizing a field read. The first node of the list already cached the value of the only observed layout and the location where the corresponding field is located for that layout. In further executions of the field read, the VM first checks if the layout of the receiver is the same as the one cached. In case of a hit, the location for the field does not have to be computed. In case of a miss, the chain executes the following node. The last node of the chain (in this case *Uninit(ialized) Read* node) must always decide if it adds a new cached entry to the chain or if it executes the generic, and thus more expensive in terms of performance, read operation.

5.1.1 From SOM to MATE

MATE$_{PE}$ (resp. MATE$_{MT}$) extends SOM$_{PE}$ (resp. SOM$_{MT}$) with two goals: 1) turn it into a reflective VM and 2) become a full-fledged Smalltalk VM. For the first, both MATE VMs implements the VM-level MOP presented in section 4.3. For the latter we add support for literal arrays, cascade message sends, file ac-

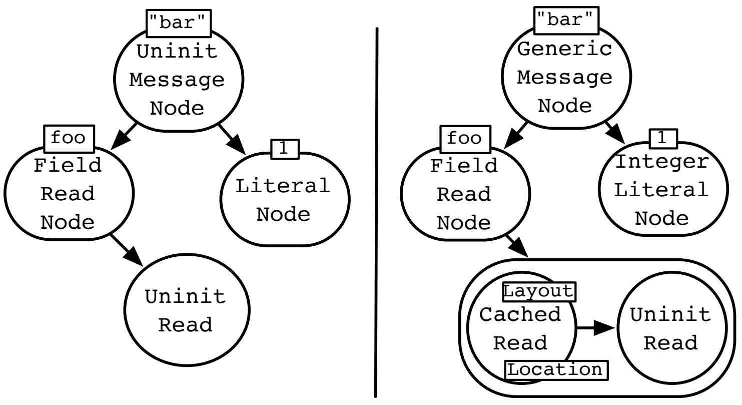

Figure 5.1: Parser output (left) and Optimized (right) ASTs for an expression `foo.bar(1)` in TruffleSOM.

cess, exceptions, and Smalltalk's streams [GR83a]. As a result, MATE is able to run programs developed for two compatible open-source Smalltalk implementations, Squeak [IKM^{+}97] and Pharo [BDN^{+}09], provided they do not make use of GUI or concurrency.

Below we provide a brief overview of the most significant differences between MATE and SOM implementations:

Environments. In MATE$_{PE}$ and MATE$_{MT}$ the semantics of each behavioral VM operation can be redefined at three different scoping levels (cf. Section 4.3.2): 1) individual objects 2) the execution of a method 3) globally. To support object scoping, we add a field to every object referring to an *environment* (See Section 4.3.1). Below we discuss its impact in terms of memory consumption. To support method scoping, we modified the calling convention so that every method receives an extra parameter with an implicit *environment*. This *environment* governs the semantics of all the operations executed within this method no matter the subjects. For the global scoping we introduced a global variable also referring to an *environment*.

Intercession Handling. VM behavioral operations included in the MOP (distinguished with italics in Figure 4.2) execute the intercession handling introduced in Section 4.3.2. Recall that the intercession handling will check for the existence of a metaobject that applies to any of the possible scopes and delegate the execution to the corresponding application-level method if there exist a redefinition. Otherwise, it executes the standard VM operation.

Objects To enable custom object representations using the MOP we expose object layouts to the language via primitive operations. This enables to reflectively inspect, create, and customize layouts of individual objects at run time. In addition, MATE$_{PE}$ and MATE$_{MT}$ incorporates support for basic Smalltalk types such as characters and bytes.

Execution Stack In Smalltalk, the *execution context* (frame) of the current method is reflectively accessible via the `thisContext` keyword. Concretely, Smalltalk natively implements the `Context` metaclass of the MOP. However, SOM does not support this behavior. We incorporate it to MATE.

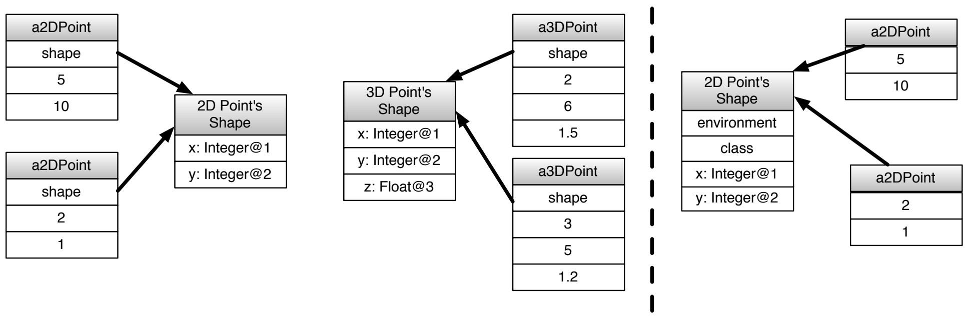

Figure 5.2: In the left, an example for the representation of objects in the Truffle Object Storage Model. In the right, the representation of objects in our MATE implementations

Files and Streams Contrary to their corresponding SOM versions, MATE_{PE} and MATE_{MT} incorporate support for I/O from the application-level via file handling. To do so, we ported the application-level file management from Pharo and implemented the required primitives at the VM level. We also ported from Pharo its Stream class hierarchy.

5.1.2 Object Model

Truffle includes an object storage model (OSM) [WWB+14] that provides an efficient representation for objects. Similar to Self's maps (cf. Section 2.4.4), Truffle's OSM collects the object structure and field types at run time to enable a record-like representation in memory. To support these features the OSM maintains a so-called object *shape*. Shapes are themselves immutable objects representing a fixed set of fields and the types that have been observed for an object type.

Figure 5.2 (left-side) illustrates Truffle's OSM. *a2DPoint1* and *a2DPoint2* are two-dimensional points while, *a3DPoint3* and *a3DPoint4* are three-dimensional points. Their memory organization consists of a pointer to a shape and a cell storing the value of each of their corresponding fields. *2Dshape* and *3DShape* describe the memory organization for the two kinds of points respectively. Analogous to the optimizations described for operations, the OSM speculates on the observed types for each shape. Furthermore, as the example shows, based on the observed type information primitive types can be stored without boxing their values in objects, thus improving performance further.

SOM_{PE} and SOM_{MT} originally provide its own object model. The basic principles behind these implementations are very similar to those of the Truffle OSM. Concerning the original object models we introduce two changes in our MATE versions. The first one is that for MATE_{PE} we decided to provide a new object model based on the Truffle OSM. Accordingly, MATE_{PE} actually expose the aforementioned shape entities to the language-level for implementing the Layout's metaobject capabilities.

The second change is that for both SOM implementations the original object model stores the class of each object as a field of each instance. In MATE, we decided to move the class reference of each instance to its shape. This way we save a pointer for every instance. Furthermore, in the shape we also decided to store the environment assuming that a semantics change will be applied to several instances of the same classes. Accordingly, we force a correlation between classes and environments. Figure 5.2, in the right-side, shows how the two dimensional points are actually modeled in MATE. We discuss below how this allows us to speculate on object layouts for further optimizing the performance of our reflective VMs.

5.2 Conjectures about the Dynamic Usage of the MOP

MOPs providing low-level reflective capabilities were historically considered slow even by the research literature [Asa14]. This is part of the reason why aspect oriented programming and partial behavioral reflection frameworks were conceived: to reduce the cost of reflection to a minimal part of the program where the semantic changes are needed. However, in stark contrast to these (and other reflective) approaches that limit their reflective capabilities to reduce the performance overheads, fully reflective VM propose a MOP with comprehensive reflective capabilities. Accordingly, to be seriously considered by the software community, reflective VMs must approach an optimization model to be competitive in terms of performance with mainstream VMs.

To optimize its dynamic features, reflective systems speculate on common execution patterns. Since reflective VMs are a recent approach for building flexible systems there are no large applications from which common usage patterns could be derived. Consequently, we propose an optimization model based on two key assumptions about the behaviour of applications that would benefit from a reflective VM's flexibility. Our conjectures can be detailed as follows:

Stable phase semantics: We expect users of the MOP to freely define and combine metaobjects describing adaptations to the VM at run time. However, we expect those adaptations to rarely change. As a consequence, eventually they would show a stable behavior. This means, essentially, that we assume that users are not going to constantly create new metaobjects with different behavior within the same application.

Low local meta-variability: We also conjecture that despite a single intercession handling site can potentially observe many different metaobjects, most will expose a bevavior similar to that observed in dynamic languages' call sites [DS84, HCU91, MSD15]. Concretely, we expect the applications to locally use a minimal degree of the potential dynamicity provided by the MOP. This means that the large majority of intercession handling sites will be monomorphic, *i.e.*, they observe only a single metaobject. Few will be polymorphic with multiple metaobjects observed. And in very rare cases, many observed metaobjects will make an intercession handling site megamorphic. We further assume that this behavior correlates strongly with method polymorphism. A megamorphic call site is likely to have a higher probability to observe more metaobjects than a monomorphic call site that is used only in very specific cases.

Finally, note that the results exposing that the usage of dynamic features in real web applications written in JavaScript is not negligible [RLBV10] do not interfere with our assumptions. Concretely, they found that only 2,5% of the call sites of these applications are megamorphic (more than 5 different method targets). Furthermore, in 80% of the applications this value was actually below 1%. Monomorphic sites were above 80%, and above 90% in more than half of the applications.

Overall, this means that for a wide variety of use cases, the local meta-variability will be minimal and most intercession handling sites will be monomorphic.

5.3 Optimizing Intercession Handling Sites

This section describes an optimization model for intercession handling sites (see definition 4.2) based on the aforementioned conjectures: stable phase semantics and low local meta-variability.

Speculate on metaobjects: We propose to cache, at each intercession handling site, a predefined number of the metaobjects that have been observed at run time locally. Furthermore, for each metaobject we propose to cache the entry point of the (language-level) method reimplementing the corresponding VM operation. This way, at run time, we only need to test whether the current metaobject is in the list of the cached ones. In case of a hit, we can directly call the corresponding cached method saving all the indirections and execution cycles that the lookup of the method in the metaobject consumes. A step further, an optimizing compiler would be able to even inline the target. On the other hand, in case of a miss, we have to execute all the checks and afterwards the method lookup and activation.

Note that misses will be infrequent when both of our assumptions hold. Although, a deoptimization will be triggered by any new metaobject introduced or by the modification of a metaobject already cached, under our assumptions, the system will eventually stabilize again. We further propose to store two different caches at each intercession handling site: one for the metaobjects observed in individual objects (metaobjects attached to the receivers of the operations) and one for the metaobjects observed at the method level (metaobjects assigned to the observed activation frames).

Combine metaobjects with object layouts: Most intercession handling sites are triggered by VM operations that may need to access the state of objects. For instance, when operations directly read/write fields or require object's meta information, as in the case of the receiver in the context of a method lookup. Hence, we designed an object model in which the metaobjects are coupled with the object's layout. This allows us to store object layouts in the dispatch chains along with the entry point of the redefined method for the corresponding metaobject. At run time, the dispatch chain guard only needs to compare the layout of the current subject with the layouts stored in the dispatch chain using pointer equality. Since most VM operations already need to access the layout our hypothesis is that this check do not introduce significant overheads.

We illustrate our strategy by showing in Figure 5.3 two alternatives for storing the metaobject on subjects. We reuse the two dimensional point classes introduced in Figure 5.2. Hence, we consider an object layout for the *Point class*, describing that its instances are composed by two fields: x in the first position, and y in the second. The grey background of the layout fields remarks that they are immutable. This is a common approach for efficiently managing object allocations in dynamic languages based the on speculation techniques supported by the TruffleOSM [WWB$^+$14].

The two alternatives are presented in the left and right sides of figure 5.3 respectively. The most straightforward alternative (A, left) is storing the metaobject just as another field. This means that the layout describes a predefined position in memory for the metaobject for all instances. As a consequence, whenever a metaobject needs to be accessed or tested by a guard, a load of the corresponding memory position is mandatory. On the other hand, the metaobject in alternative B (right) is another immutable field of the layout. This mean that the metaobject can be accessed without accessing the object's particular memory.

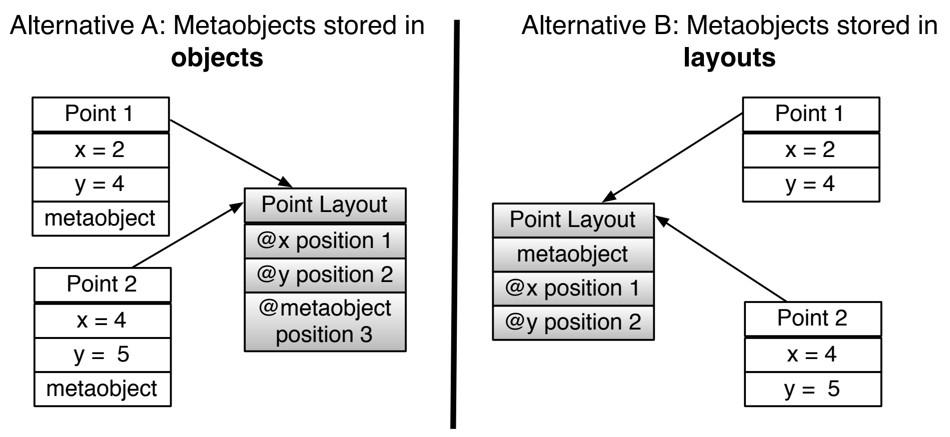

Figure 5.3: In the left of the figure, alternative A shows a layout configuration describing that each instance stores the metaobject as one of its fields. Instead, in alternative B (right-hand side) each layout stores a metaobject itself. Consequently, the metaobject is shared by all the instances pointing to this layout.

A drawback of this strategy is that objects with the same structure but with different metaobjects will have distinct layouts. As a consequence, the size of any dispatch chain guarding object layouts may increase, and thus incur in additional overheads. Although each object does not allocate extra memory for the metaobject, the overall memory consumption may increase because we need several object layouts for each class. However, under our assumptions on phase stability and under low meta variability, both phenomenon should be very infrequent.

We propose alternative B, which combines metaobjects with object layouts, as part of our optimization model.

Overall, our hypothesis is that an aggressive dynamic compiler in combination with both proposed optimizations will mitigate the cost of the indirections caused by the ubiquitous intercession handling sites of reflective VMs. We validate this hypothesis in Chapter 8.

5.4 Implementing Mate Efficiently

We describe in this section the implementation of the optimization model in MATE$_{PE}$ and MATE$_{MT}$.

5.4.1 Meta-level Nodes

We refer to nodes that are part of SOM, *e.g.*, the nodes presented in Figure 5.1, as the *base-level nodes*. Instead, in MATE$_{PE}$ and MATE$_{MT}$, the intercession handling semantics are implemented with new AST nodes. We refer to nodes related to MATE as *meta-level nodes*. To implement the intercession handling, all the base-level nodes of operations that can be customized by the MOP are wrapped with meta-level nodes. If the intercession handling does not find a *environment* redefining the operation, then the meta-level node delegates the execution to the base-level node.

Figure 5.4 shows the AST representation of the expression *foo bar: 1*. Recall that the base-level AST nodes for the same expression were already depicted on Figure 5.1. The only difference is that now we use Smalltalk jargon. For a matter

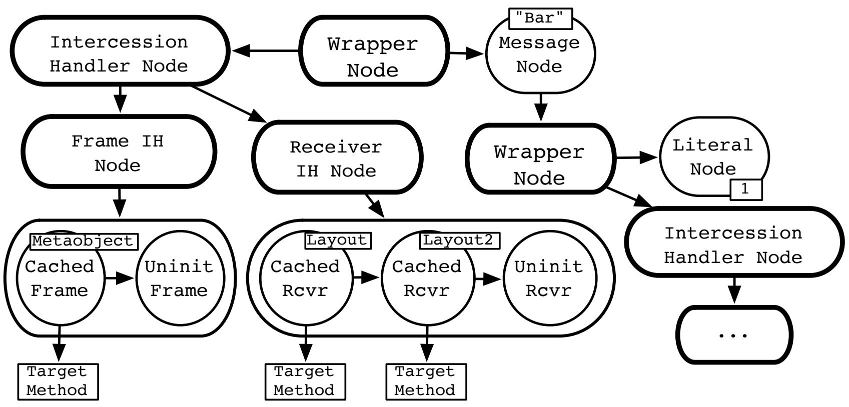

Figure 5.4: Mate's AST version of foo.bar(1) statement without including the field read node but with dispatch chains already filled. The bold circle nodes are metalevel nodes exclusive for Mate. Simple circled nodes were inherited from SOM.

of simplicity, we only show the complete wrapping of the message send operation (node). The intercession handling is composed of several meta-level nodes represented with bold thick nodes. Each node implements a fragment of the intercession handling algorithm presented in Listing 4.1 (cf. Section 6.1), or is part of a dispatch chain (compound nodes) implementing the optimizations described in Section 5.3. We introduce the meta-level nodes below and leave the description of the dispatch chains for the following section.

Wrapper Node: the entry point of every VM operation that can be redefined. It wraps a base-level node and connects it with its corresponding meta-level nodes implementing the intercession handling. Its role is mainly to arbitrate the intercession handling process. First it delegates the execution to the intercession handler. If the handler produces a result (the operation was redefined by a metaobject) it just returns it. Otherwise, it delegates the execution to the base-level node and returns its corresponding return value.

Intercession Handler Node: orchestrates the intercession handling relying on the Receiver and Frame IH nodes.

Receiver and Frame IH Node: checks whether the environment/receiver has a metaobject redefining the current operation. In that case it delegates the invocation of the corresponding language-level method to the corresponding intercession handling's site dispatch chain.

5.4.2 Optimizations

Below we explain how we implement the optimization model for intercession handling sites in general, and the dispatch chains for subjects and frames, in particular.

Speculation on the Metaobject of Subjects. As explained in Section 5.2, we expect a low variability of metaobjects at each specific intercession handling site. Thus, we cache an association between the observed metaobjects and its language-level method redefining the corresponding operation within a dispatch chain. Ideally, this optimization enables to avoid the lookup for the redefinition of an operation in a metaobject whenever the metaobject was already observed. In the dispatch chain hanging from the *Receiver IH Node* (Figure 5.4), each *Cached Rcvr Node* caches a *Layout* and the target method reimplementing the VM operation. Recall that in our optimization model each layout is in a one-to-one relation with a metaobject. In case the assumption of low meta-variability fails, the dispatch chain falls back to a generic implementation that do not use any kind of caching.

Speculation on the Metaobject of Frames. Analogously to the previous case, we implemented a similar dispatch chain speculating on the metaobjects attached to frames. This dispatch chain hangs from the *Frame IH Node* of Figure 5.4. In contrast to the previous case, this dispatch chain caches the actual metaobject because frames do not have a layout.

Optimization of Metaobject Guards. As already explained, we combine object layouts with metaobjects as a mean to reduce an extra indirection in each intercession handling (cf. Section 5.3). Concretely, every object layout is extended with an extra field denoting its metaobject. Then, we can determine any change in the metaobject assigned to an object just by testing two layouts. Accordingly, the dispatch chains that cache metaobjects store not metaobjects but layouts. At run time, whenever each *Receiver IH* node walks through the *Cached Rcvr* nodes of the dispatch chain, it first compares the current subject's layout with the stored layout to determine whether it can execute the corresponding optimized node. In case of a hit, it saves the extra indirection for reading the actual metaobject assigned to the current subject.

Speculation on IH branches. Due to low variability, it is likely that each intercession handling executes only one of its branches (*i.e.*, receiver or frame). However, the compiler is not always able to figure that out and optimize accordingly. We thus profile branches related to the usage of metaobjects to help the compiler to speculate on which branches to compile and optimize.

5.5 Resumen en Castellano

En este capítulo presentamos MATE$_{PE}$ y MATE$_{MT}$, dos prototipos de MVs reflexivas. Ambas implementan las funcionalidades necesarias para soportar el MOP presentado en el capítulo previo. MATE$_{PE}$ y MATE$_{MT}$ son artefactos de software con capacidades reflexivas no encontradas en otras MVs y por ende nos permiten una validación extensiva de las hipótesis presentadas en esta tesis. La principal diferencia entre ambas MVs es que la primera usa un mecanismo de meta compilación de evaluación parcial mientras que la segunda de meta trazas.

El capítulo además introduce varias consideraciones respecto a aspectos de desempeño de las MVs reflexivas. El motivo es que, al contrario de lo que sucede con la mayoría de los sistemas reflexivos existentes, las MV reflexivas no limitan las capacidades reflexivas para mitigar penalidades en su desempeño. Por lo tanto, presentamos un modelo de optimización desarrollado especialmente para MV reflexivas con el objetivo de reducir signifcativamente las penalidades que estos sistemas conceptualmente introducen.

Antes de ésto, presentamos (SOM) [HHP+10], un lenguaje similar a Smalltalk pero diseñado para evitar todas las complejidades no esenciales de éste. TruffleSOM (SOM$_{PE}$) es una implemetación de SOM sobre el *framework* de Truffle y RTruffleSOM (SOM$_{MT}$) una que utiliza PyPy [MSD15, MD15]. Ambos son intérpretes de árboles abstractos de sintaxis.

MATE$_{PE}$ extiende SOM$_{PE}$ (MATE$_{MT}$ y SOM$_{MT}$ respectivamente) persiguiendo dos objetivos: 1) convertirlo en una MV reflexiva y 2) convertirlo en una MV de Smalltalk. Para el primer objetivo ambas MVs implementan el MOP presentado en la sección 4.3. Para lo segundo agregamos soporte para arreglos de literales, mensajes en cascada, accesso a archivos, excepciones, y streams [GR83a]. Como resultado, nuestras implementaciones son compatibles con los programas escritos en Squeak [IKM+97] y Pharo [BDN+09] Smalltalk, siempre y cuando no requieran el uso de la interface gráfica ni mecanismos de concurrencia.

Luego el capítulo presenta algunas conjeturas que se esperan sobre el uso del MOP de nivel de la MV. En base a estas conjeturas se presenta el diseño de un modelo de optimización. La primer conjetura expresa que se espera un uso libre del MOP pero las adaptaciones que se introducen presentarán cierta estabilidad. Esencialmente, la conjetura expresa que no es esperable que los usuarios instalen constantemente metaobjetos diferentes en la misma aplicación.

La segunda conjetura, relacionada con la primera, indica que se espera una baja meta-variabilidad. Esto quiere decir que se espera el uso de diversos metaobjetos en la aplicación, pero el uso local (por cada manejador de intercesión) debería ser moderado. Concretamente, esperamos que el comportamiento sea similar al que exhiben la mayoría de los programas que corren sobre lenguajes dinámicos donde el dinamismo potencial no es frecuentemente utilizado [DS84, HCU91, MSD15].

El capítulo continúa presentando el modelo de optimización que proponemos en base a estas dos conjeturas. Básicamente, proponemos que se especule sobre los metaobjetos observados en tiempo de ejecución en cada sitio donde se instale un manejador de intercesión. Esto permite almacenar de antemano un conjunto de información que causa penalidades de desempeño significativas si debe ser obtenida en tiempo de ejecución. Por supuesto, en el caso de que las especulaciones no se cumplan se sufrirán esas penalidades. En caso de que las presunciones de comportamiento en general se cumplan, esto será poco frecuente.

También proponemos alocar los metaobjetos en el layout de los objetos. Conjeturamos que esto permitirá reducir aun más las indirecciones que deben ejecutarse en tiempo de ejecución ya que, en general, siempre es necesario acceder al layout de los objetos para casi todas las operaciones de la MV. La Figura 5.3 describe nuestra estrategia al respecto.

Nuestra hipótesis general es, entonces, que un compilador dinámico agresivo en combinación con las optimizaciones propuestas será capaz de mitigar el costo de las indirecciones introducidas por el hecho de tener mecanismos de manejador de intercesión ubicuos. Esta hipótesis es validada empíricamente en el capítulo 8.

Finalmente, el corriente capítulo describe ciertos aspectos implementativos del modelo de optimización tanto en MATE$_{PE}$ como MATE$_{MT}$. La Figura 5.1 resume los nodos más relevantes que se introducen an ambas implementaciones. Además se detallan los aspectos concretos de la implementación de los mecanismos especulativos.

CHAPTER 6

Reflective Compilation

The optimization of object-oriented languages presenting dynamic dispatching is particularly challenging. For instance, the concrete types of objects could not be known without running the application. As a consequence, the concrete behavior that should be executed is often known only at run time, hindering ahead of time optimizations. To make the things even more challenging, reflective APIs allow developers to observe or intercede the application elements programmatically while the application is running.

However, JIT compilation successfully faces many of these challenges by assuming that the run-time variability of applications is generally low. For instance, polymorphic inline caches (PICs) [HCU91] for call sites work well under low variability but their performance decreases when new types are observed. The main problem with dynamic compilation is the trade-off JIT compilers must resolve: since their compilation overhead is included in the program's execution time, JIT compilers rely on complex heuristics and budget models to decide what, when, and how to optimize the application's code [Kul11].

