Université des Sciences et Technologies de Lille – Lille 1

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Sista: a Metacircular Architecture for Runtime Optimisation Persistence

THÈSE

présentée et soutenue publiquement le 15 Septembre 2017

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

Clément Béra

Composition du jury

Président :	Theo D'Hondt
Rapporteur :	Gaël Thomas, Laurence Tratt
Examinateur :	Elisa Gonzalez Boix
Directeur de thèse :	Stéphane Ducasse
Co-Encadreur de thèse :	Marcus Denker

Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022
INRIA Lille - Nord Europe

Numéro d'ordre: XXXXX

Acknowledgments

I would like to thank my thesis supervisors Stéphane Ducasse and Marcus Denker for allowing me to do a Ph.D at the RMoD group, as well as helping and supporting me during the three years of my Ph.D.

I thank the thesis reviewers and jury members Gaël Thomas, Laurence Tratt for kindly reviewing my thesis and providing me valuable feedback. I thank the jury members Elisa Gonzales Boix and Theo D'Hondt.

I would like to express my gratitude to Eliot Miranda for his first sketch of Sista and his support during the three years of my Ph.D.

I would like to thank Tim Felgentreff for his evaluation of Sista using the Squeak speed center.

For remarks on earlier versions of this thesis, I thank, in addition to my supervisors, Guillermo Polito, Kavesseri Krishnan Subramaniam and Damien Cassou.

Abstract

Most high-level programming languages run on top of a virtual machine (VM) to abstract away from the underlying hardware. To reach high-performance, the VM typically relies on an optimising just-in-time compiler (JIT), which speculates on the program behavior based on its first runs to generate at runtime efficient machine code and speed-up the program execution. As multiple runs are required to speculate correctly on the program behavior, such a VM requires a certain amount of time at start-up to reach peak performance. The optimising JIT itself is usually compiled ahead-of-time to executable code as part of the VM.

The dissertation proposes Sista, an architecture for an optimising JIT, in which the optimised state of the VM can be persisted across multiple VM start-ups and the optimising JIT is running in the same runtime than the program executed. To do so, the optimising JIT is split in two parts. One part is high-level: it performs optimisations specific to the programming language run by the VM and is written in a metacircular style. Staying away from low-level details, this part can be read, edited and debugged while the program is running using the standard tool set of the programming language executed by the VM. The second part is low-level: it performs machine specific optimisations and is compiled ahead-of-time to executable code as part of the VM. The two parts of the JIT use a well-defined intermediate representation to share the code to optimise. This representation is machine-independent and can be persisted across multiple VM start-ups, allowing the VM to reach peak performance very quickly.

To validate the architecture, the dissertation includes the description of an implementation on top of Pharo Smalltalk and its VM. The implementation is able to run a large set of benchmarks, from large application benchmarks provided by industrial users to micro-benchmarks used to measure the performance of specific code patterns. The optimising JIT is implemented according to the architecture proposed and shows significant speed-up (up to 5x) over the current production VM. In addition, large benchmarks show that peak performance can be reached almost immediately after VM start-up if the VM can reuse the optimised state persisted from another run.

Résumé

La plupart des langages de programmation de haut niveau s'exécutent sur une machine virtuelle (VM) pour être indépendant du hardware utilisé. Pour atteindre de hautes performances, la VM repose généralement sur un compilateur à la volée (JIT), qui spécule sur le comportement du programme basé sur ses premières exécutions pour générer à la volée du code machine efficace et accélérer l'exécution du programme. Étant donné que plusieurs exécutions sont nécessaires pour spéculer correctement sur le comportement du programme, une telle VM nécessite un certain temps au démarrage pour atteindre les performances maximales. Le JIT est habituellement compilé en code exécutable avec le reste de la VM avant sa première utilisation.

La thèse propose Sista, une architecture pour un JIT, dans laquelle l'état optimisé de la VM peut être persisté entre plusieurs démarrages de la VM et le JIT s'exécute dans le même environnement d'exécution que le programme exécuté. Pour ce faire, le JIT est divisé en deux parties. Une partie est de haut niveau: elle effectue des optimisations spécifiques au langage de programmation exécuté par la VM et est méta-circulaire. Sans connaissances des détails de bas niveau, cette partie peut être lue, éditée et déboguée pendant le fonctionnement du programme en utilisant les outils de développement du langage de programmation exécuté par la VM. La deuxième partie est de bas niveau: elle effectue des optimisations spécifiques au hardware utilisé et est compilée en code exécutable, au sein de la VM, avant sa première utilisation. Les deux parties du JIT utilisent une représentation intermédiaire bien définie pour échanger le code à optimiser. Cette représentation est indépendante du hardware utilisé et peut être persistée entre plusieurs démarrages de la VM, ce qui permet à la VM d'atteindre rapidement les performances maximales.

Pour valider l'architecture, la thèse inclus la description d'une implémentation utilisant Pharo Smalltalk et sa VM. L'implémentation est évaluée par rapport à différents indices de performance, incluant l'exécution de programme utilisés en entreprise et de petits programmes utilisés pour mesurer la performance d'aspects spécifiques de la VM. Le JIT est implémenté selon l'architecture proposée et permet d'exécuter le programme jusqu'à 5x plus vite que la VM en production aujourd'hui. En outre, les indices de performance montrent que les performances maximales peuvent être atteintes presque immédiatement après le démarrage de la VM si cette dernière peut réutiliser l'état optimisé d'une autre exécution.

Contents

List of Figures

CHAPTER 1

Introduction

Contents

1.1 Context

Object-oriented languages have been one of the most popular programming languages for the past decades. Many high-level object-oriented programming languages run on top of a virtual machine (VM) which provides certain advantages from running directly on the underlying hardware. The main advantage is platform-independence: a program running on top of a VM can run on any processor and operating system supported by the VM without any changes in the program. In this thesis, the term VM is used to discuss about virtual machines for high-level programming languages, as opposed to operating system VMs which are not discussed at all.

High-performance virtual machines. High performance VMs, such as Java HotSpot [Paleczny 2001] or Javascript VMs like V8 [Google 2008] achieve high performance through just-in-time compilation techniques: once the VM has detected that a portion of code is frequently used (a *hot spot*), it recompiles it on-the-fly with speculative optimisations based on previous runs of the code. If usage patterns change and the code is not executed as previously speculated anymore, the VM dynamically deoptimises the execution stack and resumes execution with the unoptimised code.

Such performance techniques allow object-oriented languages to greatly improve their peak performance. However, a warm-up time is required for the VM

to correctly speculate about frequently used patterns. This warm-up time can be problematic for different use-cases (distributed applications with short-lived slaves, code in web pages, etc.).

Originally VMs were built in performance oriented low-level programming languages such as C. However, as the VMs were reaching higher and higher performance, the complexity of their code base increased and some VMs started to be written in higher-level languages as an attempt to control complexity. Such VMs were written either in the language run by the VM itself [Ungar 2005, Wimmer 2013, Alpern 1999] or in domain specific languages compiled to machine code through C [Rigo 2006, Ingalls 1997].

Existing design: Aosta. An optimising JIT design [Miranda 2002] for Smalltalk [Goldberg 1983], called Aosta (**A**daptive **O**ptimisations **S**mall**T**alk **A**rchitecture), emerged in the early 2000s. Aosta was designed by Eliot Miranda with contributions from Paolo Bonzini, Steve Dahl, David Griswold, Urs Hölzle, Ian Piumarta and David Simmons. The design is the convergence of multiple ideas to ease the development of the optimising JIT, to ensure that the maintainance and evolution cost of the resulting implementation is reasonnable and to attract contributors from the community.

One of the key ideas of Aosta is to split the optimising JIT in two parts, as shown on Figure 1.1. The first part, the high-level part, may deal with Smalltalk-specific optimisation and compiles to well-specified platform independent instructions (bytecodes). The second part, the low-level part, can translate such instructions into machine code, performing machine-specific optimisations.

The high-level part can be written in Smalltalk entirely, in a metacircular style. As most of the Smalltalk community has high skills in Smalltalk but little skills in low-level programming, the design aims here to allow the community to contribute to a project in Smalltalk doing Smalltalk-specific optimisations, improving performance while staying away from low-level details and machine-specific optimisations.

In addition, the high-level part of the JIT generates platform-independent optimised bytecode methods. Bytecode methods can already be persisted across multiple start-ups in Smalltalk using snapshots. This design allows therefore to persist optimised code, in the form of optimised bytecode methods, to avoid most of the warm-up time present in many modern VMs.

According to other VM implementors[1], it seems that language-specific optimisations (in this case Smalltalk-specific optimisations) are more important than machine-specific optimisations for performance. The Aosta design allows therefore to push most of the complexity from a low-level language, in which the exist-

[1] We discussed with developers from the V8 team.

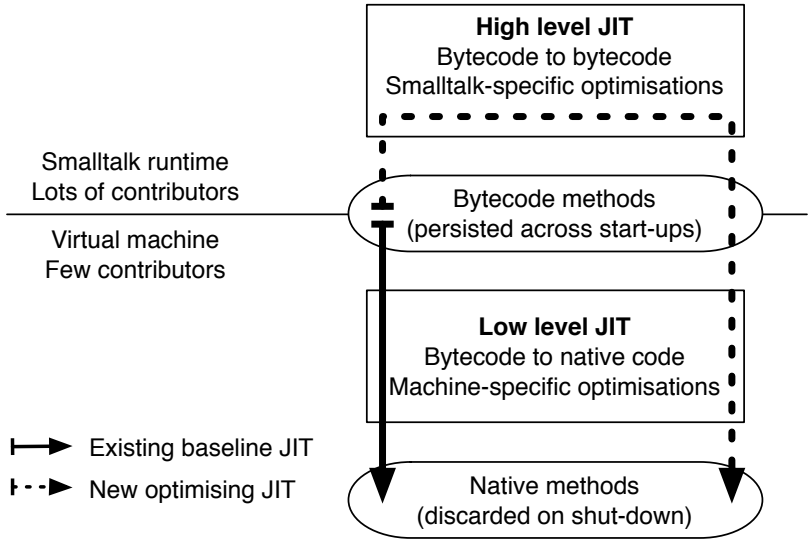


Figure 1.1: JIT compilation model design

ing Smalltalk VMs are implemented, to Smalltalk.

Another idea is then to reuse the existing baseline JIT, already present in the existing Smalltalk VMs, as the low-level part. Reusing the existing baseline JIT as a back-end for the optimising JIT means there is only one code-base to maintain and evolve for all the low-level aspects of both JITs. To ease the implementation of this design, the interface between the two parts of the optimising JIT is conceived as an extended bytecode set (the existing bytecode set with the addition of new operations used only by optimised code). This way, the existing baseline JIT already supporting the existing bytecode set would "just" needs to be slightly extended to support the new operations.

Some aspects of the design were considered, analyzed and discussed very carefully by several VM experts, making the design attractive and interesting. However, the overall design was incomplete so it was unclear how multiple parts of the system would work, especially, as no one really knew how to design and implement the high-level part of the JIT nor if the design could work.

The work done during the Ph.D started from the Aosta proposal: the goal was to complete the design and propose an implementation. The resulting architecture and implementation, presented in the thesis, is Sista (**S**peculative **I**nlining **S**mall**T**alk **A**rchitecture). The design of Sista is largely inspired from Aosta, but a working implementation to validate different aspects of the design and able to run a large suite of benchmarks is provided. Multiple aspects of Sista are different from existing VMs, such as the split in the optimising JIT or the ability to persist optimised

bytecode methods across start-ups. The pros and cons of these differences are discussed in the thesis.

Pharo programming language and community. The Aosta proposal was written for Smalltalk. One major feature of the design is the ability to persist bytecode methods across multiple start-ups. In the proposal, bytecode methods are persisted through snapshots: Snapshots allow the program to save the heap (including bytecode methods, which are normal objects in Smalltalk) in a given state, and the virtual machine can resume execution from this snapshot later. We wanted to retain this aspect of the design when working on Sista.

Snapshots are available in multiple object-oriented languages, such as Smalltalk and later Dart [Annamalai 2013]. However, they are widely used mainly in Smalltalk: in the normal development workflow, a Smalltalk programmer uses snapshots to save his code and deployment of production applications is typically done from a snapshot. For this reason, we used Smalltalk for the implementation of the Sista architecture.

As of today, multiple Smalltalk dialects are available, from commercial Smalltalk with expensive licences to open-source versions. In the thesis we focus on the Smalltalk dialect named Pharo [Black 2009], a fork of another Smalltalk dialect named Squeak [Black 2007] made by the original Smalltalk-80 implementors. We picked this dialect for two main reasons. First, both the VM and the language are under the MIT licence, allowing to read, edit and use the code base without any licence cost. Second, the community around Pharo is very active and eager to test and use new features.

In Pharo, everything is an object, including classes, bytecoded versions of methods or processes. It is dynamically-typed and every call is a virtual call. The VM relies on a bytecode interpreter and a baseline just-in-time compiler (JIT) to gain performance. Modern Smalltalk dialects directly inherit from Smalltalk-80 [Goldberg 1983] but have evolved during the past 35 years. For example, real closures and exceptions were added.