In the absence of mechanisms to interact with compilers, application developers generally treat them as black boxes. This leads to unpredictable performance gains or losses depending on the heuristics hit rate [RSB$^+$14]. For example, a call site with high variability is hard to optimize for PICs. However, in case the compiler heuristic infers that the variability comes from different contexts (when the method is called from different call sites), an aggressive optimizer can clone (split) a method. Consequently, the system incorporates contextual information reducing thus the variability [CUL89].

The problem is that it is hard for these heuristics to capture all possible patterns of execution. By definition, heuristics do not hit the optimal performance for all the cases. Even in the cases they do not hit a maximum but they do well, they may not reach an application's strong performance requirements. For instance, it is challenging for heuristics to identify an application-specific subtle phase change. However, the overall performance would probably improve if it is detected and a deoptimization to remove compiled code based on already out-dated profiling information is triggered.

Compilation heuristics could be adapted by developers whenever they notice a new case in which they do not reach the expected performance. However, this requires to engineer the compiler and possibly the virtual machine. This is costly, and gathering the necessary data to recognize variability patterns might also harm the overall application performance. Therefore, a general solution at the compiler level might be too complex and may introduce unacceptable run-time overhead for other applications. Especially for cases when the variability is highly application-specific.

In this chapter we propose to use the reflective capabilities of a FREE for enabling developers to improve the performance of their applications whenever the compilation heuristics do not reach the expected performance. Furthermore, this performance improvements can be approached without interrupting the applications. Concretely, the compiler should provide introspecting and interceding capabilities of their optimization aspects . Exploiting these capabilities, developers would be able to perform fine-tune optimizations and mitigate application-specific run-time overheads by providing additional information to the compiler about a program at run time. This has the advantage of enabling custom optimizations with a generic API, accessible at the language level, and applicable to different scenarios and applications dynamically.

To design a concrete MOP and provide the corresponding validation we circumscribed to the operations needed for approaching a particular source of sub-optimal performance. Concretely, we detected a special kind of variability in some scenarios performing dynamic adaptations such as those which extensively use dynamic proxies or perform instance migrations. This variability, usually, appears as high variability to general purpose JIT compiler heuristics. As a consequence, they produce run-time overheads. However, application developers know when and how this variability happens (and when it finishes), making it suitable for optimization.

In the remainder of this chapter we first discuss the different kinds of application-specific variability we detected. Then we describe reifications of the compiler behavioral and structural elements that could be affected by those variability conditions. Examples of such elements are dispatch chains, program specializations, application profiling, and code splitting. In Section 8.6 we provide an evaluation of how a subset of the proposed compilation reflective capabilities succeed in improving the overall performance of two scenarios exhibiting application-specific variability.

6.1 Handling Application-Specific Variability

As programming language implementers and application developers, we have been on both sides of the table. On the one hand, as implementers we know that a compiler needs as much information as possible to *recognize* optimizable patterns. On the other hand, as developers we have dealt with code whose performance has degraded significantly without a clear reason. Below, we describe a reason we identified making this undesirable situation to happen.

6.1.1 Motivating Example: Ephemeral Variability

Let us consider an application that periodically walks through the elements of a list. To make it concrete and simple, suppose the elements are instances of a `Point` class and that the application requests their `x` field. The code snippet in Smalltalk syntax is:

```
Example>>#gatherX
  ↑points collect: [:point | point x]
```

Now, suppose that for at least one of the points in the list, there is a *proxy* protecting the actual point object that has not yet been initialized. When the field

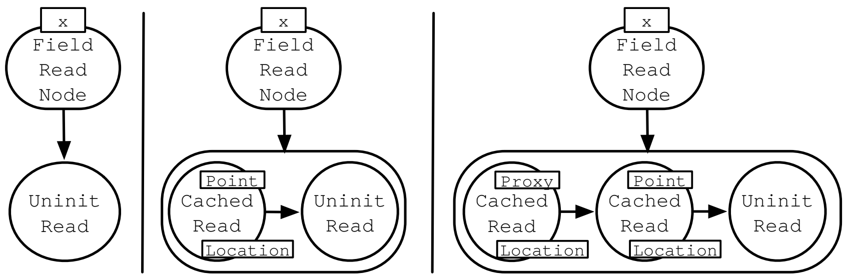

Figure 6.1: Run-time changes in the AST of the *gatherX* method.

read is triggered, the proxy lazily initializes the object and updates the list at the corresponding position. This scenario contains what we call *ephemeral variability*. This variability is in the field access operation, which refers always to the same type (`Point`) with the exception of the first iterations where it also refers to `Proxy`.

Figure 6.1 illustrates the challenges this type of variability poses to a compiler by showing the evolution of the AST representation of *gatherX*.[1] On the left we see the output after parsing. In the middle is the AST after walking through the first point of the list: the uninitialized read was replaced by a specialization that caches the memory offset to look for the `x` field of `Point` instances. The AST remains unchanged until we reach the `Proxy` in the list and add another node for caching the offset for proxies (on the right). Afterwards, the AST no longer changes.

As shown, ephemeral variability affects dispatch chains and thus, performance. In Section 8.6.2 we show preliminary empirical evidence of its performance effects. In our example, a node caching `Proxy` location is at the head of the chain although we know that proxies will never be observed after initialization. This phenomenon, which is application-specific, can be hard to detect in a general-purpose JIT compiler. In particular, the compiler cannot determine when `Proxy` instances will no longer be observed. However, this information can be determined by a developer based on the application behavior. Equipped with the right tools, she could inform the optimizer about the pattern.

6.1.2 Application-specific variabilities

What follows is an informal characterization of different kinds of variability we have observed in our experiments using a reflective VM as an adaptive platform. While we assume they have been observed by many VM implementers before, to the best of our knowledge, they have not been widely documented yet.

Ephemeral Variability: As we illustrated in the motivating example, ephemeral variability refers to an increase in variability that disappears after some time, perhaps a program phase, after which the system continues the execution with lower variability. This is often observable in applications performing adaptations at run time such as instance migration. Another example is unexpected inputs that trigger exceptions and error handling. Ephemeral variability affects optimizations based on type profiling. Concretely, it may prevent inlining and other optimizations, because the dispatch chains will keep information about all previously observed types (especially in commonly used library functions) while the application only needs a subset of them after startup.

[1] We chose ASTs for illustrative reasons but the main idea is generalizable to other representations used by speculative compilers.

Warm up Variability: A particular kind of ephemeral variability that occurs whenever there is a clearly defined initial phase where applications feature highly dynamic behavior, but afterwards, stabilize and exhibit lower variability. For instance, during startup, an application might initialize a complex data structure involving the instantiation of heterogeneous objects and the execution of complex initialization methods. Depending on the time it takes the instance to properly initialize, the motivating example could also be categorized as showing *warm up variability*.

Rare Variability: Some programs have rare but reoccurring variability in their behavior. Examples can be heterogeneous run-time values appearing occasionally or behavior triggered by a periodic task. This leads to short stretches of execution diverging from stable behavior, and thus, causing problems for existing heuristics. One of the reasons is that the rarely occurring behavior can still block important optimizations from being performed for the frequently executed code. So, it might be better to avoid optimizing rarely executed code, and instead keep interpreting it. Moreover, in the cases where it is still worth optimizing, the dispatch chains would be contaminated with the last observed values, and a reordering of the specializations would mitigate this phenomenon. Unfortunately, PICs and other heuristics usually do not take rare variability into account.

Highly Indirect Variability: This phenomenon captures variability generated in different source locations, which are usually far from the points where they finally have an impact. It is usually observed in applications that make extensive use of frameworks, libraries, and/or high-order functions. An example is any standard library method that applies a lambda received as parameter. Different call sites will dispatch different lambdas. To mitigate this variability JIT compilers usually *split* (clone) the library method to enable context-sensitive profiling. This significantly reduces the length of the dispatch chain and promotes aggressive optimizations like the inlining of the lambdas. For highly indirect variability, splitting is not sufficient to distinguish different calling contexts preventing optimization. An example is detailed in Section 8.6.2.

Summarizing, general purpose JIT compilers are usually unable to properly recognize the aforementioned variability *patterns*. The reason is that to keep them tractable, these compilers profile properties that can be easily detected and monitored while these types of variability are usually opaque and application-specific. As a consequence, they lead to performance degradation.

6.2 Towards a Compilation Metaobject Protocol

Application-specific variability can potentially be handled by a compiler if developers provide the proper information at run time. To do so, we propose the use of reflective compilers, *i.e.*, compilers exposing an API to the language level that can be exploited at run time. In this section, we propose a (not necessarily comprehensive) set of operations for such an API. We focus on compilation aspects that may be affected by the already presented types of variability. Concrete examples, along with code illustrating how to use the API, can be found in Sections 8.6.1 ands 8.6.2.

Subsequently, we present the operations based on the entity starting the communication (*i.e.*, application or compiler). We assume an optimizing compiler that works on methods represented as ASTs. However, the ideas can also be applied to other representations, *e.g.*, bytecodes. We reify the essential concepts of ASTs for dynamic languages: nodes, dispatch chains, and profilers. To connect the base and meta levels, each method has access to its AST.

6.2.1 Run-Time Directives from the Application to the Compiler

Applications can observe or modify the state of a compilation by using the following operations:

- Compile: force the compilation of a method. Handy to accelerate warm up times when the developer knows that the variability is low, especially, after a deoptimization. Also for compiling methods with customized specializations.
- Invalidate: discard the optimized code of a method. This is useful whenever compiled code contains specializations that no longer hold or are blocking optimizations.
- Manage compilation information, covering operations for: i) Inspecting the run-time AST nodes of a method to analyze its current state (specially its dispatch chains). ii) Altering the AST. Concretely: ii.a) Add new nodes, useful for customizing specializations ii.b) Reset, reorder, or remove nodes, in particular, specializations from dispatch chains. These can help to remove profile pollution caused by old phases and/or improve guard execution times (see one example in Section 8.6.1).
- Trigger optimizations: force method optimizations such as splitting, inlining, and loop unrolling. Useful when the compiler heuristics fail to provide the optimal performance. (See Section 8.6.2 for an explicit splitting example).
- Activate/Deactive profilers.

6.2.2 Run-Time Callbacks from the Compiler to the Application

These operations allow applications to customize how the compiler responds to compilation events.

- onSpecialization: when a specialization is going to be added because the current subject's type has not been observed before.
- onGuardSucess/onGuardFailure: to inform about the success or failure when executing a guard.
- onNodeReplacement: when a node is going to be replaced with a more specific behavior for the current subject's type.

Concluding, we expect that using the compilation MOP developers would be able to boost the performance of their applications at run time, whenever they notice application-specific variability that is not being fully optimized by the JIT compiler. We preliminary evaluate the usage of the MOP in two scenarios exposing application-specific variability in Section 8.6.

6.3 Resumen en Castellano

Los compiladores dinámicos enfrentan exitosamente muchos de los desafíos que los lenguajes dinámicos les presentan en terminos de desempeño asumiendo una baja variabilidad en tiempo de ejecución. Por ejemplo, las caches polimórficas (PICs) [HCU91], mitigan significativamente las penalidades en los llamados a métodos eventualmente polimórficos cuando la variabilidad es baja. Sin embargo, los compiladores deben resolver una tensión entre el tiempo que les toma compilar el código en tiempo de ejecución y la optimalidad del código resultante. Para resolver esta tensión ejecutan heurísticas que les indican qué código compilar, así como también cuándo y cómo compilarlo. [Kul11].

Dada la ausencia de mecanismos para interactuar con los compiladores dinámicos por parte de las aplicaciones, los desarrolladores generalmente tratan a los compiladores como *cajas negras*. Ésto resulta en niveles de desempeño imprevisibles dependiendo de si la heurístíca abarca los patrones de ejecución particulares de la aplicación desarrollada o no [RSB+14]. En los casos en que no produzcan los resultados esperados, las heurísticas podrían ser adaptadas. Sin embargo esto require gestionar cambios en el compilador y la MV, lo cual no solo es complejo sino que puede derivar en penalidades en otros patrones de ejecución. Sobre todo para los casos en que la variabilidad no abarcada por la heurística es específica de una aplicación y no generalizable.

En el capítulo presentamos una fuente particular de este tipo de variabilidad que resulta en un desempeño subóptimo. Detectamos esta variabilidad al usar ciertas funcionalidades dinámicas como los intermediarios (*proxies*). Describimos con detalle un ejemplo concreto de este tipo de variabilidad (variabilidad efímera) e ilustramos mediante la Figura 6.1 los desafíos que ésta presenta para un compilador. Además describimos otros tipos de variabilidad que resultan en penalidades similares: la variabilidad que se genera en las fases de inicio de algunos sistemas, la variabilidad introducida por comportamientos excepcionales o singulares y la variabilidad que surge a partir de indirecciones profundas. Sugerimos que estos tipos de varibilidad podrían ser efectivamente resueltos por los compiladores dinámicos siempre y cuando estos reciban la información suficiente para afrontarlos.

Para esto proponemos el uso de capacidades reflexivas provistas por el compilador. Nos circunscribimos a los escenarios de variabilidad específica mencionados. Concretamente, presentamos un conjunto de operaciones reflexivas sobre varios de los aspectos de compilación involucrados en esos casos de variabilidad. Este enfoque tiene la ventaja de permitir optimizaciones a partir de una API genérica, accesible desde el lenguaje de programación y aplicable a diferentes aplicaciones sin impactar en las demás.

Nuestra hipótesis es que haciendo uso de estas operaciones, los desarrolladores puedan mejorar el desempeño de las aplicaciones en tiempo de ejecución en los casos en que detecten penalidades endilgables a variabilidades específicas de la aplicación. Validamos preliminarmente esta hipótesis en la Sección 8.6.

Part III

Empirical Evaluation and Experiences

CHAPTER 7

Adaptation Capabilities of Reflective VMs

In this chapter we validate hypothesis 1 by performing a two-phase validation. Firstly, we assess how a reflective VM deals with several adaptation scenarios. Concretely, we analyze in detail extended versions of the three scenario presented in Section 3.1: REO, OPT, and PRO. Furthermore, we present another scenario requiring the organization of fields in memory in a columnar fashion. We compare MATE with partial behavioral reflection [TNCC03], a language-level adaptation framework. In a second phase, we compare MATE with the well-known established language-level approaches for performing dynamic adaptation: metaprogramming, aspect-oriented programming, and context-oriented programming.

7.1 Experimental Setup

We now present our case studies and evaluation criteria. Instructions for reproducing all the experiments can be found at `http://github.com/charig/MatePerformance/tree/papers/phdThesis`.

Case Studies We select a series of heterogeneous adaptive scenarios concerning security, optimization, and profiling aspects appeared in recent literature [TE05, ADD+10, RHB+12, MHR+15, VBLN11]. They are essentially extended examples of the REO, OPT, and PRO scenarios described in Chapter 3. Recall that these case studies usually require the modification of low-level aspects, such as the organization of objects in memory or profiling of execution information.

For REO, we describe two implementations using MATE. We first implement classical per-object immutability and then read-only references based on Arnaud et al.'s *handles* [ADD+10]. We compare our approach with delegation proxies [WNTD14], an approach that model handles without requiring modifications to the VM. More details can be found in Sections 7.2.1 and 7.2.2 respectively.

Regarding OPT, we implement a customized layout based on a hash-based semantics for instances whose fields are mainly unfilled. For PRO, we implement the same behavior as in Senseo [RHB+12], a profiler for leveraging IDEs with applications profiling behavior. More details can be found in sections 7.3.2 and 7.2.3 respectively.

Since REO and PRO concern behavioral aspects while OPT is the only one requiring structural modifications, we add another case study requiring structural adaptations. Briefly, this experiment requires the adaptation of instance fields so that they can be organized in a columnar fashion in memory [MHR+15]. We provide more details in Section 7.3.1. Overall, these scenarios make it possible to evaluate both dimensions of our MOP (organizational and execution). In sections 7.4 and 7.5 we also compare MATE against partial behavioral reflection [TNCC03], aspect-oriented programming and context-oriented programming as a means to understand how powerful reflective VMs can be.

Criteria Below we describe the quality attributes to compare a reflective VM against other solutions. For each scenario we selected the best language-level solution, which generally does not apply to the other scenarios, and compare it to the approach using MATE judging it as either better, the same, or worse. We refrain from giving finer-grain scores to each category as we are not interested in comparing all alternatives with each other.

- *Succinctness*: the amount of lines of code (#LOCs) needed for applying the adaptation. It also considers dependencies on software artifacts such as middlewares, instrumentation frameworks, etc.
- *Forward compatibility*: whether an adaptation persist during the evolution of the application. For instance, if the adaptation is about monitoring some entities and a new piece of code is added, it is desirable that the new code automatically includes the monitoring behavior.
- *Scoping*: whether the approach provides explicit ways to apply changes at different levels of granularity.
- *Modularity*: indicates whether it is possible to implement the adaptation without polluting the application's logic. It also considers undesirable effects on the application such as code bloats (*e.g.*, due to instrumentation).
- *Activation impact*: the amount of changes to the application needed for enabling/disabling the adaptation.

7.2 Extending Language Features

First we present two scenarios requiring the incorporation of new language features to the application at run time already introduced by the REO and PRO scenarios: object / reference immutability and low-level profiling capabilities.

7.2.1 REO: Immutability

Software immutability refers to the ability to turn parts of the program state immutable, that is, only accessible for reading. In OO languages, immutability has been applied at the object level, *i.e.*, protecting a single or set of individual objects, or at the reference level, for instance, for protecting any object (transitively) accessed through a particular reference.

Immutability has been proven useful for software development and testing, optimizations, and verification among other tasks [ZPA+07, TE05]. For instance, during the execution of a test suite it is desirable to enforce that assertion expressions do not cause side-effects. Activating and deactivating immutability on-the-fly for running sanity checks can protect the system against unintended modifications.

Object Immutability in MATE As the following code snippet shows, we simply need to install a metaobject (in the target object) redefining the write operation:

```
 1  class Immutable extends Layout {
 2          def writeField(aNumber, anObject){
 3                  InvalidWriteException signal
 4          }
 5  }
 6
 7  immutableLayout = new Immutable();
 8  immutableEnvironment = new Environment();
 9  immutableEnvironment.setLayout(immutableLayout);
10  obj = new Object();
11  obj.setEnvironment(immutableEnvironment);
```

On lines 1-5, we subclass `Layout` and overload the *writeField* operation so it does not perform the corresponding state mutation. From line 6 on, the code creates the *environment* and links the immutable layout metaobject to it. The last line installs the *environment* in a new object. Deactivating the immutability simply requires to unset the environment: `obj.setEnvironment(NOMETAOBJECT)`.

Comparison to other approaches Zibin et al.'s [ZPA+07] and Javari's [TE05] approaches enforce immutability by relying on static typing. They require modifications of the application-level code (to include type annotations) and recompilation (to recheck the annotations). Instead, object immutability in MATE is also applicable to dynamically typed environments. Accordingly, it is more *succinct* since it does not require modifications of the type system. It is also more *modular* since instead of using type annotations in the application's methods, the adaptation logic is isolated in metaobjects. There is no significant difference in any of the other criteria.

One alternative for implementing object immutability in dynamic environments is to instrument every method (including libraries used) that may eventually modify the state of immutable objects. This has negative impact on *succinctness*, *activation*, and *forward compatibility* compared to MATE. One way to mitigate these issues is to limit the granularity of the immutability property to classes instead of objects. This means that all the instances from a class are mutable or immutable. However, this might be prohibitively restrictive. It makes the alternative worse than MATE in terms of *scoping* and still does not resolve all the problems in the other criteria. On the other hand, VisualWorks Smalltalk[1] and some Ruby versions use a mutability flag in each instance to support per-object immutability. Every time an object is to be changed, the VM first checks this flag and raises an error if mutation is forbidden. These solutions do not suffer from the aforementioned limitations and should be better than MATE in terms of *activation impact* since the adaptation is *hardwired* at the VM level. On the other hand, they require dedicated VM support.

7.2.2 Reference Immutability

Reference immutability protects all objects that can be reached (transitively) through a reference. In Smalltalk references are not first-class citizens. To provide reference immutability we extend the language at run time with Arnaud's *handles* [ADD+10]. Recall from Section 3.2.2 that handles are like proxies to objects that delegate every operation to their targets except mutable operations. Handles must be transparent: a user should not be able to distinguish if she is accessing an object directly or

[1]http://www.cincomsmalltalk.com

through a handle. Moreover, any object accessed through a handle is wrapped into another handle, propagating this way the immutability through the complete chain of objects accessed from a handle.

Reference Immutability in MATE. Below the code for implementing handles using the MOP:

```
class ImmutableMessage extends Message (
        def lookup(subject, aMethodName){
                return super.lookup(subject.getTarget(), aMethodName);
        }

        def activateWithArgs(subject, aMethod, args){
                if (aMethod.name().equals("=="))
                        args["receiver"] = subject.getTarget();
        }
)

class ImmutableLayout extends Layout (
        def read(subject, anIndex){
                return new Handle(subject.instanceVarAt(anIndex));
        }

        def write(subject, anIndex, aValue){
                InvalidWriteException signal
        }
}

class Handle extends Object = (
        fields: target;
        static fields: semantics;
        semantics = (Environment
                withSemantics: ImmutableSemantics new
                andLayout: ImmutableLayout new);

        Constructor Handle(anObject){
                target = anObject;
                this.installEnvironment(Handle.getSemantics());
        }

        def getTarget(){
                return target;
        }

        def static getSemantics(){
                return semantics;
        }
)

class Object = (
        def readonly(){
                return Handle(this);
        }
)
```

Listing 7.1: Implementation of Handles in Mate

Our implementation encapsulates the semantics of the immutability, the transparency, and the propagation properties of handles in four methods within two metaclasses: `ImmutableMessage` and `ImmutableLayout`. Below, the description for each of these properties:

- Immutability: We reuse the *write* method from the previous example.
- Propagation: We redefine the read operation (Lines 13-15) for enforcing that any access to a field of an object referenced by a handle returns a handle wrapping the corresponding field. In addition, in Line 3 we delegate the lookup to the superclass but customize the first parameter. This ensures that the method is looked up in the class of the original subject (the target) and not in the handle. Summarizing, these two methods ensure that messages sent to a handle execute the method from the target and that side-effects are disabled in the chain of activations.
- Transparency: The activation ensures that the identity of read-only references is preserved by overloading the receiver when activating the equality test in Lines 7-8. This way clients always perceive that they interact directly with the genuine objects instead of with handles.

The Handle constructor installs the *handle* metaobject after setting the target (Lines 29-31). Finally, any object returns a handle to itself when executes the readonly method (Lines 43-46).

Comparison to other dynamic approaches Arnaud's implementation of *handles* [ADD$^+$10] duplicates classes. Every class that eventually needs to be immutable has a corresponding *shadow class*. Shadow classes wrap all the methods that change state to forbid the modification. To maintain *forward compatibility*, this mechanism requires changes in the compiler and the instrumentation of bytecode as a means to keep classes updated every time a method of the original class changes. In addition, it has a significant *activation impact* since toggling immutability requires to instrument (eventually) the whole system. Furthermore, the approach requires to adapt the VM so that messages sent to a handle are actually dispatched to the shadow classes.

More recently, an approach based in dynamic proxies modeled handles without requiring modifications to the VM [WNTD14]. Maintaining these proxies requires a code generation technique very similar to that for hidden classes, and so, *forward compatibility* and the *activation impact* are still costly. Moreover, although delegation proxies do not require a dedicated VM, they are not completely transparent: using standard reflection users can identify that they are interacting with proxies.

We showed that in MATE both, per-object and per-reference immutability, can be activated at run time even if it was not anticipated. In contrast to some of the aforementioned approaches, non ad-hoc support, such as shadow classes or method duplications, is needed. Finally, the adaptations do not affect the application's logic and are transparent, for instance, they are not observable whenever application's methods are being debugged. Accordingly, *forward compatibility* is automatic, *i.e.*, eventual modifications to the application would not affect immutability because the adaptation semantics are encapsulated in the corresponding metaobjects.

7.2.3 PRO: Profiling Applications

Profiling is useful for software development tasks such as program comprehension [HCvW07], debugging [AKE08], and the guidance of optimization strategies [DDHV03].

Prior approaches found it important to add calling context information to profiles, for instance, for improving the speed and correctness of software maintenance tasks [RHB+12] or decide whether a method deserves further optimization [Wha00]. The main idea is to make the profiling information sensitive to both, context and flow.

Senseo provides information of running applications to IDEs for helping development and maintanance tasks [RHB+12]. The information is collected in calling context trees (CCT), *i.e.*, a data structure that compactly represents information of method executions grouped by calling contexts [ABL97]. Essentially, a CCT is a tree where each node is an abstraction of calling context: represents the execution of a method and its ancestors represent its callers (*i.e.*, the methods in the call stack). Senseo's CCT collects the following metrics for each node: number of invocations, receiver, argument, and return types, number of object allocations, and allocated bytes.