As Pharo is evolving, its VM, the Cog VM [Miranda 2008], is improving. For example, a modern memory manager was added over the past few years, improving performance and allowing the VM to use a larger amount of memory. The open-source community is now looking for new directions for VM evolutions, including better VM performance. Compared to many high performance VMs, the Pharo VM is not as efficient because it lacks an optimising JIT with speculative optimisations. The optimising JIT is usually one of the most complex parts of high performance VMs. As the Pharo community has a limited amount of resources to the maintain and evolve the VM, the idea is to design the optimising JIT in a way where open-source contributors can get involved in the maintainance and evolution tasks.

Many people in the community have high skills in object-oriented programming, especially Pharo development, while few people have skills in low-level programming such as assembly code or C. Hence, the community on average understands much more Smalltalk programs than low-level programs. Assuming one is more likely to contribute to a program one can understand, the logical choice is to design the optimising JIT in Smalltalk.

The existing production VM is written in a subset of Smalltalk [Ingalls 1997], called Slang, compiling through C to machine code to generate the production VM. Hence, two directions could be taken to write the optimising JIT in Smalltalk. On the one hand, the optimising JIT could be written in Slang, the existing subset of Smalltalk, like the existing VM. On the other hand, it could be written in the complete Smalltalk language, with a design similar to the metacircular VMs [Ungar 2005, Wimmer 2013, Alpern 1999]. Compared to C and assembly code, Slang tends to abstract away machine concepts to leverage the development experience closer to Smalltalk. However, an important part of the community does not contribute to the VM because its code-base is not available in the base system (it has been compiled to an executable ahead-of-time) and because they do not entirely understand the remaining low-level aspects. For this reason, writting the optimising JIT in the complete Smalltalk language seems to be the best option.

To conclude, the Pharo community is looking for better VM performance and the next step to improve the performance of the existing VM is to add an optimising JIT.

1.2 Problem

The overall direction of the thesis is to prove that Sista, derived from Aosta, is a viable and relevant architecture to write an optimising JIT. Two specific aspects of Sista, the metacircular high-level part of the optimising JIT and the persistence of optimised code across VM start-ups are then analysed carefully.

In the Sista design, the optimising compiler is running in the same runtime as the running application. As the optimising JIT is written in the optimised language, it may be able to optimise its own code. This behavior may lead to strange interactions between multiple parts of the runtime, leading to performance loss or crashes. The Graal compiler [Duboscq 2013] has a similar design to what we are trying to build. It can run on top of the Java Hotspot VM as an alternative optimising JIT. However, the development team avoids most of these problems by keeping part of the deoptimisation logic and the stack analysis to determine what method to optimise in the Hotspot VM and not in the Java runtime. Others problems are avoided by running Graal in different native threads than the running application.

In most existing VMs, the optimised code is not persisted across multiple start-

ups, making difficult the persistence of green threads unless all stack frames present in their execution stack are deoptimised. As we implemented Sista and code was starting to get optimised and executed, we analysed the interaction between optimising JITs and Smalltalk-style *snapshots*. In Smalltalk, a normal programmer regularly takes a snapshot, a memory dump of all the existing objects, to save the running system state. By default, the Smalltalk VM starts-up by resuming execution from a snapshot, restoring all the object states and resuming all running green threads. Each green thread has its own execution stack, which may refer to optimised code. With the bytecode to bytecode optimisation design, the persistence of running green threads, including the persistence of optimised code they refer to, is possible across multiple start-ups.

Research problems. The thesis focuses on three aspects:

- *Optimising JIT architecture:* What is a good architecture for an optimising JIT running in the same runtime as the optimised application on top of a non-optimising VM?

- *Metacircular optimising JIT:* In the context of an optimising JIT written in the single-threaded language it optimises, can the JIT optimise its own code at runtime and if so, under which constraints?

- *Runtime state persistence:* How to persist the runtime state across multiple VM start-ups, including the running green threads and the optimised code?

Supervisors. In the thesis, I use the term "we" to discuss about my supervisors, the people I worked with and myself. This includes mainly Stéphane Ducasse, Marcus Denker and Eliot Miranda, but also the different persons I worked with on specific aspects of Sista.

Implementation-wise, Eliot Miranda[2] and I did over 99% of the implementation to get Sista working. Section 5.3 details for each evolution done to the Pharo runtime which one of us did the work. The most complex component of Sista is by far the optimising compiler, which I wrote myself entirely.

Publication-wise, in addition to the authors mentioned in each paper, my research supervisors Stéphane Ducasse and Marcus Denker helped me consistently and reliably to produce relevant papers.

[2]Eliot Miranda is the maintainer and main implementor of the current production VM.

1.3 Contributions

The thesis introduces Sista (**S**peculative **I**nlining **S**mall**T**alk **A**rchitecture). Sista features an optimising JIT written in Smalltalk running on top of the existing Pharo VM. The optimising JIT is running in the same runtime as the optimised application. Sista is able to persist the runtime state of the program across multiple start-ups.

The main contributions of this thesis are:

- An optimising JIT running on top of the existing production virtual machine, showing 1.5x to 5x speed-up in execution time.
- A bytecode set solving multiple existing encoding limitations.
- A language extension: each object can now be marked as read-only.
- A new implementation of closures, both allowing simplifications in existing code and enabling new optimisation possibilities.

1.4 Outline

- Chapter 2 defines the terminology and presents existing production and research virtual machines relevant in the context of the thesis.
- Chapter 3 discusses the existing Pharo runtime as Sista is built on top of it.
- Chapter 4 details Sista and Chapter 5 discuss the evolutions done on the Pharo runtime to have the Sista architecture working.
- Chapters 6 and 7 discuss the architecture in the context of metacircular optimising JITs and the runtime state persistence.
- Chapter 8 evaluates Sista by comparing the performance of the runtime in multiple contexts, showing that the Sista runtime is going up to 80% faster than the current production Pharo VM.
- Chapter 9 details the future work that could be relevant based on this dissertation.

1.5 Thesis and published papers

In the thesis, the focus is on Sista. However, during the Ph.D I worked on other topics, always related to VMs but not necessarily to Sista, leading to publications.

I did not detail such work to keep the thesis concise and structured. This section lists all my publications (the publications in parentheses are waiting for approval):

Conferences and journals:

1. Eliot Miranda and Clément Béra. A Partial Read Barrier for Efficient Support of Live Object-oriented Programming. In International Symposium on Memory Management, ISMM'15, 2015.

2. Clément Béra, Eliot Miranda, Marcus Denker and Stéphane Ducasse. Practical Validation of Bytecode to Bytecode JIT Compiler Dynamic Deoptimization. Journal of Object Technology, JOT'16, 2016.

3. Nevena Milojković, Clément Béra, Mohammad Ghafari and Oscar Nierstrasz. Mining Inline Cache Data to Order Inferred Types in Dynamic Languages. Accepted with minor revisions in Science of Computer programming, SCP'17, 2017.

4. Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker and Stéphane Ducasse. Sista: Saving Optimized Code in Snapshots for Fast Start-Up. Submitted to International Conference on Managed Languages & Runtimes (ManLang, formerly PPPJ), ManLang'17, 2017.

Workshops:

5. Clément Béra and Marcus Denker. Towards a flexible Pharo Compiler. In International Workshop on Smalltalk Technologies, IWST'13, 2013.

6. Clément Béra and Eliot Miranda. A bytecode set for adaptive optimizations. In International Workshop on Smalltalk Technologies, IWST'14, 2014.

7. Nevena Milojković, Clément Béra, Mohammad Ghafari and Oscar Nierstrasz. Inferring Types by Mining Class Usage Frequency from Inline Caches. In International Workshop on Smalltalk Technologies, IWST'16, 2016.

8. Clément Béra. A low Overhead Per Object Write Barrier for the Cog VM. In International Workshop on Smalltalk Technologies, IWST'16, 2016.

9. Sophie Kaleba, Clément Béra, Alexandre Bergel, and Stéphane Ducasse. Accurate VM profiler for the Cog VM. Submitted to International Workshop on Smalltalk Technologies, IWST'17, 2017.

CHAPTER 2

Optimising Just-in-time compiler architectures

Contents

The thesis focuses on the design and the implementation of an optimising JIT architecture for Pharo. The main goals of this architecture are to write the optimising JIT in Pharo itself and to have it running in the same runtime as the optimised application on top of the existing VM. The following paragraphs introduce briefly the need of an optimising JIT for performance and how an optimising JIT improves performance of run programs. Concrete examples and references are present in the context of the two most popular optimising JIT architecture in Section 2.2.2 and 2.3.2, which we call respectively the function-based architecture and the tracing architecture.

Standard object-oriented languages feature dynamic dispatch. This feature is typically present in the form of virtual calls: the function to activate for each virtual call depends on information available at runtime but not at compile-time. Because of dynamic dispatch, it is difficult for an ahead-of-time compiler to optimise efficiently the code to execute. This problem is especially important for languages where virtual calls are very common. In our case, in Pharo, every call is a virtual call.

To efficiently optimise code in a language featuring dynamic dispatch, one solution is to use an optimising JIT. A VM featuring an optimising JIT executes a given code snippet through different phases. The first runs of the code snippet are done through a slow execution path, such as an interpreter, which collects information about the running program while executing it. Once the code snippet has been run a significant number of times, the optimising JIT recompiles the code snippet

at runtime to optimised native code. The compiler optimisations are directed by the runtime information collected during the first runs. Further uses of the same code snippet can be executed using the optimised version. We call each different way the VM can execute the same code snippet a different *tier*.

Multiple tiers. As an optimising JIT requires runtime information to direct the compiler optimisations, high-performance VMs are implemented with at least two tiers. One tier, slow to execute code, is used for the first runs of a code snippet to collect runtime information. The other tier requires both runtime information and compilation time to generate optimised native code, but the resulting execution should be faster[1]. Conceptually, a high-performance VM can be implemented with many tiers: each tier requires more compilation time than the previous tier but the resulting generated native code is faster.

The tier concept is summarized in Figure 2.1 with a theoretical VM using two tiers. The first tier is an interpreter: it requires no compilation time and takes 0.5ms to execute the given code snippet. Once the code snippet has been executed a thousand time since the last VM start-up, the optimising JIT kicks in and generates optimised native instructions using runtime information collected during the interpreter runs. The thousand and first run requires 5 ms of compilation time to generate the optimised version. However, once the optimised version is generated, subsequent runs of the same code snippet are much faster, taking 0.1 ms instead of 0.5 ms.

<table>
<tr><th>Run Number</th><th>Tier</th><th>Compilation time</th><th>Execution time</th></tr>
<tr><td>1 to 1000</td><td>1 interpreter</td><td>0 ms</td><td>.5 ms / run</td></tr>
<tr><td>1001</td><td rowspan="2">2 optimising JIT</td><td>5 ms</td><td rowspan="2">.1 ms / run</td></tr>
<tr><td>1002 +</td><td>0 ms</td></tr>
</table>

Figure 2.1: Execution of a frequently used code snippet.

Optimising JIT architectures. Two main architectures are widely used to design an optimising JIT. One optimising JIT architecture [Hölzle 1994a], historically the first one invented, attempts to boost performance by optimising frequently used functions. Such optimising JIT generates native code snippets corresponding to

[1]The resulting execution is theoretically always faster, but in practice, unfortunately, it is slower in some uncommon cases.

optimised functions. We call this architecture the *Function-based architecture* and we describe it in Section 2.2. The second architecture focuses on the optimisation of linear sequences of frequently used instructions. We call this architecture the *Tracing architecture*. Typically, tracing JITs optimise the common execution path of one iteration of each frequently used loop. This second architecture is detailed in Section 2.3. As Sista is more similar to the function-based architecture, Section 2.2 is more detailed than the other one.

Research problems. In the context of the design and implementation of the optimising JIT for Pharo, the thesis focuses on two aspects:

- *Metacircular optimising JITs:* Optimising JITs can be written in different programming languages, including the language they optimise. In the latter case, it may be possible for the JIT to optimise its own code. Such aspects are discussed in Section 2.4.

- *Runtime state persistence:* Most modern VMs always start-up an application with only unoptimised code. The application then needs a certain amount of time, called *warm-up time*, to reach peak performance. Warm-up time is a problem if the application needs high-performance immediately. Existing solutions for this problem are detailed in Section 2.5.

Closed-source VMs. This chapter tries to discuss the main production and research open-source VMs. Specific closed-source VMs are described as they are relevant in the context the thesis. However, many closed-source VMs are ignored as it is difficult to get reliable and free-to-share information about them, especially if no publications exist on a specific aspect of the VM.

Smalltalk VMs. As Pharo is a Smalltalk dialect, it is relevant to investigate the designs of other Smalltalk VMs. However, commercial Smalltalk VMs in production today are closed-source and do not feature optimising JITs so we do not discuss them. In the 90s, the Self VM [Hölzle 1994a] and the animorphic VM for Strongtalk [Sun Microsystems 2006] were able to execute Smalltalk code using an optimising JIT. Those VMs are briefly discussed but these VMs are not actively maintained nor used in production.