Profiling Senseo's information in MATE To construct a CCT with the information gathered by Senseo it is mandatory to gather contextual and low-level information at run time. We showcase how to exploit MATE's reflective capabilities to achieve this goal:

```
CCT extends object (
        static fields: instance; // Singleton
        fields: root, current;

        static def getInstance(){
                return instance;
        }

        def logActivation(aMethod, arguments) {
                /* Look or create a node hanging from current targeted
                to aMethod. Then update current to the corresponding
                node and log the arguments to current.*/
        }
)

CCTActivation extends Message (
        def activateWithArguments(aMethod, arguments){
                CCT.getInstance().logActivation(aMethod, arguments);
                return aMethod.activateWithSemantics(this);
        }
)

CCTOperations extends Executor (
        def returnValue(aValue){
                if (thisContex.getMethod().getName() = "new")
                        CCT.getInstance().returnCreatedObject(aValue);
                else
                        CCT.getInstance().returnValue(aValue);
                return aValue;
        }
)

def main(args) {
        profilingEnv = new Environment();
        profilingEnv.setMessage(new CCTActivation());
        profilingEnv.setExecutor(new CCTOperations());
37          thisContext.installEnvironment(profilingEnv);
38          \\below follows the original main code
39  }
```

For developing the CCT data structure we followed the algorithm described in the literature. We refrain from describing it here to focus on the usage of the MOP. To profile all the required information we introduce a single metaobject redefining only the method activation from the `Message` metaclass and the return operation from the `Executor`. Note that this metaobject was designed to work at a method activation granularity, *i.e.*, the semantics of whole method executions are governed by the corresponding *environment*.

Lines 1-14 introduce a partial skeleton of our CCT implementation to help the reader follow the essential adaptation. Lines 16-21 show how to redefine the method activation so that it informs the current CCT which method and with which actual arguments is being activated. In addition, to ensure the propagation of the profiling semantics during the whole execution of the application, in line 19 the metaobject installs itself in the frame of the method to be activated. Lines 23-31 describe how to redefine the return operation so that it informs the CCT its value. In case the value is the result of an instantiation, we call a particular method so that the CCT registers the information and calculates the allocated bytes based on the type of the object.

Finally, in lines 33-39 we show how to adapt the entry point of the application to activate the profiling semantics. We just create an *environment* containing both aforementioned metaobjects and install it to the current activation frame (accessible in MATE through the *thisContext* keyword).

Comparison to other approaches The required profiling information can be gathered using language-level tools such as aspect-oriented programming (AOP) implementations or instrumentation-based frameworks [Dmi03, BBnRR12, VBMA11]. In fact, Senseo relies on Major [VBMA11], an AOP-based profiler. Language-level profiling advantages are mainly its high accuracy (precision) and that they promote portability, flexibility, extensibility, and in recent approaches even moderate overheads. However, *forward compatibility* is hard when using these approaches because every time a new method is added the system must ensure its corresponding instrumentation or the aspect weaving. On the other hand, with different degrees, these approaches have repercussions on *succinctness* because of the dependency on a heavy-weight framework, *modularity* due to the effects of instrumentation, and *activation impacts*.

Using MATE, the same information can be gathered by combining two single metaobjects (with a few lines of code each) that automatically propagate through the chain of activations during profiling. Besides, the evolution of the application does not affect the profiling behavior. Summarizing, it does not suffer from fundamental impacts on any of the aforementioned criteria.

An alternative technique, *sampling profiling* [DHV03, AG05, ZSCC06], produces partial execution information. We refrain here from further comparison because we are interested in producing the same information as Senseo which requires a precise profiling approach.

7.3 Extending Data Representation

This section presents two scenarios requiring structural adaptations to the application's objects at run time. Consequently they serve us to assess the organizational reflective capabilities of the MOP.

7.3.1 Saving Time: Columnar Objects

Most OO runtimes organize their objects in memory as a continuous block of cells (cf. Section 5.1.2 for more details). Using this organization, reading an object field means that not only the field itself is accessed, but the cache-line including the field is loaded into the processor cache. Therefore, iterating over a large amount of objects and accessing only few fields, potentially spanning multiple cache-lines, can result in sub-optimal performance because of frequent caches misses.

Since many applications need to process a great amount of data efficiently, approaches to avoid the overheads for cases where only a few of the fields are actually used have been studied. In relational databases, a technique successfully used for improving analytical algorithms is organizing the data in columns instead of rows [Pla09, Pla13].

Columnar Objects in MATE To improve expressiveness and maintainability by modeling the data in an OO environment instead of a database, below we illustrate how we implement a columnar organization of object fields [MHR+15], at run time, using the MOP:

```
class ColumnarData (
        fields: columnarData, lastPosition;
        static: columnarInstanceEnv;

        Constructor ColumnarData(aClass, initialSize) = (
                columnarData = new Array(aClass.instVars().length());
                lastPosition = 0.
                for ( i = 1; columnarData.length(); i++){
                        columnarData[i] = new Array(initialSize);
                }
        }

        def getData(aProxy, fieldIndex){
                return columnarData[fieldIndex][aProxy.getIndex()];
        }

        def setData(aProxy, anIndex, aValue){
                columnarData[fieldIndex][aProxy.getIndex()] = aValue;
        }

        def newInstance(){
                lastPosition = lastPosition + 1;
                proxy = new ColumnarProxy(lastPosition);
                proxy.installEnvironment(columnarInstanceEnv);
                return proxy;
        }
)
```

The code shows *ColumnarData*, the main auxiliary class we need to implement columnar classes in a reflective VM. It essentially stores each of the fields of the class in a separate array with one fixed position for each instance of the class. Then instances become just *proxies* storing the object class and the column position. Note that Mattis et al. [MHR+15] empirically demonstrated using a tracing JIT compiler that within loops traversing collections these proxies can be optimized out. The main idea is to leverage escape analysis techniques for converting them to just scalars (the column number). The code also defines the logic for accessing (Lines 13-15) and storing (Lines 17-19) the fields of the instances in the corresponding field array and at the proper index. Accordingly, instances of a columnar class just store their assigned index and access their fields by gathering the value in the corresponding index and array respectively.

ColumnarData is also responsible for creating the new instances of classes featuring a columnar representation (Lines 21-26). At instance creation time, we select a new index for the class arrays storing the data and create a kind of proxy storing only this index. This proxy represents the new instance. Like the handles presented in the immutability scenario, it is desirable for this in-memory organization to be transparent. To do so and also make them access the data in the proper arrays without changing the code of the getters and setters of the class, line 24 shows that we install the following simple metaobject:

```
class ColumnarFieldSemantics extends LayoutMO (
        static columnarClasses = new Hash();

        def read(anIndex){
                return columnarDataFor(this).getData(this, anIndex);
        }

        def write(anIndex, aValue){
                columnarDataFor(this).setData(this, anIndex, aValue);
        }

        def static columnarDataFor(aProxy){
                return ColumnarClasses.at(aProxy.class());
        }
)
```

ColumnarFieldSemantics metaobject defines a static association between each columnar class and its ColumnarData organization. Furthermore, it redefines the read and write operations so that the instance actually access the data in the ColumnarData arrays at the corresponding index. It is worth noting that this metaobject is actually augmented with a similar behavior to the one described for handles (see Section 7.2.2) for ensuring that proxies are transparent.

Comparison to other approaches Our approach only requires to install a metaobject redefining the reading and writing of fields. Note that the same behavior could be achieved by removing the fields of the required classes and changing the *getter/setter* for every field so they access their corresponding index in the arrays storing the data. Direct accesses to fields must be forbidden, *i.e.*, by using only accessor methods. Anyway, this alternative is clearly not *modular* (it modifies the application classes) and has a significant *activation impact*. On the other hand *forward compatibility* is not provided automatically: if the object's layout changes, for instance because a field is added, the columnar adaptation must be updated too.

Avoiding most of these problems, Mattis et al. [MHR+15] presented a Python library which introduces annotations for classes so that their fields are organized in a columnar layout. They show that tracing compilers are able to optimize operations such as selection, filtering, and mapping on large amounts of data if the instances are organized in a columnar layout in memory. Developers must annotate the classes that should use a columnar layout, but the algorithms and the code of the application for field accessing remains the same. The approach relies on implementing proxies ensuring the interception and redirection of every access to any columnar field. However, they explicitly mention in their paper limitations of the approach concerning object identity (transparency of proxies) without featuring a dedicated VM.

7.3.2 OPT: Saving Space

Following up with layouts, now consider the OPT scenario. Recall that, essentially, there exist a system whose data model includes the representation of objects with a significant amount of unused fields. To make it concrete suppose now there is a *Person* class with 20 fields for describing different attributes on an individual. If most of the fields in Person's instances are not filled, a significant amount of memory is being wasted. In case of memory pressure, an alternative to reduce the memory consumption could be to convert these instances to a dictionary-like representation. The goal is to finally store fewer fields per instance.

Hash-based Layouts in MATE. To implement a dictionary-like representation in MATE and reduce the memory consumption at run time we propose to make the layout of sparse objects rely on a hash-based representation. Concretely, the MOP provides the means to create custom structural *layouts* (layouts are first-class). Depending on the empty fields for each person we can dynamically assign a layout that stores fewer fields per instance and provide a metaobject for realizing a hash-based for this layout.

For instance, consider a layout storing ten fields. Hence, Person's instances filling the individual information with less than five fields could be assigned this layout. The reason why the layout has ten fields instead of five is that our hash-based representation needs two fields for representing each original field: the first stores the data and the second the index (or name) of the original field. Below, the customization of `Layout` for implementing the hash behavior:

```
class HashBasedLayout extends Layout {
        def read(anIndex){
                index = this.hashIndexForField(anIndex);
                if (index.isNull())
                        return null;
                else
                        return this.instVarAt(index);
        }

        def writeField(anIndex, anObject){
                index = this.hashIndexForField(anIndex);
                if (index.isNull())
                        throw new NoMoreSpaceException();
                else {
                        this.instVarAtPut(index, anObject);
                        this.instVarAtPut(index + 1, aNumber);
                }
        }

        def fieldsCount() {
                this.class().instanceVariables().size();
        }
}
```

The `HashBasedLayout` metaclass essentially adapts the reading and writing of fields for working with the aforementioned hashed-based organization. For both operations we first need to look for the position of the hash for that field and then do the concrete operation. In addition, for ensuring consistency and transparency, we also redefine the method that returns the quantity of fields of an object. If a user queries the number of fields of a person with the hash-based layout she will still receive twenty as an answer. Note that the hash-based representation we proposed allows us to save space in all the cases where the fields actually filled are less than half of the total.

Comparison to other approaches Another approach to avoid the waste of memory caused by optional fields could be the migration of sparse instances to new classes. This would however require to change both, the application code and the instantiation points, affecting thus *modularity*. Furthermore, depending on the application, this may require changes in a significant amount of lines of code or the adoption of usually large DSU frameworks for managing the migration at run time [HN05]. This impacts on *succinctness*. Furthermore, since this alternative spreads the adaptation logic into all the classes requiring a different layout it has a significant *activation/deactivation impact*. Finally, the adaptation is scattered through the application code, which is bad for *modularity* and *forward compatibility*. In contrast, a reflective VM does not depend on how the application is implemented and does not suffer from any of the aforementioned drawbacks. The core of the application remains the same, there is no need to replicate classes and swaping from array-based to hash-based layouts only requires to copy the instance's fields from one to the other.

Similar to MATE, Verwaest et al. [VBLN11] reify layouts at the language level. However, since the VM is not aware of them, these layouts can be bypassed by primitive operations that do not recognize those constructs. On the other hand, dynamic languages such as JavaScript or PHP represent properties of objects with hashed-based dictionaries. Without an aggressive optimization, this turns inefficient when most of the fields are used. In contrast to them, with a reflective VM we can switch between hash-based and array-based representations at the granularity of each instance depending on the convenience for each case.

7.4 Partial Behavioral Reflection with a Reflective VM

We now describe how a reflective VM supports the capabilities of partial behavioral reflection (PBR) [TNCC03], a language-level alternative for realizing dynamic adaptations. Reflex is the first tool implementing PBR and has been used to introduce novel features to different languages. Its limitations lie mainly in its implementation: Reflex works by instrumenting Java bytecodes at load-time, and thus, adaptations have to be anticipated. A subsequent implementation for Smalltalk, based on run-time instrumentation, supports unanticipated partial behavioral reflection (UPBR) [RDT08]. Figure 7.1, taken from the original PBR paper, illustrates its main concepts and their relationships:

- *Hooksets* are sets of operation occurrences (hooks). For each adaptation, hooksets denote all the operations of the whole system that may eventually need to delegate their execution to the metalevel.

- *Links* bound metaobjects with hooksets and establish the protocol between the base and metalevels. It specifies the pieces of information passed to the metaobject. Furthermore, other attributes characterize a link, such as the control given to the corresponding metaobject (i.e., acting before, after, or around the intercepted operation occurrence).

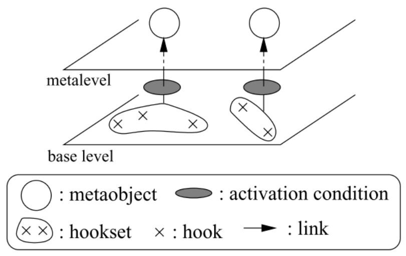

Figure 7.1: Partial Behavioral Reflection main concepts.

- For each association between a link and hook, an activation condition is executed to test whether the delegation to the metalevel must be done or not.

To illustrate these concepts with an example consider that, similarly to the PRO scenario, we need to log the activation of every method with more than one argument. In this case the hookset would denote all the method activations in the application. The link is the binding of every operation of the hookset with a metaobject responsible of doing the actual logging. The activation condition is whether the activated method contains more than one argument.

Hooksets, Links, and Conditions in MATE We devise how UPBR concepts can be modeled in a straightforward manner using the MOP:

- Hooksets are sets of operation occurrences. `Executor` and `Message` metaclasses can capture any of these operations. Furthermore, MATE enables to scope the selection of these operations to single instances.
- The link connects the base and meta level. Essentially, it bounds hooksets with language-level methods defined in metaobjects. *Environment* metaobjects play the same role. Furthermore, MATE enables to assign *Environments* to different base-level entities giving a fine-grained scoping control for defining hooksets.
- Since in MATE the IH activation conditions are fixed, the link activation condition must be defined in the methods that conform the metaobject.

Comparing the approaches We showed how a reflective VM can model PBR concepts. The inverse however does not hold: PBR can not express the structural scenarios presented in the previous section. Moreover, PBR implementations presented in literature depend on instrumentation frameworks. This makes MATE more *succinct*. In cases such as the aforementioned profiling of method activations, PBR must instrument all the methods from the application making the *activation/deactivation impact* significant. *Forward compatibility* is also harder than MATE because PBR must reexecute the instrumentation after updating any method.

7.5 A Fully Reflective VM vs. Language-Level Frameworks

A recent survey [SGP13] on self-adaptive systems asks: Is it possible to adopt a single paradigm providing all required abstractions to implement adaptive systems? To answer this question, Salvaneschi et al., the authors of the survey, evaluate contemporary reflective systems (RS), aspect-oriented programming (AOP) and context-oriented programming (COP). Since the authors identified strengths and weaknesses for all the approaches, their conclusions were not definite.

We already showed how a fully reflective VM successfully handle an heterogeneous series of adaptive scenarios. We also showed in Section 3.2.3 the limitations that reflective systems, aspect-oriented programming, and context-oriented programming present for handling directly those scenarios. We now provide a brief discussion describing how a fully reflective VM subsumes the adaptive capabilities provided by these three solutions.

Reflective Systems

By definition, a FREE subsumes contemporary RSs because, ideally, a FREE reifies every entity. Nonetheless, we have just provided the proper validation by describing how a FREE subsumes PBR, a reflective framework that beyond its standard reifications, introduces a special syntax for expressing advanced behavioral adaptations

Aspect-Oriented Programming

Joinpoints are precise locations of particular operations within the application. At these locations, a pointcut language enable to express behavioral variations that are also expressible with standard programming language methods. Since a FREE can capture any operation and redefine its semantics with language-level methods, it is possible to implement any pointcut language in top of a FREE. For instance, a FREE can redefine any operation's behavior completely, or just call to another method before or after delegating the execution to the standard behavior. Hence, AOP's *before, after, and around* join points for any operation are modeled naturally in a FREE. On the other hand, one of AOP's most salient features is the decoupling of the crosscutting concerns from the application's logic. MOPs can be designed for supporting the same property by promoting mechanisms for composing metaobjects regarding cross-cutting concerns.

Context-Oriented Programming

The main mechanism to support COP is the redefinition of method lookups and activations so that they take into account the contextual information and the activated layer. By definition, a FREE reifies both concepts. On the other hand, layers just group contextual-dependent behavioral variations. They can still be expressed with any way of grouping methods such as in a class or interface. In particular, when the adaptations concern VM semantics a FREE provides also the possibility to model layers by composing metaobjects.

7.6 Discussion

We used for the experiments a reflective VM that follows the MATE architecture. As such, it demonstrates that it is feasible to implement a VM with advanced reflective capabilities. While more experimentation is needed to fully understand the advantages and drawbacks of adding reflection to the VM domain, we consider that our validation provides positive indicators for Hypothesis 1. VMs supporting a bidirectional communication at run time between themselves and the applications are an appealing alternative for developing flexible software.

Regarding the adaptive scenarios, we compare a reflective VM with other language-level solutions. We obtained encouraging results according to the quality criteria we defined in Section 7.1. Concretely, in all the cases MATE managed to deal with the adaptation scenario by installing small metaobjects (in the order of 20-45 LOC each) and it managed to do it without changing the methods that conform the application's logic. Finally, a worth mentioning characteristic is that our MATE handled all the cases while well-known alternatives, such as AOP implementations or instrumentation frameworks, could not deal with some of them.

A potential threat to the validity of our results is the selected set of quality attributes. Furthermore, the coarse-grained values we defined could be considered too fuzzy. A more precise analysis could be obtained by conducting user studies. This is challenging because it would require stable tools for each of the alternative approaches, which are not available at the moment. Furthermore, the spectrum of tools pervades several conceptual abstractions. This becomes even more challenging due to the lack of experts for performing a fair and consistent study. Another way to mitigate this threat would be to find proper quantitative metrics such as time or number of lines of code to apply the adaptations.

We are also aware that our empirical results regarding adaptability may not generalize to every adaptation scenario. To mitigate this threat we have carefully selected examples of adaptation scenarios from existing literature. Moreover, we covered behavioral and structural adaptations at different abstraction levels. We compared our solution using MATE with other language-level approaches such as handles, AOP tools, and PBR. It is worth noticing that there exist few approaches able to address low-level adaptive scenarios at run time, and to the best of our knowledge, none is able to handle our whole set of experiments.

To partially mitigate potential risks that may appear when using the VM reflective capabilities, we select a MOP as a means to control the communication with a precise API. Besides, we expect users to apply careful (and eventually formal) reasoning before using it. It is worth noting that the architecture presented in this thesis do not include any control on who installs metaobjects. Consequently, anyone could modify the metaobjects installed in other objects. This may violate the security of the system. For instance if the metaobject enforcing the readonly property in the REO scenario is removed. We think that the proper approach to solve this issue is to provide a security layer in top of the reflective model such as, for instance, an object capabilities model.

7.7 Resumen en Castellano

En este capítulo se valida la hipotesis 1 mediante una evaluación que consta de dos fases. Primero analizamos cómo una MV reflexiva permite modelar varios escenarios adaptativos. En concreto, se analizan en detalle versiones extendidas de los tres escenarios presentados en la sección 3.1: REO, OPT, y PRO. Además, presentamos un nuevo escenario que requiere la organización de campos de los objetos en memoria en un formato de columnas. En una segunda etapa, comparamos MATE con tres enfoques establecidos de nivel de lenguaje para realizar adaptaciones dinámicas: la meta-programación, la programación orientada a aspectos y la programación orientada a contextos.

Respecto a los casos de estudio, para REO se describen dos implementaciones usando MATE. Primero se implementa la immutabilidad por objeto clásica y luego la immutabilidad por referencias basada en la solución propuesta por Arnaud y otros [ADD+10]. Para OPT implementamos una customización de layout basado en una semántica de diccionario para instancias cuyos campos son mayormente no utilizados. Por último para PRO implementamos el comportamiento de monitoreo requerido por la herramiento Senseo [RHB+12], que utiliza esta información para aumentar la información de las IDE de desarrollo con datos dinámicos. Para el nuevo caso de estudio implementamos layouts que permiten la alocación de campos de los objetos en memoria en un formato de columnas [MHR+15].

Para cada escenario, seleccionamos la solución de nivel de lenguaje que cosideramos mejor para afrontarlo y la comparamos con MATE juzgandola como mejor, peor o igual respecto a un conjunto de atributos de calidad. Estos abarcan aspectos relativos al tamaño, el alcance, la compatibilidad, la modularidad y el costo de aplicar la solución adaptativa.

Finalmente, en las secciones 7.4 y 7.5 comparamos MATE con con las soluciones alternativas de nivel de lenguaje mencionadas para corroborar cuán potentes son las MVs reflexivas. Concretamente, para ilustrar el beneficio de usar MATE describimos la manera de implementar las abstracciones que las alternativas promueven usando MATE. La recíproca no es válida ya que cada una de estas alternativas no puede lidiar con algun aspecto de los escenarios descriptos en el capítulo.

Respecto a los resultados, la conclusión es que dado que utilizamos dos MVs reflexivas que implementan la arquitectura MATE y abordan con suficiencia los casos de estudio introducidos, este capítulo demuestra la factibilidad de nuestro enfoque. También suministra un importante caudal de indicadores positivos respecto a la Hipótesis 1. Las MVs que soportan una comunicación bidireccional en tiempo de ejecución son una alternativa plausible para desarrollar software flexible.

Cabe notar que nuestros resultados empíricos podrían no ser generalizables a otros escenarios. Para disminuir este riesgo seleccionamos escenarios adaptativos de la literatura existente en el área. Además, estos cubren aspectos tanto comportamentales como estructurales y a diversos niveles de abstracción.

CHAPTER 8

Performance Evaluation

In this chapter we assess H2 and H3 by performing an empirical validation. Both hypotheses regard performance aspects and we use MATE$_{PE}$ and MATE$_{MT}$ to perform the assessment. We first present a series of VM benchmarks and compare their running times in the MATE VMs with their running times in standard VMs. This way, we assess the effectiveness of our optimizations in the cases where the VM level reflective capabilities are not used. Then we evaluate performance overheads of redefining VM operations, such as the activation of a method, at the language-level. Later on, we evaluate the performance impacts of using the MOP in more comprehensive cases such as the REO and PRO scenarios. Finally, we show two use cases in which, exploiting the compilation reflective capabilities enables performance gains on applications exposing application-specific variability.

8.1 Experimental Setup

To account for the non-determinism in modern systems and the adaptive optimization techniques in the different implementations, each reported result is based on 50 measurements after the JIT compiler has already finished its optimization work. This is usually considered the *peak-performance* of the system. However, there has been discrepancies on how to measure peak performance [BBTK$^+$17]. Therefore, we describe precisely how we determine the moment at which peak-performance starts in the context of this evaluation.

We first run 1500 iterations of the same benchmark in a single process execution and store the wall-time clock it takes to finish for each iteration. We then execute a changepoint analysis following Barrett et al.'s methodology [BBTK$^+$17]. This process divides the series of data in several segments. Each segment of data has a significantly different mean or variance compared to the previous segment. We select the first segment with at least 50 elements and which mean is no more than 10% above the minimum value of all the samples. In the particular configurations of benchmark/platform in which the condition is not met, we manually select the iteration number at which peak-performance starts based on plots of the raw data. Section 8.5 provides an elaborated discussion of the warmups for MATE$_{PE}$ and MATE$_{MT}$.

On most of the experiments we report an execution rate between our implementations and a predefined baseline. This baseline depends on the experiment. To compute the rate, for each benchmark we calculate the mean of the wall-time clock for the selected iterations and for the corresponding baseline. Then, we calculate the rate following Kalibera et al.'s [KJ13] methodology. Note that they refer to this rate as the speed-up factor. Since we are measuring overheads instead of speed-ups we call it overhead factor (OF). Consequently, we report overhead factors along with its corresponding 95% confidence interval.

The benchmarking machine is a quad-core Intel Core i7-3770, 3.40 GHz with 16 GB RAM, running Ubuntu with Linux kernel 4.2, and Java 1.8.0_91 with HotSpot 25.91-b14.

8.1.1 Research Questions

RQ1: What is the *inherent* peak performance overhead of running programs in a reflective VM?