2.1 Terminology

This section clarifies specific terms to avoid confusion.

Functions. In the thesis we use the term *function* to refer to executable code which corresponds in practice to a method or a closure. More specifically, we distinguish *virtual functions*, or v-functions, which can be executed by a virtual machine (in our case, bytecode version of functions) and *native functions*, or n-functions, the native code version of a function executed by a specific processor.

Frames. We discuss VMs using a hybrid runtime where v-functions can be executed either through a v-function interpreter or by executing the corresponding n-function generated by a JIT from the v-function. On the one hand, we call *virtual frame* or v-frame a stack frame used by the v-function interpreter. On the other hand, we call *native frame* or n-frame a stack frame used by the execution of a n-function. V-frames have typically a machine-independent representation and all the values used by the execution stored inside the frame, while n-frames may have a machine-dependent representation and may have some values in registers.

Tiered architecture. One of the most common high-performance VM architectures is the tiered architecture: the first few executions of v-functions are performed by an interpreter and subsequent executions fall into the JIT infrastructure, composed of multiple tiers. Each JIT tier requires more time to compile the v-function to n-function than the previous tier, but the resulting n-function is more efficient. In many VMs, there are two JIT compiler tiers. The first tier is called the *baseline JIT*. It translates quickly v-functions to n-functions with a limited number of optimisations. The baseline JIT typically generates n-functions with inline caches to collect type information. The other tier is called the *optimising JIT*. It translates v-functions to optimised n-functions with speculative optimisations, based on the runtime information collected on n-functions generated by the baseline JIT[2].

Sista. *Sista* (**S**peculative **I**nlining **S**mall**T**alk **A**rchitecture) is the name of the architecture detailed in the thesis. As the architecture has notable differences from the standard tiered architecture, the two runtime compilers are not really a baseline JIT and an optimising JIT. We call them by their project name in the thesis. The first runtime compiler is called *Scorch* and compiles v-functions to optimised v-functions using speculative optimisations. Scorch is written in plain Smalltalk. The second runtime compiler is called *Cogit* and compiles v-functions to n-functions. Cogit can be used alone as the baseline JIT, or as a back-end for Scorch. In the latter case, the pair of Scorch and Cogit forms an optimising JIT. Cogit is written in a restrictive Smalltalk compiled ahead-of-time to an executable as part of the VM.

[2] A common three tiers implementation is described here, but some VMs have a different number of tiers (This is detailed later in Section 2.2.3).

In this context, both v-functions and n-functions can have an optimised version. We therefore used the term v-function to discuss all v-functions (optimised or not), and specify optimised v-function and unoptimised v-function when needed. Similarly, for frames, we use v-frame to discuss v-frames in general, and specify optimised v-frame and unoptimised v-frame when discussing a v-frame respectively representing the execution state of an optimised v-function or a unoptimised v-function. The same terminology is used with native (*n*-) than with virtual (*v*-).

Basic block. A *basic block* is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. A function is sometimes represented, in compiler intermediate representations, as a control flow graph, each node being a basic block and each vertice a control flow operation (conditional or unconditonal jump forward or backward).

2.2 Function-based architecture

The first optimising JIT architecture invented [Hölzle 1994a] was designed to generate optimised n-functions. From a given v-function, the optimising JIT performs a set of optimisations which includes inlining of other v-functions, and generates an optimised n-function. The section gives firstly an overview of the architecture and then discuss concrete implementations with references in Section 2.2.2. The last sections discuss specific aspects of the architecture.

2.2.1 Architecture overview

In many VMs following this architecture, three tiers are present. The following three paragraphs detail each tier, including how virtual calls are executed in each case.

Tier 1: V-function interpreter. The first tier is a virtual function interpreter. In most VMs, no compilation time is required at all to interpret a v-function[3] but the execution of the v-function by the interpreter is not very fast. Virtual calls are usually implemented with some sort of look-up cache to avoid computing the function to activate at each call. The interpreter tier does not necessarily collect runtime information.

[3]In most programming languages, v-functions are compiled ahead-of-time from source code. However, some VMs require compilation time for interpretation because the v-functions are not provided in a format the interpreter can execute (for example source code is provided).

Tier 2: Baseline JIT. The second tier is the baseline JIT, which generates from a single v-function a n-function with a very limited number of optimisations. Once compiled, the n-function is used to execute the function instead of interpreting the v-function. A small amount of time is spent to generate the n-function but the execution of the n-function is faster than the v-function interpretation. The n-function generated by the baseline JIT is introspected to collect runtime information if the function is executed enough times to be optimised by the next tier. The goal of the baseline JIT is therefore to generate n-functions providing reliable runtime information with limited performance overhead and not to generate the most efficent n-functions. Virtual calls are usually generated in machine code using inline caches [Deutsch 1984, Hölzle 1991]: each virtual call has a local cache with the functions it has activated, both speeding-up the execution and collecting runtime information for the next tier.

Tier 3: Optimising JIT. The last tier is the optimising JIT, which generates an optimised n-function. The optimising JIT uses runtime information such as the inline cache data to speculate on what function is called at each virtual call, allowing to perform inlining and to generate the optimised n-function from multiple v-functions. Such optimisations greatly speed-up the execution but are invalid if one of the compile-time speculation is not valid at runtime. In this case, the VM deoptimises the code and re-optimises it differently [Hölzle 1994b, Hölzle 1992]. The optimising JIT requires more time than the baseline JIT to generate n-functions, but the generated code is much faster. The execution of virtual calls is not really relevant in this tier as most virtual calls are removed through inlining and most of the remaining ones are transformed to direct calls.

Run Number	Compilation time	Execution time	Tier	Comments
1 to 6	0 ms	.5 ms / run	v-function interpreter	v-function interpretation
7	1 ms	.2 ms / run	baseline JIT	non optimised n-function generation & execution
8 to 10,000	0 ms			non optimised n-function execution
10,000	5 ms	.07 ms / run	optimising JIT	optimised n-function generation based on runtime information & execution
10,001 +	0 ms			optimised n-function execution

Figure 2.2: Execution of a frequently used v-function

Figure 2.2 shows the theoretical execution of a frequently used v-function over the three tiers. The first few runs are interpreted, each run taking 0.5 ms. The following run requires some compilation time for the baseline JIT to kick in, but the function is then executed 2.5 times faster and runtime information is collected. Lastly, after 10,000 runs, the optimising JIT takes a significant amount of time to generate an optimised n-function. The optimised n-function is executed three times faster than n-function generated by the baseline JIT.

2.2.2 Existing virtual machines

The first VM featuring this function-based architecture was the Self VM [Hölzle 1994a]. The Self VM had only two tiers, the baseline JIT and the optimising JIT.

The second VM built with this design was the Animorphic VM for the Strongtalk programming language [Sun Microsystems 2006], a Smalltalk dialect. This VM is the first to feature three tiers. The first tier is a threaded code interpreter hence interpretation requires a small amount of compilation time to generate threaded code from the v-function. The two other tiers are the same as in the Self VM. The animorphic VM has never reached production.

The Hotspot VM [Paleczny 2001] was implemented from the Self and animorphic VM code base and has been the default Java VM provided by Sun then Oracle for more than a decade. In the first versions of the Hotspot VM, two executables were distributed. One was called the client VM, which included only the baseline JIT and was distributed for applications where start-up performance matters. The other one was called the server VM, which included both JIT tiers, and was distributed for application where peak performance matters. Later, the optimising JIT was introduced in the client VM with different optimisation policies than the server version to improve the client VM performance without decreasing too much start-up performance. In Java 6 and onwards, the server VM became the default VM as new strategies allowed the optimising JIT to improve performance with little impact on start-up performance. Lastly, a single binary is now distributed for the 64 bits release, including only the server VM.

More recently, multiple Javascript VMs were built with a similar design. A good example is the V8 Javascript engine [Google 2008], used to execute Javascript in Google Chrome and Node JS. Other VMs, less popular than the Java and Javascript VMs are also using similar architectures, such as the Dart VM.

One research project, the Graal compiler [Oracle 2013, Duboscq 2013], is a function-based optimising JIT for Java that can be used, among multiple use-cases, as an alternative optimising JIT in the Hotspot VM.

2.2.3 Just-in-time compiler tiers

Many VMs featuring a function-based architecture in production nowadays have three tiers. The number of tiers may however vary from two to as many as the development team feels like. The following paragraphs discuss the reasons why the VM implementors may choose to implement a VM with two tiers, three tiers or more.

Engineering cost. Each new tier needs to be maintained and evolved accordingly to the other tiers. Hence, a VM having more tiers requires more engineering time for maintainance and evolutions. Any bug can come from any tier and bugs coming from only a single tier can be difficult to track down. Evolutions need to be implemented on each tier. To lower the VM maintenance and evolution cost, a VM needs to have the least number of tiers possible.

Minimum number of tiers. By design, the optimising JIT is the key component for high-performance and it needs runtime information from previous runs to generate optimised code. Hence, a VM with a function-based architecture requires at least two tiers. One tier, the non-optimising tier, is used for the first runs to collect statistical information and is typically implemented as an interpreter tier or a baseline JIT tier. The second tier, the optimising tier, generates optimised n-functions and is implemented as an optimising JIT. To perform well, the optimising tier has to kick in only if the function is used frequently (else the compilation time would not be worth the execution time saved) and the previous tier(s) must have executed the v-function enough time to have collected reliable runtime information. For this reason, the optimising tier usually kicks in after several thousands executions of the v-function by the previous tier(s).

Two non-optimising tiers. Many VMs feature two non-optimising tiers and one optimising tier. The non-optimising tiers are composed of an interpreter tier and a baseline JIT tier. These two tiers have different pros and cons and featuring both allows the VM to have the best of both worlds. There are three main differences between the two tiers: execution speed, efficiency of runtime information collection and memory footprint. The three differences are detailed in the next three paragraphs.

Execution speed. The interpreter tier is faster than the baseline JIT tier if the function is executed a very small number of times because there are not enough executions to outweight the baseline JIT compilation time. Figure 2.3 compares the speed of one to ten executions of the frequently used v-function from Figure 2.2. As interpreting the v-function takes 0.5 ms, the compilation by the baseline

JIT 1 ms and the execution of the n-function generated by the baseline JIT 0.2 ms, the interpreter is faster if the v-function is executed less than three times. However, if the function is executed between four times and 10,000 (at which point the optimising JIT kicks in), the baseline JIT tier is faster.

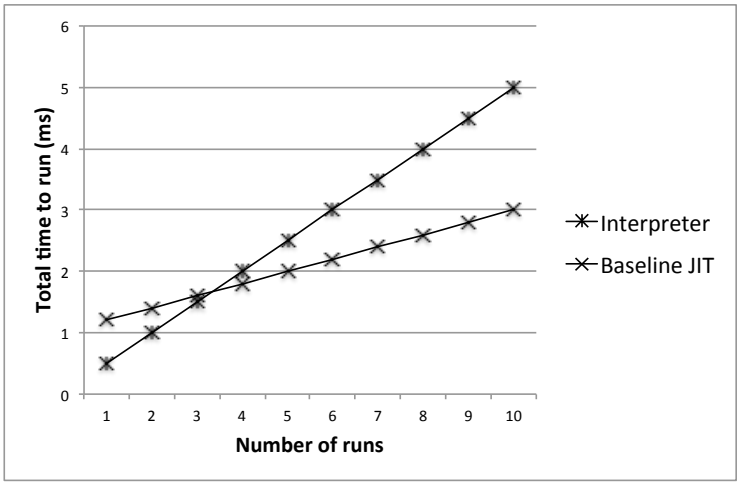


Figure 2.3: Time to execute a v-function in non-optimising tiers

Runtime information collection. One of the most relevant runtime information to collect is the function called at each virtual call. It is currently not possible to collect this information without overhead in an interpreter if the interpreter is written in a machine independent language. However, the inline cache technique [Deutsch 1984, Hölzle 1991] allows one to collect such information in a baseline JIT tier while speeding-up code execution.

Memory footprint. The interpreter does not require extra memory for the function it executes as only the v-function representation is needed. On the other hand, the baseline JIT requires memory to store the generated n-function for each v-function it executes. If many functions are executed once, not having the interpreter tier can lead to significant memory overhead.

Having both tiers allows the VM to have a lower memory footprint thanks to the interpreter tier. If the interpreter tier does not collect runtime information and is used only for the first few executions, the start-up performance is much better when both tiers are present than when one or the other is present alone. Runtime information can be collected with little overhead thanks to the baseline JIT tier. For these reasons, many VMs feature these two non-optimising tiers.

A good example of the pros and cons of multiple tiers is the evolution of the V8 Javascript engine [Google 2008]. In 2008, the first version was released featuring only the baseline JIT tier. The following year, the optimising JIT tier was added to improve performance. In 2016, the interpreter tier was added both to lower the memory footprint and to improve start-up performance.