The inherent overhead intends to capture the cost in terms of performance of running an application in a reflective VM even if none of its particular features are going to be used. In other words, it captures the overhead of using a reflective VM just as a standard VM.

RQ2: What is the peak performance overhead of redefining *individual VM operations* at the language level?

Redefining operations at the language-level means that instead of executing any of its features, the VM delegates the responsibility to a language-level method. This has a cost because more code is executed and the intercession handling must deal with the conversion of values between the VM and the supported language (*i.e.*, between Java and Smalltalk in MATE$_{PE}$).

RQ3: What is the peak performance overhead of dynamically adapting *running applications* by redefining VM operations at language level?

This question intends to capture the performance overheads when several intercession handling sites along the system observe metaobjects redefining operations. This is the scenario expected when performing dynamic adaptations in real applications. Furthermore, it also intends to capture the impact of ubiquitously creating instances with metaobjects installed. Last but not least, this question also intends to capture the overheads when attaching metaobjects to different scoping levels.

RQ4: What is the warmup overhead of reflective VMs?

The warming up includes all the time that it takes to the VM to load, plus the time it takes to the JIT compiler to optimize the current application for the observed values. It is a capital aspect for short-running applications, but it is also important for analyzing the overhead of deoptimizations. With this question we intend to capture how an ubiquitous intercession handling impacts on this performance aspect.

8.1.2 Subjects

Below we describe the subjects for each of the experiments.

Baseline. For comparing SOM$_{PE}$ and SOM$_{MT}$ (our baselines in some experiments) with state-of-the-art VMs we selected the set of benchmarks proposed by

Marr et al. [MDM16] for cross-comparing language implementations. The benchmarks were collected from various sources such as Computer Language Benchmarks, Octane, as well as independent additions. Their size ranges from 2 to 20 classes, and 20 to 350 lines of executed code. Furthermore, they usually present a low variability. There exist only 17 call sites with more than two different receivers and 15 are within the same benchmark (DeltaBlue). For further details, we suggest the reader to follow Marr et al's. paper.

Inherent. To assess the inherent behavior of a reflective VM we run the baseline benchmarks with a couple of own additions. These additions make a deep use of loops, arrays, and randomization values. We consider they may help to better understand a possible limitation of reflective VMs in general, or of our implementations in particular.

The reasons to use this whole set of benchmarks are:

- They do not execute any kind of meta-level behavior.
- They were already used to compare different language implementations mainly measuring the effectiveness of compiler optimizations.
- They were especially selected to evaluate a core of language abstractions usually found on dynamic languages.

Overall, we consider these benchmarks conform an extensive set of language behaviors to capture solely the inherent overhead of a reflective VM. The only reason why we do not also use our own additions for the baseline experiment is that they are written only in Smalltalk and the baseline also includes a JavaScript runtime.

Individual Operations. To assess the overhead of redefining operations of the VM we designed micro-benchmarks with the goal of capturing only the overhead of those operations. For each operation, we provide a benchmark that does the minimal things to perform the operation. Another version of the benchmark does exactly the same but including a metaobject that redefines only the corresponding VM operation. The redefinitions in MATE implement exactly the original VM behavior but using language-level operations. We use randomization to hinder optimizations that may remove the evaluated operations. To illustrate these benchmarks we show the code for the read field operation:

```
class FieldRead extends Benchmark {
  def oneTimeSetup() {
    object = new Pair();
  }

  def benchmark() {
    object.key(Random.next() % 100 + 1);
    return object key;
  }

  def verifyResult(result) {
    return result <= 100;
  )
}

class VMReflectiveFieldRead extends FieldRead {
  def oneTimeSetup() {
    super.oneTimeSetup();
    readEnv = new Environment();
    readEnv.setExecutor(new FieldReadSemanticsMO());
    object.installEnvironment(readEnv);
  }
}

FieldReadSemanticsMO = OperationalSemanticsMO (
  def read(anIndex) {
    return self.instVarAt(anIndex);
  }
)
```

MOP. We evaluate the performance overheads for simpler versions of the REO and PRO scenarios. For REO, recall our implementation of Handles in MATE from Section 7.2.2. For the sake of analyzing the performance overhead of MATE against language-level alternatives, we also implemented a version of delegation proxies [WNTD14] that *does not* use metaobjects. To compare both implementations, we run a program that creates a (read-only) reference to the head of a linked list and traverses the list attempting to write some of its elements. We use the MATE's implementations with standard mutable references as baseline. Note that the version with handles attaches metaobjects to handles whenever a reference is read (Lines 3 and 24, Listing 7.1). In our setting, changing metaobjects is a slow operation since it requires to modify the object's layout. To avoid this slow operation during the traversal, we exploit MATE's reflective capabilities on layouts. Concretely, we cache a predefined layout with a metaobject defining the read-only behavior already assigned, and instantiate handles using this layout.

For PRO, recall that in Section 7.2.3 we introduce a metaobject that profiles the system by building calling context trees. For the experiment, we implemented a simpler version that only accounts for the number of activations made by the program. It is intentionally simple because we are interesting in measuring the overhead of the intercession handlers rather than the "tracing" itself. To do so, the metaobject redefines the method activation operation to increment a counter before invoking the original activation logic. We run this tracing behavior on DeltaBlue, CD, and Json macro benchmarks, as well as on QuickSort and NBody micro benchmarks. In an attempt to simulate the execution of more realistic applications we increased the input size for the benchmarks, and used a 55MB file for the Json parser instead of the original of 90KB.

Assumptions. We also design two benchmarks for assessing the performance of the system when the assumptions of the optimization model do not hold. To do so, we pseudo-randomly select an element from a list of 20 objects and read a fixed field. In a monomorphic version (mono), all the objects use the same metaobject redefining the read operation. In a megamorphic (mega), we simulate a megamorphic intercession handling site by assigning a distinct metaobject (redefining the read operation with the same behavior) to each object.

8.1.3 Reflective Virtual Machines

This chapter's goal is to assess the feasibility of running a reflective VM with an ubiquitous intercession handling mechanism for providing VM level reflective capabilities with low overheads. To do so, the best approach would have been to build an

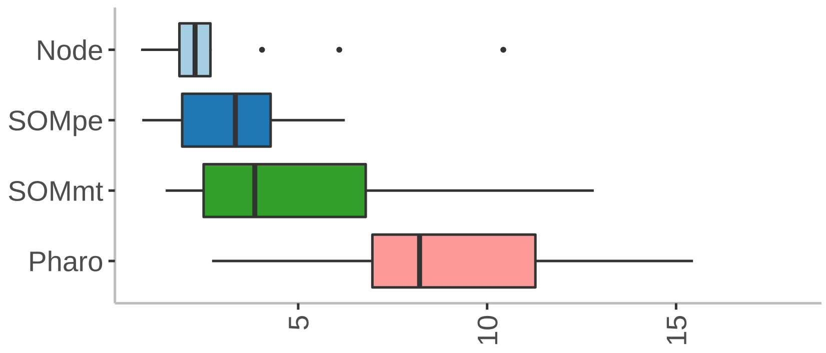

Figure 8.1: Overhead factor of different dynamic programming language implementations normalized to Java. Benchmarks were selected from Marr et al. in [MDM16]. The suite was designed for cross-comparing language implementations

Table 8.1: Overall Baseline Results

Runtime	OF	CI-95%	Sd.	Min	Max	Median
Node	3.01	<1.58 - 4.45>	2.49	0.84	10.43	2.27
SOMpe	3.35	<2.34 - 4.37>	1.75	0.87	6.23	3.34
SOMmt	5.26	<3.12 - 7.39>	3.70	1.49	12.82	3.85
Pharo	8.88	<6.5 - 11.26>	3.94	2.72	15.45	8.21

ad-hoc runtime specially targeted to optimize the particularities of reflective VMs. However, this is a highly resource demanding task. In contrast, we decided to approach this goal with the VMs we already introduced in Chapter 5: MATE$_{PE}$ and MATE$_{MT}$.

We provide two implementations because each framework (meta-compilation approach) further optimizes some scenarios while present limitations with others. Consequently, the usage of two reflective VMs allows us to better explore if the limitations in any of the cases suggests that there is a fundamental issue or if it seems to be accidental. For instance, if a benchmark presents significant overheads in both implementations, that would be a strong indicator that some fundamental indirections can not be removed. Otherwise, it is more plausible that some accidental issue is jeopardizing optimizations.

8.1.4 Baseline Performance

In several experiments we present overhead factors of MATE$_{PE}$ and MATE$_{MT}$ with respect to SOM$_{PE}$ and SOM$_{MT}$ respectively. To understand the impact of using SOM implementations as baseline we first analyze how its implementations stand in terms of performance. We compare the performance of SOM$_{PE}$ and SOM$_{MT}$ with two industrial-strength implementations of languages with similar characteristics: the Cog VM [Mir11] for Pharo Smalltalk and Node.js for a mainstream language like JavaScript. Note that Node.js run on top of Google's V8 VM, highlighted as an example of a highly-optimized VM by the programming language community. Java 8 on the HotSpot JVM is the baseline of this experiment.

Figure 8.1 shows that both SOM implementations considerably outperform Pharo. Furthermore, their mean peak-performance approaches that of Node.js. Table 8.1 shows the relevant statistical variables for each implementation. Precisely, SOM$_{PE}$ and SOM$_{MT}$ are on average only 3.35 x and 5.26 x slower than Java in the JVM for this set of benchmarks. Note that SOM$_{PE}$ is significantly faster than SOM$_{MT}$. The main reason was suggested as the amount of engineering effort put on the

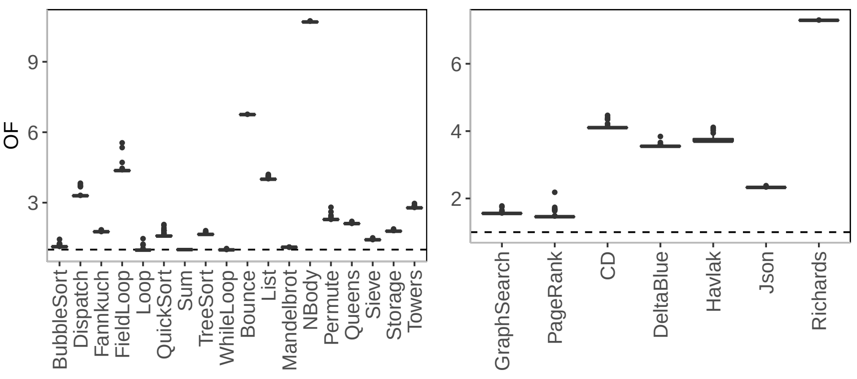

Figure 8.2: Overhead of MATEpe normalized to SOMpe when optimizations at the VM-level are not enabled. The results are clustered by micro (left) and macro (right) benchmarks.

meta-compilation frameworks [MD15]. While Truffle and Graal produce very efficient machine code, RPython has still optimization opportunities. Finally, Node.js presents an overhead factor of 3.01 while Pharo is 8.88 x slower than Java.

We conclude that, although research prototypes, our baseline implementations can be considered highly optimized. The complete results for each benchmark and VM configuration individually can be found in the appendix (Section 10.3.1).

8.2 Basic Performance Overheads

In this section we assess the inherent overhead of running programs in a reflective VM as well as the overhead of redefining individual VM operations at the language-level.

8.2.1 Inherent Overhead

For assessing the inherent overhead we analyze the overhead factors of MATE$_{PE}$ and MATE$_{MT}$ compared to SOM$_{PE}$ and SOM$_{MT}$ respectively.

Overhead without optimizing the meta level. Figure 8.2 shows that, even if no metaobjects are being used, there is an inherent overhead from executing intercession handling sites that the compiler could not optimize. The overall results are shown in table 8.2. The overhead factor is 2.78 for micro benchmarks, 3.45 for macro benchmarks, and the worst cases are NBody NBody with an overhead factor of 10.7 and Richards with 7.29 correspondingly.

Overhead when optimizing the meta level. Figure 8.3 shows the results when our optimizations are enabled. The overall results are also summarized in table 8.2. in MATE$_{PE}$ the mean overhead factors are 0.85 and 0.81 for micro and macro benchmarks respectively, with worst cases of 1.03 for BubbleSort and 0.96 for CD. Notice that some runs are even faster in MATE$_{PE}$ than in SOM$_{PE}$. This may be influenced by the use of different memory layouts, different inlining decisions, and cache effects triggered by the additional intercession handling code. On the other hand, in MATE$_{MT}$ the overhead factors are 1 and 1.05. The worst cases are 1.05 for QuickSort and 1.19 for CD.

Overall, this results shows that our optimization strategy (cf. Section 5.3) manages to reduce the overheads caused by MATE's intercession handling significantly

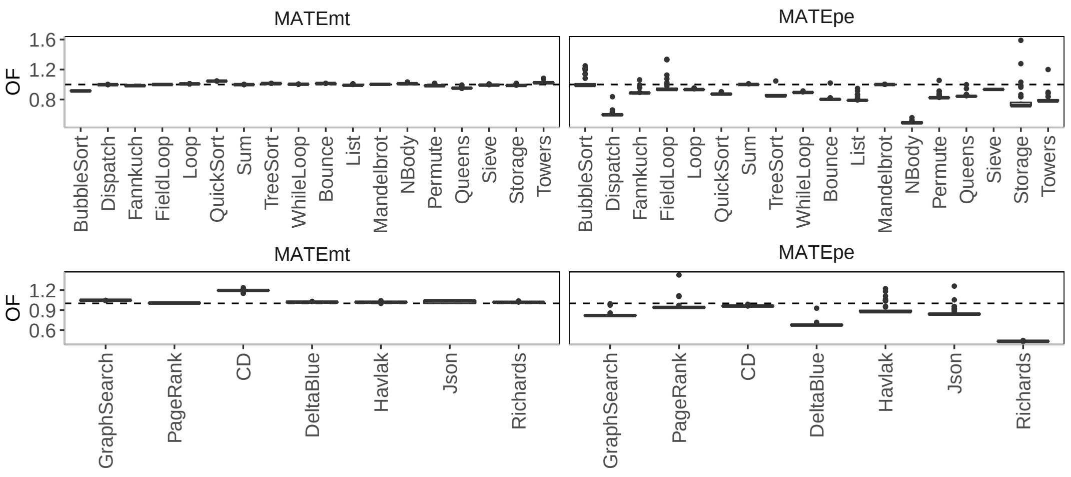

Figure 8.3: Overhead of MATEpe normalized to SOMpe when optimizations at the VM-level are enabled. The results are clustered by micro (top) and macro (bottom) benchmarks

in the case the VM reflective capabilities are available but not used.

Answering RQ1, we conclude from these results that a reflective VM can present almost neglible performance overheads in comparison to state-of-the-art VMs when metaobjects are not activated.

8.2.2 Individual Operations of the MOP

To analyze the impact of redefining VM operations at the language level we assess the overhead of redefining field reads/writes, method dispatches, local variable reads/writes, parameter readings, returns, and all operations together.

Table 8.2: Overall Inherent Results

Runtime	OF	CI-95%	Sd.	Min	Max	Median
MicroBenchmarks						
MATEpe	0.85	<0.79 - 0.92>	0.13	0.49	1.03	0.86
MATEmt	1.00	<0.98 - 1.01>	0.03	0.91	1.05	1.00
MATEpe-NoOpt	2.78	<1.54 - 4.01>	2.49	0.99	10.70	1.78
MacroBenchmarks						
MATEpe	0.81	<0.63 - 0.98>	0.19	0.43	0.96	0.87
MATEmt	1.05	<0.99 - 1.11>	0.07	1.01	1.19	1.02
MATEpe-NoOpt	3.45	<1.6 - 5.3>	2.00	1.50	7.29	3.56
Aggregated						
MATEpe	0.84	<0.78 - 0.9>	0.15	0.43	1.03	0.87
MATEmt	1.01	<0.99 - 1.03>	0.05	0.91	1.19	1.01
MATEpe-NoOpt	2.97	<2 - 3.93>	2.34	0.99	10.70	2.11

Overhead without optimizing the meta level. When using the MOP, the VM without optimizations incurs in overheads of between 2 and 4 orders of magnitude. Since this makes each benchmark to take days instead of minutes to finish, we refrain from now on to show plots when the optimizations are not enabled.

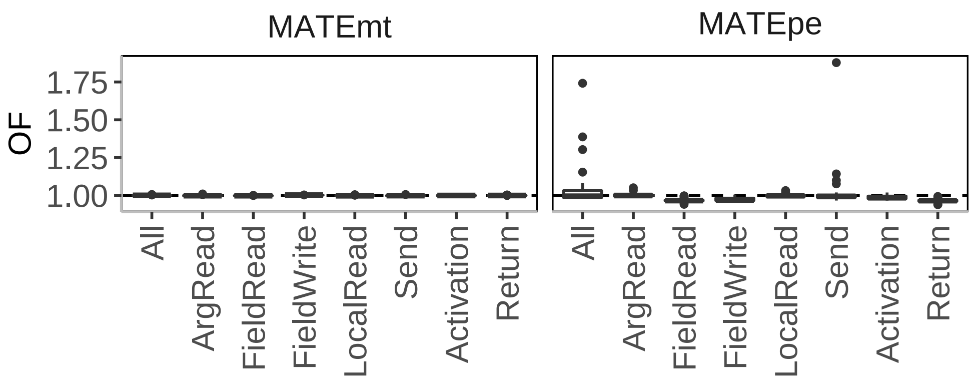

Figure 8.4: Overhead factor for redefining VM operations at the language-level. The baseline for each bench, is the execution of the same benchmark in the same VM but without redefining the corresponding VM operation.

Table 8.3: Overall Results for the Individual Operations

Benchmark	VM	OF	Confidence	Sd	Median	Min	Max
All	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.01
	MATEpe	1.04	<1 - 1.08>	0.13	0.99	0.97	1.74
ArgRead	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.01
	MATEpe	1.00	<1 - 1.01>	0.01	1.00	0.99	1.05
FieldRead	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.00
	MATEpe	0.97	<0.95 - 0.99>	0.01	0.97	0.94	1.00
FieldWrite	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.00
	MATEpe	0.97	<0.95 - 1>	0.01	0.97	0.96	1.01
LocalRead	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.00
	MATEpe	1.00	<0.99 - 1>	0.01	1.00	0.98	1.03
Send	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.01
	MATEpe	1.02	<0.98 - 1.06>	0.13	0.99	0.97	1.88
Activation	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.00
	MATEpe	0.99	<0.98 - 0.99>	0.01	0.98	0.97	1.02
Return	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.00
	MATEpe	0.97	<0.94 - 0.99>	0.01	0.97	0.94	0.99

Overhead when optimizing the meta level. The boxplots of Figure 8.4 indicates that there is no significant overhead in redefining a VM operation at the language level in terms of peak performance when applying our optimizations. Table 8.3 shows the results for each benchmark and VM. The overhead factor is negligible for all the operations.

To answer RQ2, we conclude from this experiment that for cases with low meta-variability there is no perceptible overhead in redefining a VM operation at the language-level. The considerably high overhead of the unoptimized versions (2-4 orders of magnitude) expose the positive impact of our optimizations.

8.3 Overheads When Extensively Using the MOP

We now evaluate more extensive usages of the MOP. To assess the peak performance overhead of creating and activating metaobjects at run time, we analyze particular cases of the read-only (REO) and profiling (PRO) experiments presented in Sections 7.2.2 and 7.2.3 respectively. The former evaluates a more in-depth usage of metaobjects attached to subjects while the latter attaches metaobjects to frames. Both install metaobjects dynamically.

8.3.1 REO Performance

We compare two implementations of read-only reference: one using handles and the other using delegation proxies. Section 7.2.2 already introduced the problem and showed our implementation of handles. The most important thing to recall is that the handles version use the MOP (VM-level reflective capabilities) while the delegation proxies version implements almost the same behavior using application-level reflective capabilities.

Also recall that we compare both approaches using a benchmark that creates a (read-only) reference to the head of a linked list and then traverses the list attempting to write some of its elements. It is worth noting that during the list traversal we need to wrap each access to the subsequent position of the list with a proxy or handle. As a consequence, the handles version introduce new metaobjects dynamically and delegation proxies new proxies. This implies the instantiation of considerably more objects than the baseline version and thus we expect a significant overhead of both approaches in comparison to the baseline.

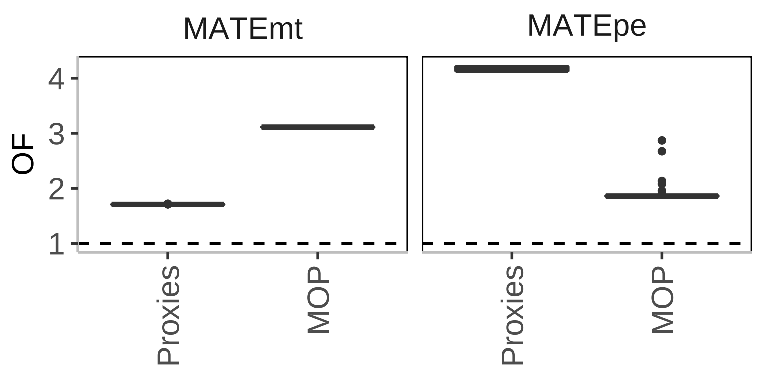

Figure 8.5: Overhead factors of implementing read-only reference using Handles with Mate (MOP) and using delegation proxies (Proxies).

Table 8.4: Overall Results for the Morphicness Benchmarks

Benchmark	VM	OF	Confidence	Sd	Median	Min	Max
Proxies	MATEmt	1.71	<1.71 - 1.71>	0.00	1.71	1.71	1.72
	MATEpe	4.17	<4.15 - 4.18>	0.04	4.14	4.12	4.24
MOP	MATEmt	3.11	<3.11 - 3.11>	0.00	3.11	3.11	3.13
	MATEpe	1.91	<1.85 - 1.97>	0.19	1.86	1.85	2.87

Overhead when optimizing the meta level. Figure 8.5 and Table 8.4 show the results of the optimized version. In MATE$_{PE}$, the MOP version outperforms delegation proxies with an overhead factor of 1.7 vs. 4.1. On the other hand, In

MATE$_{MT}$ delegation proxies outperform the MOP version with similar results, 1.7 for Proxies and 3.1 for the MOP. Overall, having one version with an overhead factor in the level of 1.7 for each benchmark suggests that both readonly references using the MOP and delegation proxies can achieve similar performance levels.

8.3.2 Tracing

In this experiment we assess the overhead of MATE when a large number of intercession handlers trigger the execution of metaobjects. To measure that behavior, we attach a metaobject to an execution context which propagates itself to the execution frame of subsequent method activations. Before, the metaobject increments a counter of method activations. Consequently, if we attach the metaobject to the beginning of a benchmark, we monitor all the methods executed by that benchmark. Furthermore, all the intercession handling sites for method activations within the code of the benchmark execute meta behavior.

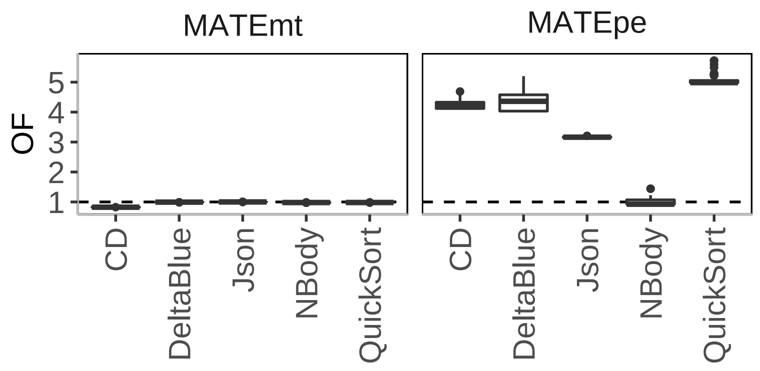

Figure 8.6: Overhead factors of profiling the amount of method activations in a set of benchmarks.

Table 8.5: Overall Results for the Profiling Benchmarks

Benchmark	VM	OF	Confidence	Sd	Median	Min	Max
CD	MATEpe	4.24	<4.09 - 4.4>	0.14	4.25	4.11	4.69
	MATEmt	0.82	<0.82 - 0.83>	0.00	0.82	0.82	0.83
DeltaBlue	MATEpe	4.37	<4.14 - 4.64>	0.30	4.36	4.01	5.20
	MATEmt	0.99	<0.99 - 0.99>	0.00	0.99	0.99	1.00
Json	MATEpe	3.16	<3.16 - 3.17>	0.02	3.16	3.13	3.21
	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.00
NBody	MATEpe	1.00	<0.95 - 1.05>	0.13	0.92	0.92	1.44
	MATEmt	0.98	<0.98 - 0.98>	0.00	0.98	0.98	0.98
QuickSort	MATEpe	5.08	<4.92 - 5.25>	0.20	4.98	4.98	5.72
	MATEmt	0.98	<0.98 - 0.98>	0.00	0.98	0.98	0.98

Overhead when optimizing the meta level. Figure 8.6 shows the results. It can be seen that MATE$_{MT}$ performs much better than MATE$_{PE}$. We assign this mainly to the fact that MATE$_{PE}$ is more sensitive to the inlining decisions and making the benchmarks bigger hinder several inlining opportunities. This may increase their running times because the cost of invoking a method is usually higher than inlining the callee. This goes deeper when method contains intercession handlers. For instance, we found that randomly increasing the inlining threshold of the graal compiler these benchmarks tend to significantly improve. What means there is still room for improvements in MATE$_{PE}$. Nevertheless, the overall results in MATE$_{MT}$ indicate that a reflective VM can run a comprehensive profiling with reasonably small overheads.