In general, the two non-optimising tiers are kept as simple as possible to ease maintainance and evolutions. Only the third tier, the optimising JIT, may be more complex to be able to generate efficient n-functions.

More than three tiers. Adding more than three tiers is usually not worth it as it would mean additional maintenance and evolution cost. However, in specific languages such as Javascript where the start-up performance is critical, it can be worth it to have two optimising JIT tiers to increase start-up performance. The Javascript Webkit VM has four tiers since 2015 [Webkit 2015]. In this case, the VM team introduced two optimising JIT tiers after the interpreter and baseline JIT. One optimising JIT tier has smaller compilation time than the other one but produce less efficient n-function.

Independent compiler tiers. In most VMs, the baseline JIT and the optimising JIT are completely independent entities. Indeed, both JIT tiers are fundamentally different and it is difficult to share code between both tiers.

Baseline JIT. The baseline JIT has to be as simple as possible to limit the maintenance cost, simplicity is more important than generated code quality as most of the VM performance comes from the optimising JIT. The n-functions it generates need to be easily introspected to collect runtime information about the previous runs for the optimising JIT to direct compiler optimisations. The baseline JIT compilation time has to be very small.

The baseline JIT is typically implemented as a template-based engine [Deutsch 1984], generating a predefined sequence of native instructions for each virtual instruction. Template-based generation engines are relatively simple to implement and maintain. Templates are very convenient for native code introspection because the JIT knows the exact sequence of native instructions generated for each virtual instruction so it knows the exact bytes to read to extract runtime information. Lastly, template-based compilation is usually very efficient, providing low compilation time.

Optimising JIT. The optimising JIT is significantly different. It needs to generate n-functions as efficient as possible with a reasonable compilation time, but potentially much higher than the baseline JIT. The n-functions generated by

the optimising JIT are not introspected in most VMs, allowing the optimising JIT to generate the most efficient instructions. As any software project, complexity has to be controlled but it is usually worth to add complexity in the optimising JIT to allow it to generate more efficient code as it leads to overall better VM performance. The optimising JIT is typically implemented, as shown in Figure 2.4, by translating the v-function to a high-level intermediate representation to perform language-specific optimisations. It then transforms the representation to another intermediate representation, closer to native instructions, where machine-specific optimisations are performed. Lastly it generates native code.

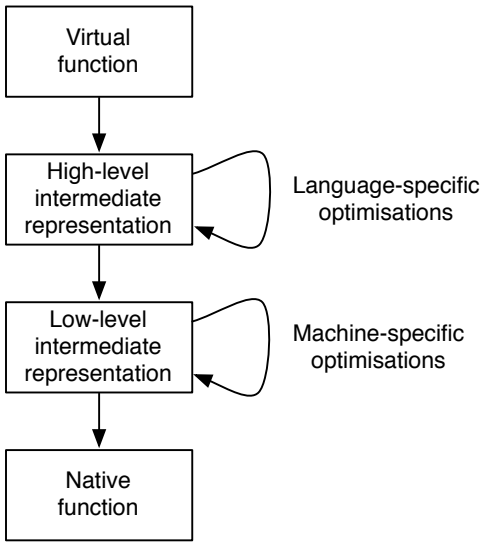


Figure 2.4: Classical optimising JIT architecture

Sharing code between compiler tiers. Because of the fundamental differences, most optimising JITs use a completely different code base than the baseline JIT they work with. However, there are some rare cases where part of the JIT compilers are shared between multiple tiers.

The first case is the Javascript Webkit VM [Webkit 2015]. As four tiers are present, it is possible to share portions of the compilers because some features are required in multiple tiers. For example, both the baseline JIT and the first-level optimising JIT requires the VM to be able to instrospect the generated machine code. In addition, both optimising JITs have optimisation logic in common allowing to share part of the optimisation pipeline. In this case, they share the high-level optimisations while the low-level optimisations are done in different back-ends.

The second case is related to VM extensions. The Javascript VMs are now

attempting to support, in addition to Javascript, an abstract assembly language called WebAssembly [Group 2015]. WebAssembly allows the programmer to compile ahead-of-time specific frameworks or libraries for use-cases difficult to optimise efficiently at runtime, such as real-time libraries. WebAssembly provides both abstract assembly code instructions and convenient instructions to interface WebAssembly with Javascript and the web page. In the V8 Javascript engine [Google 2008], the low-level intermediate representation of TurboFan, the optimising JIT of V8, is shared between the WebAssembly back-end and the optimisation path for Javascript code.

2.2.4 Concurrent compilation

The first optimising JIT, designed and implemented in Self [Hölzle 1994a], was done in a single-threaded environment. In this case, the optimising JIT had a limited time period to produce optimised code, and if the time period was not enough, the function was not optimised. Since the early 2000s, multi-threaded environments have become more common and many optimising JITs now perform optimisations concurrently to the application native thread(s) [Arnold 2000,Stadler 2012].

In most cases, not all the runtime compilations are however done concurrently. The baseline JIT is typically executed in the same native thread as the application. As it has very small compilation time, the compilation time overhead is usually not significant enough to justify concurrent compilation. When a frequently used portion of code is detected, the optimising JIT has to choose a function to optimise based on the current stack. This cannot be done concurrently as the stack needs to be introspected. Once the function to optimise is chosen, the optimisation of the function can be done concurrently. The optimising JIT has usually access to a pool of native threads which take functions to optimise in a compilation queue, optimises them and installs them. Further calls on such functions can use the optimised version installed. The optimising JIT may insert guards to ensure assumptions speculated at compile-time (such as the type of a specific variable) are valid at runtime. If one of the guard fails, the stack needs to be deoptimised to resume with non-optimised code. Deoptimisation of the stack is not done concurrently as the application requires the deoptimisation to be finished to resume execution.

2.2.5 Aosta technical report

Normally technical reports are not relevant enough to be mentioned, but as Sista is based on the Aosta technical report [Miranda 2002], it is definitely worth talking about it.

Aosta is a design sketch for an adaptive optimiser implemented in Smalltalk above a conventional Smalltalk virtual machine (a virtual machine featuring a base-

line JIT) with *minor* extensions. Adaptive optimisations are discussed in the sense of Urs Hölzle [Hölzle 1994a]. The sketch is far from complete, focusing on the interface between the optimiser and the virtual machine, hence outlining a potentially portable architecture where an optimiser written in Smalltalk can be hosted above a range of specific virtual machines. This portability is intended to allow the Smalltalk community to collaborate on the project without having to define and implement a common VM, with all the difficulties of migrating current systems to a new VM, allowing the community to apply the optimiser within the existing systems. Of course, this architecture still requires significant extensions to the execution machinery of existent VMs but these extensions amount to something far from a rewrite.