Table 8.5 shows the overhead factors. They are between 1.1 and 1.4 in MATE$_{MT}$ and between 1.4 and 4.5 in MATE$_{PE}$. Recall that this overhead includes the cost of the counting logic in all activations, including even activations on primitive types like integers and strings (*e.g.*, sum, multiplication, etc.).

To answer RQ3, our experiments suggest that the peak performance overhead when dynamically adapting applications can be reasonably low in reflective VMs, especially when compared with language-level approaches.

8.4 Breaking the Assumptions

We now assess how our reflective VMs perform when the assumptions of the optimization model do not hold. To do so, we design an ad-hoc micro benchmark that compares the cost of reading a field on a mono and megamorphic intercession handling site respectively. The benchmark instantiates a list of elements an reads a field of each element within a loop. Consequently, there exists only one field reading intercession handling site observing many objects of the same class. In the mono version all the observed objects has the same metaobject installed. For the mega version, we install a different metaobject (each performing the same behavior) to every instance forcing this way a megamorphic intercession handling site.

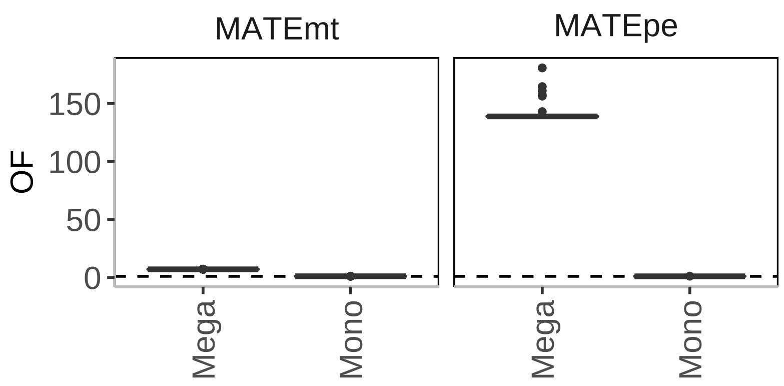

Figure 8.7: Overhead factor a monomorphic versus a megamorphic IH site.

Table 8.6: Overall Results for the Morphicness Benchmarks

Benchmark	VM	OF	Confidence	Sd	Median	Min	Max
Mega	MATEmt	7.05	<7.04 - 7.05>	0.03	7.03	7.03	7.12
	MATEpe	141.56	<138.83 - 144.31>	8.39	138.87	138.26	180.72
Mono	MATEmt	1.04	<1.04 - 1.05>	0.00	1.04	1.04	1.06
	MATEpe	1.00	<0.99 - 1.01>	0.03	0.99	0.96	1.14

Figure 8.7 and Table 8.6 shows the results. The monomorphic site exposes no significant overheads as expected in MATE$_{PE}$ while incurs in an overhead factor of 1.06 in MATE$_{MT}$. The only difference with the individual operation case (with zero overhead) is that the monomorphic site observes much more objects. This suggests that, in some cases, the tracing compiler may not be able to remove all

indirections when dealing with an intercession handling site that executes multiple subjects presenting all, the same metaobject. On the other hand, the megamorphic site is 6.47x slower in MATE$_{MT}$ while 165x slower in MATE$_{PE}$. This exposes that the the partial evaluation compiler is severely affected when the assumptions do not hold. The main problem is that graal is very sensitive to the inlining decisions. Breaking of the assumptions at run time jeopardize its inlining heuristics.

8.5 Warmup

To assess the warmup behavior MATE, we compare the first 100 iterations of all the benchmarks. Figure 8.8 shows the results for a subset of the benchmarks and in the appendix we present the plots for all the remaining ones. Each plot illustrates the overhead factor of each iteration independently, relative to the same iteration in the baseline version. There is a line with a ribbon for the overhead factor along with its corresponding 95% confidence interval for each benchmark/VM configuration.

The results show that each configuration of benchmark and VM has a particular warmup behavior in MATE$_{PE}$ (blue). For instance, for Havlak, FieldRead, Readonly, CDT, and Json MATE$_{PE}$ starts (or quickly goes) significantly below the overhead factor but after some iterations it is significantly above. Since the compilation task use significant processor resources, this suggest that the baseline finishes the profiling and starts the optimization task earlier. On the other hand, in cases such as CD, Bounce, All, NBody, JsonT, and Richards, MATE$_{PE}$ is above the overhead factor for several iterations. This suggests that the compilation starts at similar times and that the cost of running in interpreter mode for profiling run-time values incurs in significant overheads during these first iterations in MATE$_{PE}$. On the other hand, the results of MATE$_{MT}$ shows a more stable warming up behavior.

Finally, as expected, in all the cases (except All in MATE$_{MT}$ that has a very long warmup process) the lines tend to converge to the mean overhead. Table 8.7 shows the overall overhead factor clustered by the amount of initial iterations we take into account. For the inherent case we see that there is a minor overhead in the first iteration of MATE$_{PE}$ but not in MATE$_{MT}$. The individual operations micro benchmarks show very stable behaviors for both VMs. The readonly experiment is faster in the first iterations of MATE$_{PE}$ than the mean overhead factor of the original benchmark which is 1.7. This can be observed in the plot (the compilation overhead appears to happen after the 40th iteration). Surprisingly, MATE$_{MT}$ presents an overhead below its final overhead factor during the first iteration and then converges to its final overhead factor. Finally, for the tracing experiment the first iterations of MATE$_{PE}$ are significantly faster than the peak-performance overhead factor.

As a conclusion, and unlike what we would have expected, our empirical results shows that, at least for this set of benchmarks, the intercession handlers of a reflective VM does not increase significantly the warming up of the system in comparison to the overhead factor of the compiled code. In other words, the results suggest that when running the first iterations at interpretation level (for profiling the run-time values), the overhead of the intercession handlers is equal or less that the overhead after the compilation is done.

8.6 Reflective Compilation

In Chapter 6 we proposed a compilation MOP for enabling developers to improve the performance of their applications whenever the compilation heuristics do not reach the expected performance. We now present a preliminary evaluation of this

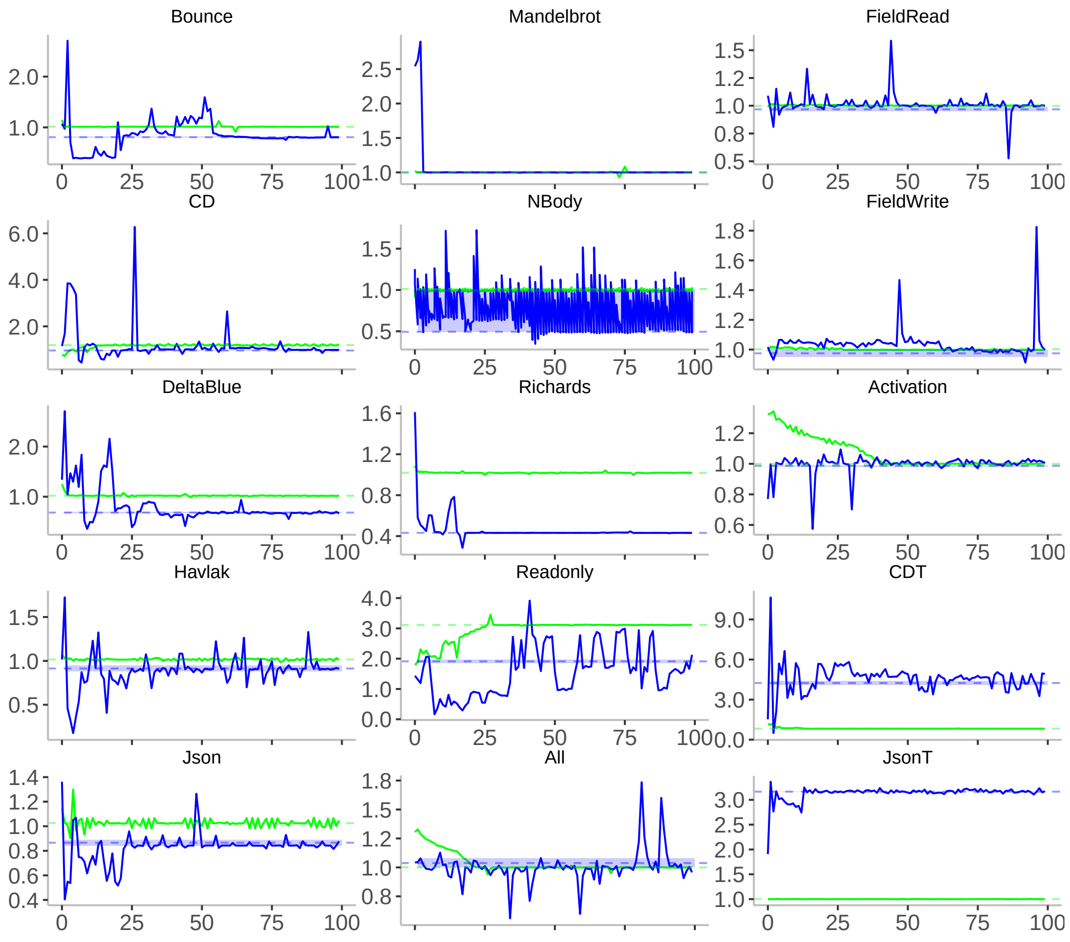

Figure 8.8: The first 100 iterations of a subset of the benchmarks in MATEpe and MATEmt normalized (per iteration) to their corresponding SOM versions.

Table 8.7: Overall warmup overhead factor until n iterations clustered by benchmarks

Iterations	VM	OF	CI-95%	Sd.	Min	Max	Median
Inherent							
MATEpe	1	1.33	<1.05 - 1.61>	0.485135395121758	0.80	2.54	1.20
MATEmt	1	1.08	<1 - 1.16>	0.139495583269205	0.78	1.37	1.07
MATEpe	5	1.24	<0.94 - 1.55>	0.530604839387995	0.24	2.67	1.14
MATEmt	5	1.03	<0.98 - 1.07>	0.0804070922750607	0.82	1.17	1.02
MATEpe	20	1.11	<0.82 - 1.4>	0.498211716597319	0.24	2.52	0.99
MATEmt	20	1.01	<0.99 - 1.03>	0.0334474451569285	0.96	1.08	1.01
MATEpe	40	1.04	<0.78 - 1.31>	0.457970581885389	0.30	2.36	0.93
MATEmt	40	1.01	<1 - 1.03>	0.0273053568031064	0.96	1.06	1.01
Individual							
MATEpe	1	0.94	<0.83 - 1.04>	0.127490160074401	0.77	1.09	0.93
MATEmt	1	1.07	<0.93 - 1.21>	0.16541789283935	0.87	1.33	1.00
MATEpe	5	0.93	<0.86 - 1>	0.0849749670791935	0.82	1.05	0.93
MATEmt	5	1.09	<0.97 - 1.21>	0.141548679660104	0.97	1.32	1.01
MATEpe	20	0.93	<0.86 - 1>	0.0836030236106841	0.82	1.04	0.94
MATEmt	20	1.07	<0.99 - 1.16>	0.0998279597197901	1.00	1.24	1.01
MATEpe	40	0.93	<0.86 - 1>	0.0810962834277958	0.82	1.03	0.94
MATEmt	40	1.05	<0.99 - 1.1>	0.0674241516826273	1.00	1.19	1.01
Readonly							
MATEpe	1	1.43	-	-	2093.81	2093.81	2093.81
MATEmt	1	1.79	-	-	89.70	89.70	89.70
MATEpe	5	1.45	-	801.01	251.66	2093.81	283.28
MATEmt	5	1.95	<0.31 - 43.53>	29.65	19.58	89.70	27.80
MATEpe	20	1.31	-	466.97	7.22	2093.81	12.56
MATEmt	20	2.16	<1.21 - 4.22>	16.59	15.55	89.70	15.71
MATEpe	40	1.29	<-0.12 - 45.62>	337.68	7.17	2093.81	11.60
MATEmt	40	2.48	<1.74 - 3.73>	11.91	15.45	89.70	15.56
Tracing							
MATEpe	1	1.56	<1.14 - 1.98>	0.335591120510194	1.07	1.91	1.53
MATEmt	1	1.00	<0.87 - 1.12>	0.100922495920333	0.92	1.17	0.96
MATEpe	5	2.49	<0.64 - 4.35>	1.49222520465305	1.07	4.99	1.93
MATEmt	5	1.00	<0.91 - 1.09>	0.0723605221626415	0.94	1.12	0.97
MATEpe	20	2.75	<0.56 - 4.94>	1.76343674327522	0.99	5.67	2.19
MATEmt	20	0.98	<0.96 - 0.99>	0.0149860465052862	0.96	1.00	0.97
MATEpe	40	2.83	<0.77 - 4.89>	1.65776566946782	1.00	5.49	2.50
MATEmt	40	0.97	<0.92 - 1.01>	0.0379363559151132	0.90	1.00	0.98

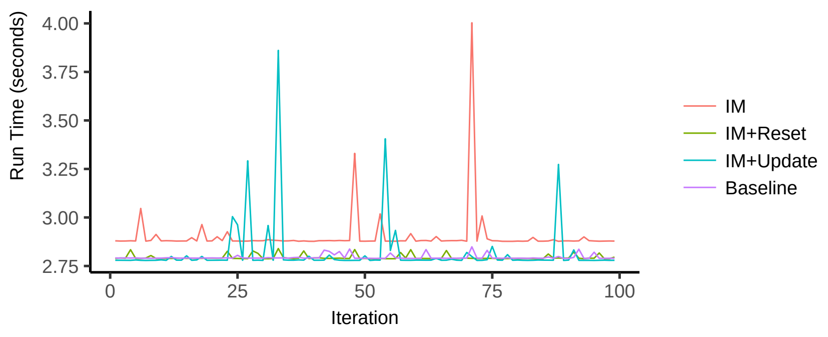

Figure 8.9: Peak performance execution time for the instance migration micro benchmark with ephemeral variability.

proposal. To do so, we implemented almost all the application from compiler directives of the API presented in Section 6.2 in MATE$_{PE}$. Concretely, we added reflective capabilities for invalidating compiled code, forcing the splitting of methods, and customizing dispatch chains. Below we show running time reductions for two scenarios exhibiting application-specific variability. In section 8.7 we provide some preliminary conclusions and in chapter 10 we discuss some future directions on these ideas.

8.6.1 Ephemeral Variability

We used as baseline the `gatherX` method presented in Section 6.1 and execute three variations with subtle differences:

1. Instance Migration (IM): One of the points in the list is wrapped by a proxy as introduced in the motivation example.
2. IM+Reset: Same as IM, but after the proxy initialization, we use the compilation API to find the field reading node of the `x` field, and reset its dispatch chain:

```
Example>>#InstanceMigrationReset
  | ast node |
  ast ← (Baseline>>#gatherX) compilation.
  node ← (ast fieldReadsWithName: 'x') first.
  ↑node dispatchChain reset.
```

3. IM+Update: Same to the previous case but instead of resetting the dispatch chain we just remove the specialization generated by the proxy instance.

Results. Confirming our hypothesis, Figure 8.9 shows IM is slower than the baseline after stabilization: warm up + compilation. Furthermore, both, IM+Reset and IM+Update show performance boosts in comparison to IM. Their running times are similar to the baseline. The mean overall results for the iterations of each benchmark are: Baseline 2795 ms, IM 2903 ms, IM+Update 2819 ms, IM+Reset 2794 ms.

8.6.2 Highly Indirect Variability

Let us now consider the call to `Vector»collect:` in the `gatherX` method from the previous example. The callee is part of the MATE$_{PE}$ standard library. MATE$_{PE}$ speculates on the block (closure) received as parameter for optimizing its dispatching and enabling the JIT compiler to inline it. To illustrate a highly indirect scenario, suppose now a subtle difference with the previous example: we need to collect both the `x` and `y` values from the vector:

```
Example>>#gatherAndProcessXandY
  | xValues yValues |
  xValues ← points collect: [:point | point x].
  yValues ← points collect: [:point | point y].
  self process: xValues and: yValues.
```

This example presents highly indirect variability because it calls `collect:` twice within the same method (context) but using two different blocks. Since the context is the same, the (Graal) compiler heuristics avoid splitting `collect:`.

Benchmarks. We ran the previous example in two different flavors: *Indirect* runs exactly the example while *Indirect+Split* forces the splitting of the second call to collect using the compilation API:

```
Example>>#splitCollect
  | ast send callSite |
  ast ← (Example>>#gatherAndProcessXandY) compilation.
  send ← (ast messageWithSelector: 'collect:') second.
  callSite ← send dispatchChain firstSpecialization.
  ↑callSite split.
```

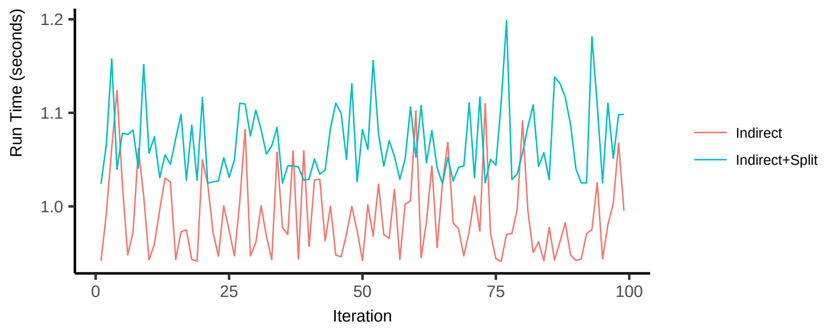

Figure 8.10: Peak performance execution time for the splitCollect micro benchmark with highly indirect variability.

Results. Figure 8.10 shows a reduction in the execution time of Indirect+Split. The mean overall results for the iterations of each benchmark are: Indirect 989 ms and Indirect+Split 1068 ms, resulting in a performance gain of about 7%. In the case of critical methods, this simple fine-tuning at run time appears worth trying.

8.7 Discussion

Our empirical evaluation showed that the proposed optimizations remove most of the indirections introduced by MATE's intercession handling sites. Also that a dynamic compiler cannot remove all those indirections without the proper optimization techniques. The experiments also illustrate that a high local meta-variability leads to a severe degradation of the performance, implying that the flexibility of the MOP comes with a price and must be used with care. Nevertheless, it is worth mentioning that the experiments with megamorphic intercession handling sites are purely synthetic. Currently, we do not anticipate concrete use cases using the MOP to this extent. As discussed in Section 5.2, we assume low local meta-variability as the most common scenario. Summarizing, the experiments up to Section 8.6 support that reflective VMs can run efficiently (Hypothesis 2).

All these performance results were achieved by using two general purpose compilers for different approaches to self-optimizing interpreters: partial evaluation and meta-tracing. The comparison of the two techniques in the context of reflective VMs raised similar conclusion to what Marr et al. [MD15] had already pointed out for reflective systems: they reach very similar peak-performance results. Moreover, for the cases where a scenario did not perform well with one implementation, this approach allowed us to suggest that the limitation was accidental and not fundamental by showing its proper behavior in the other implementation.

It is worth noting that the partial evaluation approach requires much more effort by the language implementers to reach similar levels of peak-performance. For instance, precise branch profilers must be installed in the system to avoid compiling branches of a method that are never reached but may jeopardize relevant optimizations. Furthermore, the tracing experiment suggest that partial evaluation is very sensitive to the inlining heuristics which may need a fine-tuning when the metalevel is being used. This kind of drawbacks were not exposed by the tracing compiler.

Based on the fact that there are still cases presenting small overheads without fundamental grounds, our intuition is that an ad hoc compiler with a full knowledge of the meta model would be able to remove even more indirections. For instance, by enabling a fine-tuning of the inlining budgets or the dispatch chain lengths. However, engineering or customizing an efficient compiler is a resource demanding task and out of the scope of this thesis.

On the other hand, results in Section 8.6 indicate that a reflective compiler could improve the obtained performance of a general-purpose JIT compiler by leveraging developers knowledge of the application to fine-tune the optimizations. This partially supports Hypothesis 3. A more comprehensive validation is still needed to better understand the best applicable scenarios and the limits and drawbacks of the approach.

The selection of benchmarks for the performance results may lead to conclusions that do not generalize to other programs. To mitigate this threat we selected a set of benchmarks included in the literature of the field [MDM16]. We also evaluated the MOP in use cases appearing in previous works (read-only [ADD+10], tracing [RHV+09]). To ensure a fair comparison for the readonly experiment we implemented Delegation Proxies following the guidelines of the original paper [WNTD14]. However, it is worth noting that the field of reflective VMs is yet unexplored and the work on applications that harness the capabilities of reflective VMs has just started.

Our main focus was to demonstrate that reflective VMs could run efficiently. Accordingly, our evaluation focused only on measuring peak performance. Considering memory consumption, all the dispatch chains per intercession handling site consume extra memory. Nevertheless, since the length of dispatch chains is typically bounded to avoid the overhead of long linear searches, they introduce only a memory overhead with an upper bound per intercession handling site. On the other hand, the incorporation of metaobjects governing the semantics of every instance also has an impact on memory consumption. Our optimization for blending metaobjects with object layouts along with our assumption of low metavariability significantly mitigate this overhead.

Regarding the compiler reflective capabilities, it is worth noting that our approach assumes developers have an advanced knowledge of both, the application specifics and compiler optimizations for dynamic languages. We expect the API to be used in cases where the application-specific variability is known *a priori* and is hindering optimizations. On the negative side, developers might not use the compiler API properly. Wrong hints and directives given to the optimizer may lead to potential performance penalties. More comprehensive use cases stressing the usage of the MOP are needed to analyze its eventual negative impact.

8.8 Resumen en Castellano

En este capítulo validamos H2 y H3, dos hipótesis sobre el desempeño de las MVs reflexivas. Para esto realizamos una evaluación empírica del desempeño de MATE$_{PE}$ y MATE$_{MT}$. En primer lugar, utilizamos una serie de benchmarks para la evaluación de MVs y comparamos los tiempos que tardaron en ejecutar en nuestras MVs reflexivas contra los tiempos que tardaron en MVs estándar. De esta manera evaluamos la efectividad de nuestro modelo de optimización para los casos en los que las capacidades reflexivas de la MV no son utilizados. Luego, evaluamos los costos en términos de desempeño al redefinir operaciones individuales, tal como la activación de métodos, desde la aplicación. Además, evaluamos los impactos en el desempeño al usar el MOP de manera exhaustiva en casos como REO y OPT. Finalmente, exponemos dos casos en los cuales usando las capacidades reflexivas del compilador logramos mejorar el desempeño de aplicaciones con variabilidad específica.

Para contemplar el no determinismo de los sistemas modernos y el impacto de las optimizaciones adaptativas en las diferentes implementaciones, los resultados que presentamos se basan en 50 muestras luego de que el compilador dinámico produce las optimizaciónes primordiales. Para identifiar este momento, tomamos 1500 muestras y luego ejecutamos un análisis de puntos de cambio siguiendo la metodología propuesta por Barrett y otros [BBTK+17]. En la sección 8.5 discutimos los impactos concretos de los tiempos de inicialización de las MVs reflexivas a partir de los valores de estos tiempos en MATE$_{PE}$ y MATE$_{MT}$.

Con el objetivo de validar H2 de manera exhaustiva nos enfocamos en cuatro preguntas, cada una inquiriendo sobre un aspecto distinto del desempeño de las MVs reflexivas. La primera analiza las penalidades inherentes de este tipo de MVs, o sea las penalidades que existen aunque no se utilicen sus capacidades reflexivas. La segunda, evalúa las penalidades de redefinir operaciones de la MV de manera individual. La tercera inquiere sobre las penalidades de adaptar aplicaciones dinámicamente usando las capacidades reflexivas del nivel de la MV. La última pregunta persigue el objetivo de entender los impactos de las capacidades reflexivas a nivel de MV en los tiempos de inicialización.

En el capítulo se detallan los benchmarks que utilizamos para cada caso. Para el caso inherente utilizamos el conjunto de benchmarks propuesto por Marr y otros [MDM16] para comparar diferentes implementaciones de lenguajes de programación dinámicos. A este conjunto le agregamos algunos benchmarks específicos para tener un mayor cubrimiento de aspectos tales como la ejecución de ciclos, el uso de arrays o la randomización de valores. En general, consideramos que este conjunto de benchmarks captura un abanico extensivo de comportamientos que permiten entender las penalidades inherentes de una MV reflexiva.

Para las operaciones individuales desarrollamos benchmarks ad-hoc que simplemente ejecutan la operación en cuestión. En la versión para la MV reflexiva, esta operación es redefinida a nivel del lenguaje usando el MOP. La redefinición en MATE implementa exactamente el mismo comportamiento que la versión original. Para evaluar el uso extensivo del MOP, analizamos las penalidades en diversas versiones de REO y PRO. Finalmente, también diseñamos dos benchmarks para estudiar el desempeño cuando las presunciones del modelo de optimización no se cumplen. Esencialmente, simulamos un sitio de IH monomórfico en un benchmark y uno megamórfico en el otro.

Las conclusiones exhiben que las optimizaciones que propusimos eliminan la mayor parte de las indirecciones que conciernen al manejador de intercesión de MATE. También que los compiladores dinámicos de proposito general no son capaces de remover muchas de estas indirecciones sin mecanismos de optimización especialmente diseñados para el meta nivel. Los experimentos también ilustraron que cuando la meta-variabilidad es considerable se producen degradaciones severas del desempeño, sugiriendo que la flexibilidad del MOP esta asociada a un costo, y que por ende debe ser usado con atención. De todos modos, es importante mencionar que los experimentos con meta-variabilidad que expusimos son puramente sintéticos. En resumen, los experimentos hasta la sección 8.6 respaldan que las MV reflexivas pueden correr de manera eficiente (Hipótesis 2).

Por otro lado, los resultados de la sección 8.6 indican que las capacidades reflexivas a nivel de compilación pueden mejorar el desempeño de los compiladores dinámicos mediante el aprovechamiento del conocimiento de los desarrolladores sobre la variabilidad específica de sus programas. Los experimentos de esta sección respaldan parcialmente la hipótesis 3. Sin embargo, una evaluación más exhaustiva es necesaria para una mejor comprensión de los escenarios en que estas técnicas son aplicables.

Part IV

Discussion

CHAPTER 9

Related Work

In this chapter we discuss related work.

9.1 Reflective Solutions

Pinocchio first class interpreter [VBG$^+$10] is a practical implementation, in the context of an OO language, of Smith's tower of interpreters [Smi84]. The interpreter is first-class and extensible from language level. In contrast to MATE, Pinocchio does not impose a fixed number of metalevels for dealing with eventual metaregressions. It adapts to different levels on demand. On the other hand, Pinocchio is a reflective interpreter while a reflective VM covers more low-level features. For instance, Pinocchio is not able to deal with the structural case studies of section 7.3 because it does not reify object layouts. Similar to Pinocchio, Asai [Asa11] proposes a first-class interpreter but in the context of a functional language. It shares with Pinocchio the same fundamental differences with MATE.

CLOS [KR91] is an object-oriented layer for LISP that implements an advanced MOP, regarded as one of the most complete in terms of introspection and intercession reflective capabilities. CLOS reifies *Slots*, a language level representation of instance variables (fields). It also provides means to customize methods with generic functions, method combinators, and before/after methods. Since CLOS' main goal is enabling language customizations rather than being a reflective VM, it does not include extensive reflective capabilities for low-level functionalities such as the complete operational semantics of the language. According to our understanding, using CLOS it may not be possible to handle the read-only references scenario of section 7.2.2 in a transparently because of its limitations for interceding the method lookup and activation on individual objects. CLOS is also not able to approach a direct adaptation of method's local/argument accesses.

CodA proposes a metamodel that decomposes execution concepts into different roles. This decomposition is independent of the concrete programming language [McA95]. It is richer than MATE's MOP when considering the *message* metaobject because it reifies the receiver and sender roles separately and the synchronization of message activations. On the other hand, CodA reifications are similar to those of CLOS, more related to language-level concepts than to the VM perspective. For instance, CodA does not reify object layouts nor execution aspects such as method return values, or even the execution stack.

Flexible Object Layouts [VBLN11] reifies the internal structure of objects. Its main reification is the *Slot*, similar to slots in CLOS. Slots can be extended at run time by redefining four main operations: read, write, initialize and migrate. We followed a similar approach for implementing the `Layout` metaclass in MATE, just leaving out the migration operation which was not needed in our case studies.

9.2 Virtual Machines

Several self-hosted approaches for VM construction support some forms of VM-level reflection. Klein [USA05] for Self has similar goals to ours but its support for modifying VM-level entities at run time is not explained in the literature. The paper only mentions support for mirror-based debugging tools to inspect and modify a remote VM.

Tachyon [CBLFD11] translates the VM sources written in JavaScript to native code. Then, it uses special bridges for interacting with low-level entities of the VM. However, bridges are low-level mechanisms that only allow to call remote functions. Tachyon uses them to initialize a new VM during the bootstrapping process. In contrast to reflective VMs, Tachyon was not designed with VM-level reflection as a goal and it does not provide adavanced run-time adaptation capabilities of VM-level entities.

Maxine [WHVDV$^+$13] for Java, uses abstract and high-level representations of VM-level concepts and consistently exposes them throughout the development process. Development tools like inspectors provide a live interaction with the running VM at multiple abstraction levels while debugging. However, Maxine allows to inspect but not to modify the VM at run-time. Similarly, in the JikesRVM [AAB$^+$05] VM components can be inspected but not modified at run-time. Reflection on VM components is mainly used for the bootstrapping of the system. In contrast, a reflective VM focuses on providing interactivity during run time.

9.3 Dynamic Adaptations

To the best of our knowledge, Partial Behavioral Reflection (PBR) [TNCC03] is the most complete reflective solution for supporting unanticipated adaptations. PBR relies on bytecode instrumentation. Hence, it is restricted to adapting operational semantics. In addition, instrumentation techniques modify the application code and, from a VM perspective, the original code becomes indistinguishable from the instrumented code. In contrast, MATE fulfills the adaptations by using reified VM-level components and does not modify the application code. Concretely, MATE focuses on VM-level reflection while PBR depends on application-level reflection for (simulating) the low-level adaptations.

The Iguana/J environment [RC02] has similar characteristics to PBR. However, Iguana/J provides these capabilities with a MOP similar to ours in terms of behavioral adaptive capabilities. Similar to CLOS, Iguana/J provides intercession handlers for method interceptions, reading, and writing of fields. MATE allows to intercept a broader set of operations such as the whole operational semantics of the system and provides structural VM-level reflective capabilities.

Cazzola et al.'s [CCRS17] recently propose to move the evolution from the application level to the programming language level to support direct dynamic adaptations. They use Neverlang [VC15], a micro-language framework for modular language development. Their goals are similar to ours: separation of concerns, maintainability, and reuse of adaptations. However, in contrast to a reflective VM their adaptation abstractions are at the programming language and not the VM level. On the one hand, the approach enables adaptations that a reflective VM does not provide. This includes the direct customization of arithmetic operations or control flow operators, such as loop, even for single instances On the other hand, it does not enable developers to directly customize object layouts, the execution stack, the memory organization, as well as other VM-level aspect. Accordingly, the approaches are conceptually complementary. Moreover, since Neverlang is based on micro operations described using grammars, grammars are also the means to provide the adaptations. In contrast, a reflective VM is based on MOPs and stays within the application's programming language syntax and semantics.

Finally, recall that in Sections 3.2.4 and 3.2.5 we already analyzed aspect-oriented and context-oriented programming paradigms.

9.4 Efficient Run-time Adaptability

Based on Self-Optimizing Interpreters. Marr et al. [MSD15] showed that dynamic compilation and dispatch chains can drastically reduce the run-time overhead of reflective operations for language level concepts. They showed that speculating on stable behavior for intercession handlers of field accesses or method invocations provides the compiler with enough information to optimize the operation to a normal base-level access or invocation. Although we use the same fundamental technique (dispatch chains) for caching meta-level information, our work has significant differences with theirs: 1) MATE aims at providing reflection at VM level, *i.e.*, targeting low-level operations of VM components instead of including only language-level operations 2) Marr et al.'s MOP considers only field accesses and method execution. Mate's MOP in addition enables to customize object layout, stack frames and compilations aspects; separates intercession handling for method lookup and activation; and provides fine-grained intercession handlers for variable and argument access. This results in a much higher degree of flexibility and consequently a much higher number of intercession handling sites exacerbating any inefficiencies in the runtime system 3) we designed and described a new optimization model targeted to our particular intercession handling that combines object layouts and meta-level semantics to minimize run-time cost 4) our evaluation also considers more comprehensive usages of the MOP. For instance, we evaluated dynamic adaptation scenarios of more realistic applications to support the claims that the ubiquitous intercession handling sites can be optimized effectively.

Seaton et al. [SVDVH14] implement a native debugger for a Truffle-based language and optimizes out its overhead when it is not activated. This is achieved by wrapping operations with a kind of intercession handler that checks if debugging for the current operation is activated. We use a similar approach for wrapping MATE's base-level nodes. In contrast to MATE, Seaton et al.'s intercession handling is simpler because it does not depend on metaobjects and is fixed at compile time. Thus, they do not need to handle the meta-variability problem.

General Approaches. Several recent works delegate the burden of optimizing performance to the developer. Such approaches provide language abstractions for communicating hints to the compilers so that they can optimize execution. The main idea is that programmers indicate that certain expressions are to be evaluated at compile time so that they can be precomputed via partial evaluation.

For instance, Shali and Cook [SC11] proposed the notion of hybrid partial evaluation, which is a combination of partial evaluation and compile-time metaprogramming. DeVito et al. [DRF+14] proposed Exotypes which are similar in spirit. Their idea is to give programmers language abstractions to use staged programming, which allows the generation of specialized implementations for a given problem such as serialization at runtime. Their implementation is then able to use the known types to generate efficient code that can outperforms similar custom implementations. Finally, Asai [Asa14] explores the usage of staged programming for improving the performance of a tower of meta interpreters with a powerful metaobject protocol. However, the staging approach limits the expressiveness of the meta interpreters and the ability to change language behavior is restricted under compilation.

While these approaches leads to major performance benefits, the burden is put on programmers. In contrast, MATE only requires developers to manually provide hints when a fine-tuning of performance is required. To do so, Mate provides a compilation API. However, most of the indirections caused by the intercession handlers are automatically removed by the optimization model. Moreover, these approaches usually forbid the use of compile-time objects at run time, which might be a strong restriction for some use cases. In contrast, while a high meta-variability can lead to slower execution in MATE, expressiveness is not restricted by the system. It integrates with JIT compilation and dynamic code invalidation to support dynamic behaviors.

9.5 Mitigating Application-Specific Variability

Hölzle and Ungar [HU94] proposed that method activation counters represent invocation rates instead of counts. They suggested counters that decay exponentially to avoid the compilation overhead for methods that are not performance critical. Furthermore, they proposed to dynamically adapt the decay rate depending on the stability of the system. This would mitigate problems with transient variability such as warm up, rare, and ephemeral variability. However, this heuristic does not apply to all issues we identified. Concretely, it would fail whenever an application phase containing transient behavior executes the corresponding method frequently enough. Our approach complements compilation heuristics for the cases when they are not enough, by giving developers the opportunity to amend those scenarios using a comprehensive language-level API.

Recently, Rompf et al. presented the Lancet JIT compiler framework [RSB+14] for Java bytecode that enables programs to control several aspects of the JIT compilation process. Lancet features hooks that enable to trigger a predefined set of macros whenever a method is going to be compiled. On top of these macros, Lancet features an API allowing developers to *annotate* methods with compilation directives such as compile, unroll, or freeze (partial evaluate at compile time). We share the same vision and goals noticing a potential in connecting a JIT compiler with the application it optimizes. However there are differences in the approaches. While Lancet's API is mainly used at compile time, our API was designed to enable the inspection and modification of the compilation aspects of a method at run time. In addition, we focused on reifying aspects regarding speculative optimizations such as dispatch chains and splitting, while Lancet focused on other aspects such as partial evaluation and inlining. Lastly, Lancet advocates for an integration of the compilation directives within the source code, while we promote the modularization of the optimization aspects by the use of a MOP.

9.6 Resumen en Castellano

Este capítulo discute el trabajo relacionado. Comienza con las soluciones reflexivas. Pinocchio [VBG+10] es una implementación práctica de la torre de intérpretes presentada por Smith y otros [Smi84]. Similar a Pinocchio, Asai [Asa11] propone una torre de intérpretes pero para un lenguaje funcional. CLOS [KR91] es una capa orientada a objetos para LISP que implementa un MOP, considerado uno de los más completos en términos de capacidades reflexivas. CodA propone un metamodelo que descompone la ejecución de conceptos en diversos roles. La descomposición es independiente del lenguaje de programación [McA95]. MATE presenta diferencias significativas respecto a todas estas altenativas. A modo de síntesis, dado que MATE es el único que abarca los diversos conceptos que involucran a una MV, es el único también en poder abordar todos los escenarios de adaptación introducidos en esta tesis de manera directa.

Se continúa con la descripción de MVs relacionadas. Sobre todo, MVs desarrolladas en lenguajes de alto nivel que promuevan algún tipo de flexibilidad en tiempo de ejecución. Se mencionan Klein [USA05] para el lenguaje Self, Tachyon [CBLFD11] para JavaScript, y Maxine [WHVDV+13] y JikesRVM [AAB+05] para Java. La única con objetivos similares es Klein. Sin embargo la literatura y los prototipos existentes no exhiben mecanismos claros de adaptación en tiempo de ejecución de ninguno de sus componentes. Las soluciones restantes permiten, como máximo, realizar computos reflexivos instrospectivos pero no la customización del comportamiento de la MV en tiempo de ejecución.

Luego se presentan las soluciones existentes a nivel de lenguaje que permited adaptar software dinámicamente. Se menciona Partial Behavioral Reflection (PBR) [TNCC03], como el framework más completo para afrontar adaptaciones no anticipadas. También Iguana/J [RC02] con similares características. Por último la solución reciente propuesta por Cazzola y otros [CCRS17] para transladar la evolución de los sistemas desde el nivel de la aplicación al nivel del lenguaje de programación. Todas las soluciones presentan alguna limitación para abordar alguno de los casos de estudio. Además las MV reflexivas presentan diferencias en cuanto al nivel de abstracción en el cual se expresan las adaptaciones lo cual redunda en diversas ventajas respecto a la mantenibilidad, modularidad, y el desempeño de las adaptaciones.

Respecto al desempeño de los sistemas reflexivos, Marr y otros [MSD15] mostraron que los compiladores dinámicos junto con las *cadenas de despacho* pueden reducir drásticamente las penalidades de desempeño de este tipo de sistemas. Aunque utilizamos las mismas técnicas fundamentales en este trabajo, las MV reflexivas exhiben diferencias significativas con los sistemas estudiados por estos autores. Por otro lado Seaton y otros [SVDVH14] también implementaron un depurador eficiente usando técnicas similares. Sin embargo, el manejador de intercesiones del enfoque es mucho más simple que el de una VM reflexiva ya que no debe lidiar con los problemas de meta-variabilidad.

Por otro lado, varios trabajos delegan la responsabilidad de optimizar el sistema a los desarrolladores proveyendo indicios al compilador sobre los lugares en donde se puede realizar una evaluación parcial. Por ejemplo, Shali y Cook [SC11] propusieron una combinación de evaluación parcial y metaprogramación en tiempo de compilación. DeVito y otros [DRF+14] propusieron la idea de proveer abstracciones para usar programación por etapas lo cual permite especializar el código generado para ciertos problemas particulares. Por su parte Asai [Asa14] exploró la el uso de la programación por etapas para mejorar el desempeño de la torre de intérpretes. Aunque todas estas soluciones presentan beneficios en terminos de desempeño, a diferencia de MATE que optimiza los manejadores de intercesión de manera automática, estas soluciones ponen la carga de las optimizaciones sobre los desarrolladores.

Finalmente el capítulo analiza los trabajos relacionados con la compilación reflexiva. Hölzle y Ungar [HU94] propusieron agregar contadores de activación a los métodos para representar tasas de invocación. También propusieron adaptar dinámicamente el umbral a partir del cual estos contadores disparan la compilación dinámica para resolver problemas de variabilidad efímera o de tiempo de inicialización. A diferencia de nuestro enfoque, el umbral sigue siendo un heurística que, por definición, no cubre todos los casos.

Más recientemente, Rompf y otros presentaron el compilador dinámico Lancet [RSB+14] el cual permite controlar varios aspectos del proceso de compilación desde las aplicaciones. Nos inspiramos en este trabajo para desarrollar nuestra API de compilación y perseguimos objetivos similares. Sin embargo, Lancet está enfocado en ofrecer herramientas para mejorar el programa en tiempo de compilación mientras que nuestro enfoque promueve las mejoras de desempeño en tiempo de ejecución.

CHAPTER 10

Conclusions

We introduced the notion of fully reflective execution environments and discussed the main properties they should fulfill. In order to validate the feasibility and understand the overall potential of reflective VMs, we designed and implemented MATE$_{PE}$ and MATE$_{MT}$: two fully functional Smalltalk VMs featuring a dynamic compiler and advanced reflective capabilities at the VM level. Both enable introspection and intercession of behavioral and structural VM concepts. These are: several operations of the interpreter, the method lookup mechanism, the layout of objects and the execution stack. The degree of reflective capabilities reached by MATE$_{PE}$ and MATE$_{MT}$ is already a good indicator for the feasibility of VMs with extensive reflective capabilities.

By performing an empirical evaluation we assessed the potential of reflective VMs for handling *unanticipated fine-grained adaptation* scenarios on the fly. We contrasted our solutions against alternative approaches considering a wide range of qualitative aspects. The results showed that reflective VMs successfully handle a series of heterogeneous adaptation scenarios covering structural and behavioral low-level adaptive requirements. In all the cases, the solution consisted of few lines of code (between 20-45) and presented benefits over the existing alternatives. Furthermore, we are not aware of any platform capable of dealing with all the scenarios like MATE$_{PE}$ and MATE$_{MT}$ did. Concluding, we provide evidence showing that reflective VMs are a suitable alternative for dealing with flexible software.

We also provided an analysis sustaining why, from our perspective, contemporary reflective systems, aspect-oriented programming, and context-oriented programming present fundamental limitations for handling software adaption in general. Concretely, we showcased the limitations of these paradigms for approaching the adaptive scenarios our reflective VMs were able to properly approach. The main reason for their limitation is that they are not enable to express direct adaptations for a wide-range of low-level entities. As a consequence, we conjecture that reflective VMs are a more suitable foundation for developing flexible software than other language-level approaches. Although this needs a more exhaustive proof, we already provided an elaborated discussion about why reflective VMs conceptually subsume the adaptive capabilities of the three aforementioned alternatives.

This work also shows evidence supporting that a reflective VM can run programs efficiently. This contradicts the belief that reflection at the VM level is impractical and opens the door to further research in this area. The main optimizations applied by MATE$_{PE}$ and MATE$_{MT}$ are the speculation on low local meta-variability and the combination of the meta behavior with object layouts. Our performance benchmarks showed that when the MOP is not activated, or when it only redefines individual VM operations, the mean peak performance overhead is neglibile in most of the cases. Furthermore, the overhead is at most 5x when dealing with elaborated on the fly adaptations. Note that reflective VMs will be a suitable alternative for deploying applications only if they do not incur in excessive overheads.

Regarding more advanced usages of the MOP, we also presented a series of application-specific variabilities, namely: ephemeral, warm up, rare, and highly indirect. We showed how they may challenge state-of-the-art JIT compilers, leading to performance overheads. These overheads could be mitigated if application developers had the means to supply the compiler with additional information and improve over heuristics. Therefore, we proposed to exploit the compilation capabilities of a reflective VM. We presented an API covering several aspects of the compilation process, implemented a subset in MATE$_{PE}$, and obtained significant overhead reductions in a couple of preliminary scenarios including application-specific variability.

We believe that our work opens the door to a broader an deeper study of the fundamental limitations of advanced reflective systems in general. Traditionally, reflective capabilities have been considered slow, insecure, or unnecessary. That is the reason why mainstream software platforms provide them only in limited ways. This work discusses that belief by providing empirical evidence showing that advanced reflective capabilities are useful and can be provided without considerably affecting the performance of the overall system. At least, for our case studies.

Summarizing, we were able to validate (some of them partially) our hypotheses about the potential of reflective VMs to support software evolution (H1), the possibility of making them efficient in terms of performance (H2), and to show other applications of reflective VMs like fine-tune optimizations on the fly (H3).

10.1 Future Work

With these results as a foundation, future work could explore how the MATE approach could be applied to other low-level VM components. For instance, our MATE implementations do not reify the `Memory` component and, thus, we can neither directly adapt the memory management nor the garbage collector. We plan to explore the impacts of providing reflective capabilities to this component. We also plan to provide more extensive reflective capabilities to the `Executor` (*e.g.*, to support a trace-based execution schema that could provide history-aware semantics to the applications).

We also plan to further refine, extend, and implement the compilation reflective API. We conjecture that the capabilities provided by a reflective compiler could also further improve the performance of reflective VMs. For instance to improve warmup times. On the other hand, in performance critical parts, a user could explicitly disable or lower the degree of dynamicity supported by a MOP. In the context of a particular paradigm, such as context-oriented programming, the necessity of a reflective compiler has already been suggested [PFH16].

It is worth noting however that the compiler community usually think that giving this power to developers has several problems: it is not secure, it would harm performance in other places, and it does not scale [RSB^{+}14]. An alternative to improve the effectiveness of JIT compilers under these scenarios may be to apply artificial intelligence techniques so that compilers learn and improve their performance automatically. Clearly, more research is needed to understand the better alternative to this problem and we plan to conduct it.

In general, our long-term goal is understanding the fundamental limits of VM-level reflection, both in terms of feasibility and performance impacts. We want to address questions like which is the minimal core of a reflective VM that cannot be reifed or (more practically) what are the consequences of reifying each VM component/operation. As long as we incorporate more reflective capabilities we expect to encounter new challenges like stronger causal connections and new performance issues. Some of them may lead to new ways of modeling reflection not applied in this work. For instance, the ideas implemented in high-level low-level programming frameworks such as *Benzo* [BDSC14] and *org.vmmagic* [FBC$^+$09] may be suitable for supporting reflective capabilities for the memory component.

A set of quantitative metrics is needed to precisely distinguish the reflectivity of different solutions incorporating VM-level reflection. Classical models of reflection like Smith's et al. [Smi84] and the denotational semantics presented by Wand and Friedman [WF86] do not enable to distinguish reflective capabilities at fine-grained levels. Therefore, we would like to devise a novel formalization model fulfilling our needs. On the other hand, we would also like to perform a user study to investigate how developers manage to handle the adaptation scenarios with the different approaches. Furthermore we also plan to extend our benchmarks with real resource demanding applications. Last but not least, we plan to reproduce the same experiments in reflective VMs for different dynamic languages to validate if our results can be generalized to other languages.

Finally, we would like to understand the potential consistency issues raised when low-level components are adapted. This can arise when modifying the representation of objects in memory at run time because the system may have different representations for the same instances. Encapsulation and indirection are the standard ways of dealing with them, but other approaches like monitoring of invariants and on-demand compensation might be good alternatives.

10.2 Resumen en Castellano

Esta tesis introduce el concepto de plataforma de software reflexiva y discute sus principales propiedades. Para evaluar su factibilidad y comprender mejor su potencial, diseñamos e implementamos MATE$_{PE}$ y MATE$_{MT}$: dos MVs de Smalltalk que soportan compilación dinámica y capacidades reflexivas avanzadas a nivel de la MV. Ambas permiten inspeccionar e interceder tanto el comportamiento como la estructura de diversos conceptos de la MV: operaciones del intérprete, el mecánismo de lookup de métodos, el layout de objetos y la pila de ejecución. El grado de capacidades reflexivas alcanzado por MATE$_{PE}$ y MATE$_{MT}$ es ya un indicador positivo respecto a la factibilidad de MVs con capacidades reflexivas avanzadas.

Mediante una evaluación empírica estudiamos el potencial de las MVs reflexivas para abordar escenarios de adaptación dinámica no anticipados y de trazo fino en tiempo de ejecución. Contrastamos nuestra solución con diversas alternativas considerando un amplio espectro de aspectos cualitativos. Los resultados mostraron que nuestras MVs reflexivas pudieron afrontar éxitosamente tanto los escenarios que involucraban operaciones estrucurales como aquellos que requerían cambios comportamentales. En todos los casos, la solución usando la MV reflexiva requirió pocas líneas de código y presentó beneficios respecto a las alternativas estudiadas. Además, no tenemos conocimiento de ninguna otra plataforma capaz de abordar todos los escenarios. Resumiendo, presentamos evidencia que muestra que las MVs refexivas son una alternativa adecuada para encarar el desarrollo de software flexible.

También presentamos un análisis sustentando por qué, desde nuestra perspectiva, los sistemas reflexivos contemporáneos, la programación orientada a aspectos y la programación orientada a contextos exponen limitaciones fundamentales para afrontar la adaptación dinámica de modo general. Particularmente, exhibimos las limitaciones de estas soluciones para abordar los escenarios adaptativos que nuestras MVs reflexivas afrontaron exitosamente. La razón principal de esta limitación es que no son capaces de expresar adaptaciones dinámicas para un amplio espectro de entidades de bajo nivel. En consecuencia, conjeturamos que, a diferencia de las soluciones alternativas, las MVs reflexivas exhiben los fundamentos adecuados para desarrollar software dinámico. Aunque este enunciado requiere de una evaluación más exhaustiva, presentamos una discusión elaborada sobre las razones por las cuales las MVs reflexivas subsumen, al menos conceptualmente, a las tres alternativas mencionadas.

Este trabajo también proporciona evidencia de que las MVs reflexivas pueden correr aplicaciones de manera eficiente. Esto contradice la creencia general de que los mecanismos reflexivos, al ser adoptados a bajo nivel, no son prácticos. Además, despeja el camino para que se realicen más estudios en el área. Las optimizaciónes principales aplicadas tanto a MATE$_{PE}$ como MATE$_{MT}$ son la especulación respecto a una baja meta-variabilidad y la combinación del almacenamiento de los metaobjetos con los layouts de los objetos estándar. Nuestros *benchmarks* mostraron que cuando el MOP no es activado, o cuando solo se usa para redefinir operaciones individuales, las penalidades de la MV reflexiva al analizar el desempeño asintótico son despreciables. Más aún, las penalidades son de cómo máximo 5x cuando se abordan escenarios de adaptación dinámica más complejos y abarcativos.

También presentamos algunos tipos de variabilidad dependientes de los aspectos específicos de cada aplicación: variabilidad efímera, de tiempo de inicialización, profundamente indirecta y anómala. Todas estas variabilidades acarrean penalidades de desempeño que podrían ser mitigadas si el desarrollador tuviese las herramientas necesarias para transmitirle al compilador información adicional con la cual podría mejorar el funcionamiento de las heurísticas en estos casos. Para esto, propusimos aprovechar las capacidades reflexivas de compilación de una MV reflexiva. Presentamos una API cubriendo varios aspectos del proceso de compilación, implementamos un subconjunto en MATE$_{PE}$ y obtuvimos disminuciones significativas en las penalidades de desempeño para un par de escenarios con variabilidad específica.

Creemos que este trabajo sienta las bases para abordar un estudio exhaustivo de las limitaciones fundamentales que exhiben los sistemas reflexivos en general. Tradicionalmente, fueron considerados lentos, inseguros y hasta innecesarios. Por eso los lenguajes y las plataformas de software más establecidas solo implementan capacidades reflexivas de manera muy restringida. Este trabajo discute esa creencia al presentar evidencia empírica de que los sistemas reflexivos avanzados son útiles y pueden implementarse sin afectar necesariamente el desempeño general del sistema.

En resumen, pudimos validar nuestras hipótesis respecto al potencial de las MVs reflexivas para afrontar la evolución dinámica de software (H1) y respecto a la posibilidad de construirlas de manera eficiente (H2). Finalmente, también mostramos que pueden servir para mejorar el desempeño de aplicaciones (H3).

A partir de estos resultados, el trabajo futuro podría explorar cómo el enfoque propuesto por la arquitectura MATE podría ser aplicado a otros componentes de la MV. Por ejemplo, nuestras implementaciones no reifican la memoria y, por lo tanto, no pueden adaptar directamente nociones como los manejadores de basura. Nuestro plan es explorar el impacto de proveer capacidades reflexivas para este componente. También planificamos proveer capacidades reflexivas más completas para los componentes de ejecución.

Respecto al desempeño, tenemos planes de extender la API de compilación. Conjeturamos que las capacidades reflexivas de un compilador dinámico podrían mejorar diversos aspectos de las aplicaciones dinámicas. Por ejemplo los tiempos de inicialización.

En general, nuestro objetivo de largo plazo es comprender cabalmente las limitaciones fundamentales de la reflexión a nivel de la MV. Nos interesa responder preguntas del estilo de cúal es el subconjunto mínimo de elementos de una MV reflexiva que no pueden ser reificados de manera práctica y cuál la consecuencia de reificar cada componente/operación en particular.

Por otro lado, métricas cuantitativas para distinguir con precisión los grados de reflexividad de las diferentes soluciones son necesarias. Nos gustaría desarrollar un nuevo modelo al respecto. También, realizar un experimento con usuarios para comprender cómo los desarrolladores afrontan escenarios de adaptación con las distintas herramientas.

Finalmente, nos interesan estudiar los eventuales problemas de consistencia que podrían emerger cuando los componentes de bajo nivel son adaptados en tiempo de ejecución. La encapsulación y la indirección son los mecanismos estándar para tratar con estos problemas, pero otros enfoques como el monitoreo de invariantes y la compensación bajo demanda podrían ser buenas alternativas para el dominio de las MVs reflexivas.

Bibliography

[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, and V. Sarkar. The jikes research virtual machine project: Building an open-source research community. *IBM Syst. J.*, 44(2):399–417, January 2005.

[AACM07] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. Rpython: A step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM.

[ABB$^+$89] Giuseppe Attardi, Cinzia Bonini, Maria Rosario Boscotrecase, Tito Flagella, and Mauro Gaspari. Metalevel programming in clos. In *ECOOP*, volume 89, pages 243–256, 1989.

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM.

[ACS$^+$14] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. Improving javascript performance by deconstructing the type system. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 496–507, New York, NY, USA, 2014. ACM.

[ADD$^+$10] Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Mathieu Suen. Read-only execution for dynamic languages. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 117–136. Springer-Verlag, 2010.

[AG05] Matthew Arnold and David Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society.

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-oriented Applications*. PhD thesis, Stanford, CA, USA, 1996. UMI Order No. GAX96-20452.

[AKE08] Shay Artzi, Sunghun Kim, and Michael D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 542–565, Berlin, Heidelberg, 2008. Springer-Verlag.

[Asa11] Kenichi Asai. Reflection in direct style. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 97–106. ACM, 2011.

[Asa14] Kenichi Asai. Compiling a reflective language using metaocaml. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 113–122, New York, NY, USA, 2014. ACM.

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

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, pages 257–281, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[BBH$^+$15] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: A tracing jit for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 22–34, New York, NY, USA, 2015. ACM.

[BBnRR12] Alexandre Bergel, Felipe Bañados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. *Comput. Lang. Syst. Struct.*, 38(1):16–28, April 2012.

[BBTK$^+$17] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA):52:1–52:27, October 2017.

[BC89] J.-P. Briot and P. Cointe. Programming with explicit metaclasses in smalltalk-80. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 419–431, New York, NY, USA, 1989. ACM.

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[BDN$^+$09] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example.* Square Bracket Associates, 2009.

[BDSC14] Camillo Bruni, Stéphane Ducasse, Igor Stasenko, and Guido Chari. Benzo: Reflective Glue for Low-level Programming. In *International Workshop on Smalltalk Technologies*, August 2014.

[BDT13] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage strategies for collections in dynamically typed languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 167–182. ACM, 2013.

[BFJR98] John Brant, Brian Foote, Ralph E. Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 396–417, London, UK, UK, 1998. Springer-Verlag.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 215–230, New York, NY, USA, 1993. ACM.

[BKL$^+$08] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. Self-sustaining systems. chapter Back to the Future in One Week – Implementing a Smalltalk VM in PyPy, pages 123–139. Springer-Verlag, Berlin, Heidelberg, 2008.

[BNRR11] Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-specific profiling. In *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns*, TOOLS'11, pages 68–82, Berlin, Heidelberg, 2011. Springer-Verlag.

[BR07] Carl Friedrich Bolz and Armin Rigo. How to not write virtual machines for dynamic languages. In *3rd Workshop on Dynamic Languages and Applications*, 2007.

[Bra04] Gilad Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, volume 4, 2004.

[BT15] Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on vm design and implementation. *Sci. Comput. Program.*, 98(P3):408–421, February 2015.

[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 '04, pages 331–344. ACM, 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.

[CCR15] Ruzanna Chitchyan, Walter Cazzola, and Awais Rashid. Engineering sustainability through language. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 501–504, Piscataway, NJ, USA, 2015. IEEE Press.

[CCRS17] Walter Cazzola, Ruzanna Chitchyan, Awais Rashid, and Albert Shaqiri. μ-dsu: A micro-language based approach to dynamic software updating. *Computer Languages, Systems & Structures*, 2017.

[CEI$^+$12] Jose Castanos, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Toshio Nakatani, Takeshi Ogasawara, and Peng Wu. On the benefits and pitfalls of extending a statically typed language jit compiler for dynamic scripting languages. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 195–212, New York, NY, USA, 2012. ACM.

[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 2016, pages 60–71, New York, NY, USA, 2016. ACM.

[CGM17a] Guido Chari, Diego Garbervetsky, and Stefan Marr. Fully-reflective vms for ruling software adaptation. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 229–231, Piscataway, NJ, USA, 2017. IEEE Press.

[CGM17b] Guido Chari, Diego Garbervetsky, and Stefan Marr. A metaobject protocol for optimizing application-specific run-time variability. In *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS'17, pages 3:1–3:5, New York, NY, USA, 2017. ACM.

[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!)*, Onward! 2015, pages 240–253, New York, NY, USA, 2015. ACM.

[CKL96] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The Meta-Helix. In *Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software*, ISOTAS '96, pages 157–172. Springer-Verlag, 1996.

[CMS$^+$11] Mason Chang, Bernd Mathiske, Edwin Smith, Avik Chaudhuri, Andreas Gal, Michael Bebenita, Christian Wimmer, and Michael Franz. The impact of optional type information on jit compilation of dynamically typed languages. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 13–24, New York, NY, USA, 2011. ACM.

[Coi87] Pierre Cointe. Metaclasses are first class: The objvlisp model. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 156–162, New York, NY, USA, 1987. ACM.

[CSR$^+$09] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: Trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 71–80, New York, NY, USA, 2009. ACM.

[CU89] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 146–160, New York, NY, USA, 1989. ACM.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. OOPSLA '91, pages 1–15. ACM, 1991.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, pages 49–70. ACM, October 1989.

[DDHV03] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 149–168, New York, NY, USA, 2003. ACM.

[DHV03] Bruno Dufour, Laurie Hendren, and Clark Verbrugge. *j: A tool for dynamic analysis of java programs. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 306–307, New York, NY, USA, 2003. ACM.

[Dmi03] Mikhail Dmitriev. Design of jfluid: A profiling technology and tool based on dynamic bytecode instrumentation. Technical report, Mountain View, CA, USA, 2003.

[DRF$^+$14] Zachary DeVito, Daniel Ritchie, Matt Fisher, Alex Aiken, and Pat Hanrahan. First-class runtime generation of high-performance types using exotypes. In *PLDI*, pages 77–88. ACM, 2014.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings POPL '84*, January 1984.

[DSD08a] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The meta in meta-object architectures. *Objects, Components, Models and Patterns*, pages 218–237, 2008.

[DSD08b] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. *The Meta in Meta-object Architectures*, pages 218–237. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in smalltalk. *JOURNAL OF OBJECT ORIENTED PROGRAMMING*, 12:39–50, 1999.

[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*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM.

[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*, VEE '09, pages 81–90. ACM, 2009.

[Fut83] Yoshihiko Futamura. Partial computation of programs. In *RIMS Symposia on Software Science and Engineering*, pages 1–35. Springer, 1983.

[Fut99] Yoshihiko Futamura. Partial evaluation of computation process&mdash;anapproach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, December 1999.

[GDGC94] Charles D Garrett, Jeffrey Dean, David Grove, and Craig Chambers. Measurement and application of dynamic receiver class distributions. Technical report, Technical Report CSE-TR-94-03-05, University of Washington, 1994.

[GES+09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[Goo] The v8 javascript vm, 2009. https://developers.google.com/v8.

[GR83a] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[GR83b] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[GTL+10] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. Vmkit: A substrate for managed runtime environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 51–62. ACM, 2010.

[Hc11] Andrei Homescu and Alex Şuhan. Happyjit: A tracing jit compiler for php. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 25–36, New York, NY, USA, 2011. ACM.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[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, London, UK, UK, 1991. Springer-Verlag.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM.

[HCvW07] Danny Holten, Bas Cornelissen, and Jarke J. van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 0:47–54, 2007.

[HG12] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM.

[HGA$^+$09] M. Haupt, C. Gibbs, B. Adams, S. Timbermont, Y. Coady, and R. Hirschfeld. Disentangling virtual machine architecture. *Software, IET*, June 2009.

[HH09] Alex Holkner and James Harland. Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pages 19–28, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[HHP$^+$10] Michael Haupt, Robert Hirschfeld, Tobias Pape, Gregor Gabrysiak, Stefan Marr, Arne Bergmann, Arvid Heise, Matthias Kleine, and Robert Krahn. The som family: Virtual machines for teaching and research. In *ITiCSE*, pages 18–22. ACM, 2010.

[HN05] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, November 2005.

[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, New York, NY, USA, 1994. ACM.

[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.

[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.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[KJ13] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 63–74, New York, NY, USA, 2013. ACM.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[KR91] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KRR$^+$13] Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. Improved type specialization for dynamic scripting languages. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 37–48, New York, NY, USA, 2013. ACM.

[Kul11] Prasad A. Kulkarni. Jit compilation policy for modern machines. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 773–788, New York, NY, USA, 2011. ACM.

[KWM$^+$08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 147–155. ACM, 1987.

[McA95] Jeff McAffer. Meta-level programming with coda. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 190–214, London, UK, UK, 1995. Springer-Verlag.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[MD08] Tom Mens and Serge Demeyer. *Software Evolution.* Springer Publishing Company, Incorporated, 1 edition, 2008.

[MD12] Stefan Marr and Theo D'Hondt. Identifying a unifying mechanism for the implementation of concurrency abstractions on multi-language virtual machines. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 171–186, Berlin, Heidelberg, 2012. Springer-Verlag.

[MD15] Stefan Marr and Stéphane Ducasse. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In *OOPSLA*, pages 821–839. ACM, 2015.

[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. ACM, 2016.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction.* Prentice-Hall, Inc., 1997.

[MHR$^+$15] Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. Columnar objects: Improving the performance of analytical applications. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 197–210, New York, NY, USA, 2015. ACM.

[Mir11] Eliot Miranda. The Cog Smalltalk virtual machine. In *Proceedings of the 5th Workshop on Virtual Machines and Intermediate Languages*, VMIL '11. ACM, 2011.

[MSD15] Stefan Marr, Chris Seaton, and Stéphane Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 545–554. ACM, 2015.

[NBD$^+$05] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gälli, and Roel Wuyts. On the revival of dynamic languages. In *Proceedings of the 4th International Conference on Software Composition*, SC'05, pages 1–13, Berlin, Heidelberg, 2005. Springer-Verlag.

[NDG$^+$08] Oscar Nierstrasz, Marcus Denker, Tudor Gîrba, Adrian Lienhard, and David Röthlisberger. Change-enabled software systems. In Martin Wirsing, Jean-Pierre Banâtre, Matthias Hölzl, and Axel Rauschmayer, editors, *Software-Intensive Systems and New Computing Paradigms*, pages 64–79. Springer-Verlag, Berlin, Heidelberg, 2008.

[Nie89] Oscar Nierstrasz. Object-oriented concepts, databases, and applications. chapter A Survey of Object-oriented Concepts, pages 3–21. ACM, New York, NY, USA, 1989.

[PDF$^+$15] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Max Mattone. Virtualization support for dynamic core library update. In *Onward! 2015*, 2015.

[PFH16] Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. Optimizing sideways composition: Fast context-oriented programming in contextpypy. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, COP'16, pages 13–20, New York, NY, USA, 2016. ACM.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[Pla09] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 1–2, New York, NY, USA, 2009. ACM.

[Pla13] Hasso Plattner. *A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases*. Springer Publishing Company, Incorporated, 2013.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[RBN12] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 485–495. IEEE Press, 2012.

[RC02] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 205–230. Springer-Verlag, 2002.

[RDT08] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Comput. Lang. Syst. Struct.*, 34(2-3):46–65, July 2008.

[RHB$^+$12] David Rothlisberger, Marcel Harry, Walter Binder, Philippe Moret, Danilo Ansaloni, Alex Villazon, and Oscar Nierstrasz. Exploiting dynamic information in ides improves speed and correctness of software maintenance tasks. *IEEE Trans. Softw. Eng.*, 38(3):579–591, May 2012.

[RHV$^+$09] D. Rothlisberger, M. Harry, A. Villazon, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in ides with dynamic metrics. In *ICSM '09*, pages 253–262, Sept 2009.

[RLBV10] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. *SIGPLAN Not.*, 45(6):1–12, June 2010.

[RLZ10] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. Jsmeter: Comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[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.

[RRGN10] Jorge Ressia, Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. O.: Run-time evolution through explicit meta-objects. In *In: Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS*, pages 37–48, 2010.

[RSB$^+$14] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. Surgical precision jit compilers. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 41–52. ACM, 2014.

[SC11] Amin Shali and William R. Cook. Hybrid partial evaluation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 375–390, New York, NY, USA, 2011. ACM.

[SGP13] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. An analysis of language-level support for self-adaptive software. *TAAS*, 8(2):7, 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.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[Spi] Spidermonkey.

[SVDVH14] Chris Seaton, Michael L. Van De Vanter, and Michael Haupt. Debugging at full speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 2:1–2:13, New York, NY, USA, 2014. ACM.

[SWHJ16] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. Optimizing r language execution via aggressive speculation. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 84–95, New York, NY, USA, 2016. ACM.

[SWU$^+$15] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. Snippets: Taking the High Road to a Low Level. *ACM Trans. Archit. Code Optim.*, 12(2):20:1–20:25, June 2015.

[SYK$^+$05] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785, July 2005.

[Tan09] Éric Tanter. Reflection and open implementations. Technical report, DCC, University of Chile, 2009.

[TE05] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 211–230. ACM, 2005.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 27–46. ACM, 2003.

[Tra09] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, July 2009.

[USA05] David Ungar, Adam Spitz, and Alex Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, 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. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 774–789. ACM, 2010.

[VBLN11] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: Enabling lightweight language extensions by intercepting slot access. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11. ACM, 2011.

[VBMA11] Alex Villazón, Walter Binder, Philippe Moret, and Danilo Ansaloni. Comprehensive aspect weaving for java. *Sci. Comput. Program.*, 76(11):1015–1036, November 2011.

[VC15] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

[VCM13] Tom Van Cutsem and Mark S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 154–178, Berlin, Heidelberg, 2013. Springer-Verlag.

[VRDJ95] Guido Van Rossum and Fred L Drake Jr. *Python reference manual.* Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[WBLF12] Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. Fine-grained modularity and reuse of virtual machine components. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 203–214. ACM, 2012.

[WF86] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 298–307. ACM, 1986.

[Wha00] John Whaley. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 78–87, New York, NY, USA, 2000. ACM.

[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.

[WLB+15] Kunshan Wang, Yi Lin, Stephen M. Blackburn, Michael Norrish, and Antony L. Hosking. Draining the Swamp: Micro Virtual Machines as Solid Foundation for Language Development. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 321–336, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[WNTD14] Erwann Wernli, Oscar Nierstrasz, Camille Teruel, and Stéphane Ducasse. Delegation proxies: The power of propagation. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 1–12. ACM, 2014.

[WWB+14] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the truffle language implementation framework. In *PPPJ*, pages 133–144. ACM, 2014.

[WWH+17] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 662–676, New York, NY, USA, 2017. ACM.

[WWS+12] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *DLS*, pages 73–82. ACM, 2012.

[WWW+13] 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.

[ZBB17] Yudi Zheng, Lubomír Bulej, and Walter Binder. An empirical study on deoptimization in the graal compiler. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, pages 30:1–30:30, 2017.

[ZPA+07] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kie, un, and Michael D. Ernst. Object and reference immutability using java generics. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 75–84. ACM, 2007.

[ZSCC06] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 263–271, New York, NY, USA, 2006. ACM.

10.3 Appendix

In this appendix we present the detailed results from our performance evaluation.

10.3.1 Baseline

Table 10.1 shows the results for the baseline experiment for each configuration individually.

Table 10.1: Baseline Results

Benchmark	VM	OF	Confidence	Sd	Median	Min	Max
Bounce	Node	6.09	<6.08 - 6.09>	0.02	6.08	6.07	6.17
	Pharo	11.28	<11.27 - 11.29>	0.00	11.28	11.27	11.30
	SOMmt	2.28	<2.28 - 2.29>	0.01	2.28	2.28	2.31
	SOMpe	2.06	<2.05 - 2.06>	0.02	2.05	2.04	2.09
CD	Node	2.22	<2.19 - 2.25>	0.01	2.22	2.21	2.26
	SOMmt	8.45	<8.32 - 8.59>	0.19	8.57	8.11	8.62
	SOMpe	5.99	<5.85 - 6.14>	0.40	5.84	5.83	7.45
DeltaBlue	Node	4.04	<3.82 - 4.28>	0.57	3.83	3.49	5.30
	Pharo	6.96	<6.29 - 7.66>	2.14	5.69	5.56	10.50
	SOMmt	5.71	<5.49 - 5.95>	0.02	5.71	5.66	5.78
	SOMpe	3.51	<3.37 - 3.65>	0.12	3.47	3.41	4.11
Havlak	Node	10.43	<10.08 - 10.8>	0.16	10.41	10.16	10.98
	Pharo	15.45	<14.92 - 16.01>	0.51	15.42	14.51	16.26
	SOMmt	12.82	<12.4 - 13.28>	0.08	12.83	12.66	12.97
	SOMpe	2.65	<2.54 - 2.76>	0.21	2.56	2.55	3.40
Json	Node	1.50	<1.45 - 1.56>	0.00	1.50	1.50	1.50
	Pharo	9.35	<9.01 - 9.72>	0.01	9.35	9.33	9.37
	SOMmt	1.69	<1.63 - 1.76>	0.03	1.69	1.65	1.73
	SOMpe	1.80	<1.73 - 1.87>	0.02	1.79	1.79	1.86
List	Node	2.58	<2.57 - 2.58>	0.00	2.58	2.57	2.59
	Pharo	14.73	<14.72 - 14.75>	0.00	14.73	14.73	14.74
	SOMmt	12.06	<12.04 - 12.08>	0.07	12.02	12.01	12.22
	SOMpe	6.23	<6.06 - 6.4>	0.58	6.09	5.84	9.06
Mandelbrot	Node	0.84	<0.84 - 0.84>	0.00	0.84	0.84	0.85
	Pharo	3.61	<3.61 - 3.61>	0.00	3.61	3.61	3.62
	SOMmt	1.49	<1.49 - 1.49>	0.00	1.49	1.49	1.49
	SOMpe	0.96	<0.96 - 0.96>	0.00	0.96	0.96	0.97

Table 10.1: Baseline Results *(continued)*

Benchmark	VM	OF	Confidence	Sd	Median	Min	Max
NBody	Node	2.00	<2 - 2.01>	0.01	2.00	1.99	2.03
	Pharo	13.37	<13.36 - 13.39>	0.01	13.37	13.36	13.39
	SOMmt	3.18	<3.17 - 3.18>	0.01	3.17	3.17	3.23
	SOMpe	3.17	<3.16 - 3.18>	0.02	3.17	3.14	3.24
Permute	Node	1.90	<1.9 - 1.9>	0.00	1.90	1.90	1.91
	Pharo	7.93	<7.91 - 7.94>	0.01	7.92	7.91	7.97
	SOMmt	6.98	<6.96 - 7>	0.05	6.97	6.96	7.17
	SOMpe	3.92	<3.85 - 4>	0.26	3.82	3.82	5.48
Queens	Node	2.39	<2.39 - 2.4>	0.01	2.39	2.39	2.43
	Pharo	8.21	<8.19 - 8.23>	0.01	8.21	8.19	8.23
	SOMmt	6.20	<6.18 - 6.22>	0.04	6.20	6.19	6.44
	SOMpe	4.30	<4.29 - 4.31>	0.00	4.30	4.29	4.31
Richards	Node	2.71	<2.71 - 2.71>	0.00	2.71	2.71	2.73
	Pharo	7.29	<7.28 - 7.3>	0.00	7.29	7.29	7.31
	SOMmt	2.84	<2.84 - 2.84>	0.01	2.84	2.83	2.89
	SOMpe	4.19	<4.19 - 4.2>	0.00	4.19	4.19	4.20
Sieve	Node	2.32	<2.31 - 2.34>	0.00	2.32	2.32	2.33
	Pharo	8.68	<8.6 - 8.77>	0.24	8.63	8.63	9.82
	SOMmt	4.53	<4.5 - 4.55>	0.02	4.52	4.51	4.58
	SOMpe	1.89	<1.88 - 1.9>	0.00	1.89	1.88	1.90
Storage	Node	1.33	<1.29 - 1.38>	0.00	1.33	1.33	1.34
	Pharo	2.72	<2.64 - 2.81>	0.05	2.72	2.57	2.82
	SOMmt	2.38	<2.31 - 2.46>	0.02	2.38	2.36	2.44
	SOMpe	0.87	<0.84 - 0.9>	0.06	0.85	0.84	1.11
Towers	Node	1.84	<1.84 - 1.85>	0.00	1.84	1.84	1.85
	Pharo	5.82	<5.8 - 5.83>	0.00	5.82	5.81	5.82
	SOMmt	2.99	<2.98 - 3>	0.04	2.98	2.97	3.13
	SOMpe	5.39	<5.36 - 5.41>	0.07	5.36	5.33	5.66

10.3.2 Inherent

Tables 10.2 and 10.3 shows the results for the inherent experiment for each configuration individually separated by macro and micro benchmarks.

Table 10.2: Micro Benchmark Inherent Performance Results

Benchmark	VM	OF	Confidence	Sd	Median	Min	Max
BubbleSort	MATEmt	0.91	<0.91 - 0.92>	0.00	0.91	0.91	0.92
	MATEpe	1.03	<1 - 1.05>	0.08	0.99	0.98	1.25
Dispatch	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.00
	MATEpe	0.61	<0.59 - 0.62>	0.04	0.60	0.59	0.84
Fannkuch	MATEmt	0.98	<0.98 - 0.98>	0.00	0.98	0.98	0.99
	MATEpe	0.90	<0.88 - 0.91>	0.03	0.89	0.89	1.06
FieldLoop	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.00
	MATEpe	0.97	<0.92 - 1.01>	0.09	0.93	0.93	1.34
Loop	MATEmt	1.01	<1.01 - 1.01>	0.00	1.01	1.01	1.01
	MATEpe	0.93	<0.91 - 0.96>	0.01	0.93	0.93	0.95
	MATEmt	1.05	<1.05 - 1.05>	0.00	1.05	1.04	1.05

Table 10.2: Micro Benchmark Inherent Performance Results *(continued)*

Benchmark	VM	OF	Confidence	Sd	Median	Min	Max
QuickSort	MATEpe	0.87	<0.85 - 0.9>	0.01	0.87	0.87	0.90
Sum	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.00
	MATEpe	1.00	<1 - 1>	0.00	1.00	1.00	1.01
TreeSort	MATEmt	1.01	<1.01 - 1.01>	0.00	1.01	1.01	1.02
	MATEpe	0.85	<0.84 - 0.87>	0.03	0.85	0.84	1.05
WhileLoop	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.00
	MATEpe	0.90	<0.88 - 0.91>	0.00	0.89	0.89	0.91
Bounce	MATEmt	1.01	<1.01 - 1.01>	0.00	1.01	1.01	1.02
	MATEpe	0.81	<0.8 - 0.82>	0.03	0.80	0.79	1.02
List	MATEmt	0.99	<0.99 - 1>	0.01	0.99	0.99	1.01
	MATEpe	0.80	<0.78 - 0.82>	0.03	0.79	0.79	0.95
Mandelbrot	MATEmt	1.00	<1 - 1>	0.00	1.00	1.00	1.00
	MATEpe	1.00	<1 - 1>	0.00	1.00	1.00	1.00
NBody	MATEmt	1.01	<1.01 - 1.01>	0.01	1.01	1.00	1.03
	MATEpe	0.49	<0.49 - 0.5>	0.02	0.49	0.48	0.56
Permute	MATEmt	0.99	<0.98 - 0.99>	0.01	0.98	0.98	1.02
	MATEpe	0.84	<0.82 - 0.86>	0.04	0.82	0.82	1.05
Queens	MATEmt	0.95	<0.95 - 0.96>	0.01	0.95	0.95	0.99
	MATEpe	0.85	<0.84 - 0.86>	0.03	0.84	0.84	1.00
Sieve	MATEmt	0.99	<0.99 - 1>	0.00	0.99	0.99	1.01
	MATEpe	0.93	<0.93 - 0.93>	0.00	0.93	0.93	0.94
Storage	MATEmt	0.99	<0.99 - 0.99>	0.01	0.99	0.98	1.02
	MATEpe	0.78	<0.74 - 0.83>	0.16	0.72	0.72	1.59
Towers	MATEmt	1.03	<1.02 - 1.03>	0.01	1.02	1.02	1.08
	MATEpe	0.80	<0.78 - 0.82>	0.07	0.78	0.78	1.20

Table 10.3: Macro Benchmark Inherent Performance Results

Benchmark	VM	OF	Confidence	Sd	Median	Min	Max
GraphSearch	MATEmt	1.05	<1.05 - 1.05>	0.00	1.05	1.04	1.05
	MATEpe	0.83	<0.81 - 0.84>	0.03	0.82	0.81	1.00
PageRank	MATEmt	1.01	<1 - 1.01>	0.00	1.01	1.00	1.02
	MATEpe	0.96	<0.93 - 0.99>	0.08	0.94	0.93	1.43
CD	MATEmt	1.19	<1.18 - 1.2>	0.01	1.19	1.15	1.24
	MATEpe	0.96	<0.94 - 0.98>	0.01	0.96	0.96	0.99
DeltaBlue	MATEmt	1.02	<1.02 - 1.02>	0.00	1.02	1.01	1.03
	MATEpe	0.68	<0.67 - 0.7>	0.04	0.68	0.66	0.93
Havlak	MATEmt	1.02	<1.01 - 1.02>	0.01	1.02	1.00	1.04
	MATEpe	0.91	<0.88 - 0.95>	0.09	0.88	0.86	1.22
Json	MATEmt	1.03	<1.02 - 1.03>	0.02	1.04	1.00	1.05
	MATEpe	0.87	<0.84 - 0.89>	0.07	0.84	0.84	1.26
Richards	MATEmt	1.02	<1.02 - 1.02>	0.00	1.02	1.02	1.04
	MATEpe	0.43	<0.43 - 0.43>	0.00	0.43	0.43	0.44

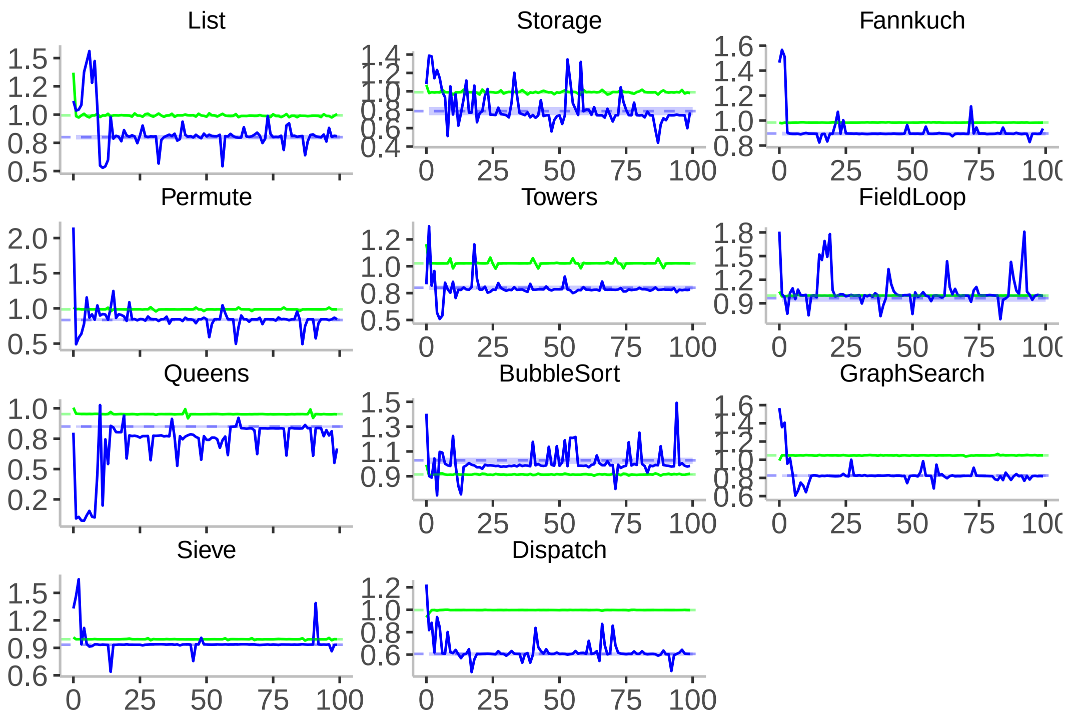

Figure 10.1: The first 100 iterations of a subset of the benchmarks in MATEpe and MATEmt normalized to their corresponding SOM versions.

10.3.3 Warmups

Table 10.4 shows the warmup times for each configuration clustered by the amount of iterations we take into account. On the other hand, Figures 10.1 and 10.2 shows the warmups of all the remaining benchmarks.

Table 10.4: Overall Warmup Results

Benchmark	VM	Iterations	Sd	OF	Confidence
Bounce	MATEpe	1	-	1.07	-
	MATEmt		-	1.15	-
	MATEpe	5	950.96	1.08	-
	MATEmt		44.38	1.07	<0.32 - 2.97>
	MATEpe	20	506.91	0.88	-
	MATEmt		22.81	1.03	<0.78 - 1.35>
	MATEpe	40	361.26	0.88	<-0.02 - 5.67>
	MATEmt		16.19	1.02	<0.89 - 1.18>
CD	MATEpe	1	-	1.15	-
	MATEmt		-	0.78	-
	MATEpe	5	4904.45	1.35	-
	MATEmt		392.77	0.82	-
	MATEpe	20	2621.51	1.29	-
	MATEmt		201.17	0.96	<0.51 - 2.08>
	MATEpe	40	1878.95	1.28	-
	MATEmt		142.86	1.05	<0.73 - 1.59>
	MATEpe	1	-	1.34	-
	MATEmt		-	1.25	-
	MATEpe		3904.68	1.44	-

Table 10.4: Overall Warmup Results *(continued)*

Benchmark	VM	Iterations	Sd	OF	Confidence
DeltaBlue	MATEmt	5	263.29	1.17	-
	MATEpe	20	2239.65	1.38	-
	MATEmt		133.74	1.08	<0.51 - 2.09>
	MATEpe	40	1631.72	1.32	<-0.02 - 50.31>
	MATEmt		94.93	1.06	<0.72 - 1.51>
Havlak	MATEpe	1	-	1.02	-
	MATEmt		-	1.07	-
	MATEpe	5	6738.95	0.92	-
	MATEmt		284.4	1.04	<0.73 - 1.45>
	MATEpe	20	3673.28	0.89	-
	MATEmt		146.5	1.02	<0.94 - 1.11>
	MATEpe	40	2634.06	0.88	<-0.05 - 9.93>
	MATEmt		103.92	1.02	<0.98 - 1.06>
Json	MATEpe	1	-	1.36	-
	MATEmt		-	1.14	-
	MATEpe	5	2965.49	1.14	-
	MATEmt		294.4	1.09	<0.16 - 5.48>
	MATEpe	20	1496.21	0.99	<-0.24 - 7.85>
	MATEmt		159.45	1.05	<0.65 - 1.65>
	MATEpe	40	1064.42	0.94	<0.04 - 3.21>
	MATEmt		114.1	1.04	<0.8 - 1.33>
List	MATEpe	1	-	1.12	-
	MATEmt		-	1.37	-
	MATEpe	5	854	1.12	-
	MATEmt		138.44	1.09	<0.65 - 1.6>
	MATEpe	20	435.43	1.00	<0.28 - 2.93>
	MATEmt		69.47	1.02	<0.92 - 1.12>
	MATEpe	40	312.59	0.92	<0.46 - 1.69>
	MATEmt		49.14	1.01	<0.96 - 1.06>
Mandelbrot	MATEpe	1	-	2.54	-
	MATEmt		-	1.02	-
	MATEpe	5	4244.42	2.67	<0.17 - 32.17>
	MATEmt		33.25	1.01	<0.62 - 1.62>
	MATEpe	20	2833.12	2.52	<-0.15 - 121.7>
	MATEmt		17.24	1.00	<0.89 - 1.13>
	MATEpe	40	2062.57	2.36	<-0.06 - 22.56>
	MATEmt		12.24	1.00	<0.94 - 1.07>
	MATEpe	1	-	1.25	-
	MATEmt		-	1.00	-
	MATEpe	5	1454.49	1.14	-
	MATEmt		34.5	1.00	<0.61 - 1.64>
	MATEpe	20	727.03	0.97	-
	MATEmt		17.52	1.01	<0.89 - 1.14>
	MATEpe	40	514.38	0.86	<-0.05 - 3.5>
	MATEmt		12.4	1.01	<0.95 - 1.07>
	MATEpe		-	1.07	-

Table 10.4: Overall Warmup Results *(continued)*

Benchmark	VM	Iterations	Sd	OF	Confidence
NBody	MATEmt	1	-	0.92	-
	MATEpe	5	1783.67	1.07	-
	MATEmt		27.82	0.95	<0.5 - 1.88>
	MATEpe	20	887.48	0.99	<0.02 - 10.84>
	MATEmt		14.54	0.97	<0.82 - 1.15>
	MATEpe	40	632.34	1.00	<0.29 - 3.09>
	MATEmt		10.34	0.98	<0.89 - 1.06>
Permute	MATEpe	1	-	2.15	-
	MATEmt		-	1.01	-
	MATEpe	5	1851.75	1.72	-
	MATEmt		48.9	0.99	<0.7 - 1.39>
	MATEpe	20	929.9	1.44	<-0.57 - 12.48>
	MATEmt		24.77	0.99	<0.91 - 1.07>
	MATEpe	40	658.89	1.25	<-0.05 - 3.83>
	MATEmt		17.58	0.99	<0.95 - 1.03>
Queens	MATEpe	1	-	0.80	-
	MATEmt		-	1.01	-
	MATEpe	5	886.8	0.24	<-0.2 - 0.69>
	MATEmt		50.07	0.97	<0.61 - 1.52>
	MATEpe	20	464.09	0.24	<0.02 - 0.62>
	MATEmt		25.09	0.96	<0.86 - 1.07>
	MATEpe	40	332.79	0.30	<0.1 - 0.68>
	MATEmt		17.77	0.96	<0.9 - 1.01>
Richards	MATEpe	1	-	1.61	-
	MATEmt		-	1.09	-
	MATEpe	5	3400.79	1.35	-
	MATEmt		112.82	1.05	<0.56 - 1.89>
	MATEpe	20	1699.83	0.94	<-0.36 - 4.67>
	MATEmt		58.35	1.03	<0.88 - 1.2>
	MATEpe	40	1206.74	0.77	<-0.06 - 2.07>
	MATEmt		41.58	1.02	<0.95 - 1.1>
Sieve	MATEpe	1	-	1.33	-
	MATEmt		-	1.01	-
	MATEpe	5	672.84	1.31	-
	MATEmt		35.81	1.00	<0.78 - 1.27>
	MATEpe	20	348.69	1.13	<0.07 - 5.77>
	MATEmt		18	1.00	<0.94 - 1.05>
	MATEpe	40	247.75	1.06	<0.36 - 2.46>
	MATEmt		12.78	0.99	<0.97 - 1.02>
	MATEpe	1	-	1.08	-
	MATEmt		-	1.07	-
	MATEpe	5	1225.2	1.10	-
	MATEmt		68.39	1.01	<0.64 - 1.55>
	MATEpe	20	622.47	1.04	-
	MATEmt		34.45	1.00	<0.9 - 1.1>
	MATEpe		442.4	0.98	<-0.01 - 12.22>

Table 10.4: Overall Warmup Results *(continued)*

Storage

Benchmark	VM	Iterations	Sd	OF	Confidence
	MATEmt	40	24.4	0.99	<0.94 - 1.05>
Towers	MATEpe	1	-	0.83	-
	MATEmt		-	1.21	-
	MATEpe	5	1067.76	0.85	-
	MATEmt		68.15	1.09	<0.46 - 2.24>
	MATEpe	20	558.26	0.82	<0.12 - 5.05>
	MATEmt		34.28	1.05	<0.86 - 1.25>
	MATEpe	40	398.24	0.81	<0.31 - 2.03>
	MATEmt		24.28	1.04	<0.94 - 1.14>
All	MATEpe	1	-	1.04	-
	MATEmt		-	1.31	-
	MATEpe	5	990.73	1.05	-
	MATEmt		5.58	1.29	<0.71 - 2.3>
	MATEpe	20	593.04	1.04	-
	MATEmt		3.7	1.20	<0.97 - 1.48>
	MATEpe	40	427.99	1.03	<0 - 80.38>
	MATEmt		3.03	1.11	<0.97 - 1.26>
ArgRead	MATEpe	1	-	0.85	-
	MATEmt		-	1.00	-
	MATEpe	5	881	0.86	-
	MATEmt		4.63	1.02	<0.45 - 2.35>
	MATEpe	20	518.93	0.86	-
	MATEmt		2.7	1.02	<0.78 - 1.33>
	MATEpe	40	373.88	0.86	-
	MATEmt		2.22	1.01	<0.85 - 1.21>
FieldRead	MATEpe	1	0	1.09	-
	MATEmt		0	0.99	-
	MATEpe	5	939.48	1.00	-
	MATEmt		5.25	1.00	<0.61 - 1.9>
	MATEpe	20	569.1	1.00	-
	MATEmt		3.54	1.01	<0.82 - 1.26>
	MATEpe	40	412.05	1.00	-
	MATEmt		2.99	1.00	<0.87 - 1.17>
FieldWrite	MATEpe	1	0	1.02	-
	MATEmt		0	0.99	-
	MATEpe	5	863.65	0.99	-
	MATEmt		5.16	1.01	<0.61 - 1.88>
	MATEpe	20	530.65	0.99	-
	MATEmt		3.51	1.01	<0.82 - 1.26>
	MATEpe	40	384.74	1.00	-
	MATEmt		2.99	1.01	<0.87 - 1.17>
	MATEpe	1	-	0.84	-
	MATEmt		-	0.96	-
	MATEpe	5	797.02	0.86	-
	MATEmt		4.53	0.99	<0.45 - 2.28>
	MATEpe		473.7	0.86	-

Table 10.4: Overall Warmup Results *(continued)*

Benchmark	VM	Iterations	Sd	OF	Confidence
LocalRead	MATEmt	20	2.67	1.00	<0.77 - 1.3>
	MATEpe	40	341.59	0.87	-
	MATEmt		2.21	1.00	<0.83 - 1.2>
Send	MATEpe	1	-	0.82	-
	MATEmt		-	0.87	-
	MATEpe	5	624.02	0.88	-
	MATEmt		2.97	0.97	<0.54 - 1.9>
	MATEpe	20	379.55	0.89	-
	MATEmt		1.92	1.00	<0.81 - 1.24>
	MATEpe	40	274.26	0.89	-
	MATEmt		1.72	1.00	<0.86 - 1.17>
Activation	MATEpe	1	-	0.77	-
	MATEmt		-	1.33	-
	MATEpe	5	624.02	0.82	-
	MATEmt		2.97	1.32	<0.78 - 2.23>
	MATEpe	20	379.55	0.82	-
	MATEmt		1.92	1.24	<1.04 - 1.48>
	MATEpe	40	274.26	0.82	-
	MATEmt		1.72	1.19	<1.04 - 1.34>
Return	MATEpe	1	-	1.05	-
	MATEmt		-	1.13	-
	MATEpe	5	996.47	0.98	-
	MATEmt		5.56	1.14	<0.59 - 2.21>
	MATEpe	20	576.54	0.98	-
	MATEmt		3.58	1.10	<0.87 - 1.4>
	MATEpe	40	414.68	0.99	-
	MATEmt		3.01	1.07	<0.91 - 1.25>
SumKeys	MATEpe	1	-	1.43	-
	MATEmt		-	1.79	-
	MATEpe	5	801.01	1.45	-
	MATEmt		29.65	1.95	<0.31 - 43.53>
	MATEpe	20	466.97	1.31	-
	MATEmt		16.59	2.16	<1.21 - 4.22>
	MATEpe	40	337.68	1.29	<-0.12 - 45.62>
	MATEmt		11.91	2.48	<1.74 - 3.73>
CDT	MATEpe	1	-	1.53	-
	MATEmt		-	1.17	-
	MATEpe	5	8865.2	1.91	-
	MATEmt		496.29	1.12	-
	MATEpe	20	4800.73	2.19	-
	MATEmt		260.95	0.96	<0.44 - 1.84>
	MATEpe	40	3439.26	2.50	-
	MATEmt		186.28	0.90	<0.6 - 1.3>
	MATEpe	1	-	1.84	-
	MATEmt		-	0.94	-
	MATEpe		6200.88	1.93	-

Table 10.4: Overall Warmup Results *(continued)*

Benchmark	VM	Iterations	Sd	OF	Confidence
DeltaBlueT	MATEmt	5	279.92	0.94	-
	MATEpe	20	3873.48	2.03	<0.21 - 80.93>
	MATEmt		142.02	0.97	<0.43 - 2.27>
	MATEpe	40	2838.88	2.18	<0.39 - 35.55>
	MATEmt		100.78	0.98	<0.63 - 1.53>
JsonT	MATEpe	1	-	1.91	-
	MATEmt		-	1.00	-
	MATEpe	5	8755.98	2.56	<1.05 - 12.14>
	MATEmt		95.05	1.00	<0.97 - 1.03>
	MATEpe	20	5037.26	2.85	<2.1 - 4.06>
	MATEmt		52.09	1.00	<0.99 - 1.01>
	MATEpe	40	3659.08	2.99	<2.51 - 3.61>
	MATEmt		37.27	1.00	<1 - 1>
QuickSortT	MATEpe	1	-	1.46	-
	MATEmt		-	0.96	-
	MATEpe	5	1552.71	4.99	-
	MATEmt		127.85	0.97	<0.74 - 1.3>
	MATEpe	20	1412.73	5.67	-
	MATEmt		64.21	0.98	<0.92 - 1.05>
	MATEpe	40	1091.58	5.49	<2.84 - 17.98>
	MATEmt		45.44	0.98	<0.95 - 1.01>

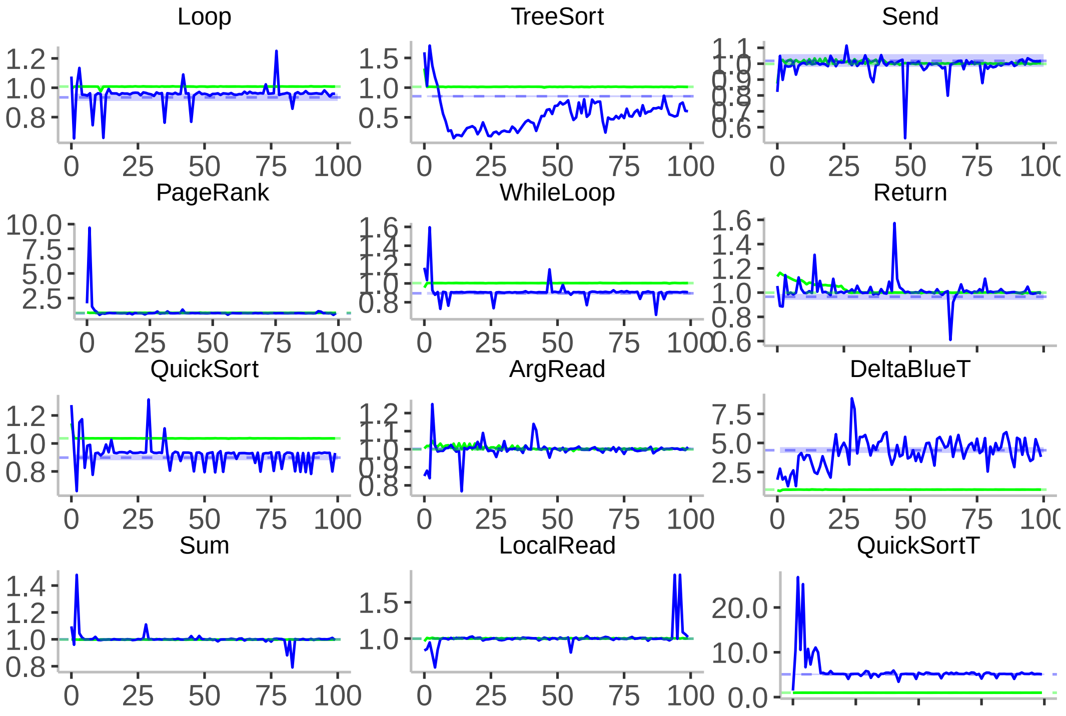

Figure 10.2: The first 100 iterations of a subset of the benchmarks in MATEpe and MATEmt normalized to their corresponding SOM versions.