The sketch is then detailed in the context of HPS, the VisualWorks VM, which is a second generation implementation of Peter Deutsch's PS Smalltalk [Deutsch 1984]. The authors chose to describe the architecture with this VM as he is familiar with it and the sketch needed (according to the author) to be based on an existing VM to make it as real as possible. The sketch is expected to apply more broadly than just HPS, though until Sista no implementation was running so it has yet to be proven. The sketch was, in 2002, functioning as a specification for the HPS implementation.

2.3 Tracing architecture

The main alternative to the function-based architecture is the tracing architecture. Tracing JITs do not optimise entire functions but instead focus on optimising linear sequences of instructions. As most tracing JITs focus on the optimisation of loop iterations, we detail this case in this section. The section starts by providing an overview of the architecture and then discusses concrete implementation with references in Section 2.3.2.

2.3.1 Architecture overview

VMs with tracing JITs generate optimised native code only for the frequently used paths of commonly executed loops and interpret virtual instructions for the rest of the program. Tracing JITs are built on the following basic assumptions:

- Programs spend most of their runtime in loops.
- Several iterations of the same loop are likely to take similar code paths.

Typically, in VMs with tracing JITs, the first executions of a loop are done using a v-function interpreter. The interpreter profiles the code executed to detect

frequently used loops, usually by having a counter on each backward jump instruction that counts how often this particular backward jump is executed. When a hot loop is identified, the interpreter enters a special mode, called tracing mode. During tracing, the interpreter records a history of all the operations it executes during a single execution of the hot loop. The history recorded by the tracer is called a trace: it is a list of operations, together with their actual operands and results. Such a trace can be used to generate efficient native code. This generated machine code is immediately executable and can be used in the next iteration of the loop.

Being sequential, the trace represents only one of the many possible paths through the code. To ensure correctness, the trace contains a guard at every possible point where the path could have followed another direction, for example at conditional branches or virtual calls. When generating native code, every guard is turned into a quick check to guarantee that the path we are executing is still valid. If a guard fails, the execution immediately quits the native code and resumes the execution by falling back to the interpreter.

Aside from loops. Some tracing JITs are able to trace code aside from loops. In this case, profiling counters are added on functions to see if they are worth compiling or not.

2.3.2 Existing VMs

Tracing optimisations were initially explored by the Dynamo project [Bala 2000] to dynamically optimise native code at runtime. Its techniques were then successfully used to implement a JIT compiler for a Java VM [Gal 2006]. The technique was used in Mozilla's JavaScript VM from Firefox 3 to Firefox 11 [Gal 2009] until Mozilla removed it to replace it by a function-based JIT.

The most famous tracing JITs in production are certainly the ones generated from the RPython toolchain [Rigo 2006]. The RPython toolchain allows the generation of a tracing JIT for free if one writes a virtual function interpreter in RPython. The most popular example is Pypy [Rigo 2006, Bolz 2009], a Python VM using a tracing JIT through the RPython toolchain framework.

2.3.3 Sista and tracing JITs

Sista was not designed as a tracing JIT. There were two main reasons. First, the design was inspired from the Aosta proposal, which is a function-based architecture design. Second, we did not believe that optimising loop bodies would make sense in the context of Smalltalk and we will explain why in the second paragraph of Section 4.4.1.

2.4 Metacircular optimising Just-in-time compiler

An optimising JIT is implemented in a programming language and is able to optimise code from one or multiple programming languages. If the implementing language of the optimising JIT is one of the language it can optimise, is the optimising JIT able to optimise its own code?

The section starts by discussing the programming languages in which the optimising JITs are written. For the rare case where the implementing language of an optimising JIT is included in the languages the JIT can optimise, we detail if such optimisations are possible and used in production.

2.4.1 Implementation language

Historically, VMs have been implemented in low-level languages such as C++. Low-level languages are very convenient for multiple VM development tasks, such as direct memory access or optimisation of specific portion of the VM code for performance. The first optimising JITs, including the Self, Animorphic and Java Hotspot VMs [Hölzle 1994a, Sun Microsystems 2006] were written in C++. More recently, Javascript VMs such as V8 or Webkit [Webkit 2015] were still written in C++. As far as we know, there is no optimising JIT in production optimising C++ code, hence none of these JITs are able to optimise their own code.

Another approach is to use a high-level language compiled ahead-of-time to assembly code to write the optimising JIT. This approach is used by the RPython toolchain [Rigo 2006], where RPython is a restricted Python that can be compiled to native code through C. RPython was used to write Pypy's meta-tracing optimising JIT. In the case of Pypy, the JIT is able to optimise Python code, and as RPython is a subset of Python, the JIT is able to optimise its own code. However, in production, all the RPython code is compiled ahead-of-time to an executable binary. All the JIT code base is therefore translated to native code, and as the JIT cannot optimise native code, the JIT does not optimise itself in production.

Figure 2.5 shows the compilation of the production VM using RPython. The Core VM is written in RPython, and the RPython to C compiler generates C files from the RPython code. The final VM is compiled using a C compiler from the generated C files and additional C files for platform-specific code.

Metacircular VMs. Multiple research projects showed that it is possible to implement an entire VM in the programming language the VM runs. Such VMs are called metacircular VMs.

The Jalapeño project, now called Jikes RVM [Alpern 1999], was the first successful VM with such a design. Jikes RVM is a Java VM written in Java. On the Jikes RVM official page, it is written that the current version of the VM does not

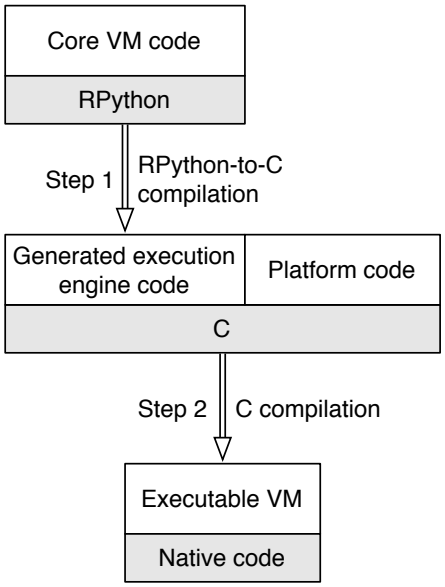


Figure 2.5: RPython VM executable generation

perform runtime optimisations based on runtime information. There is however a runtime compiler present in Jikes RVM [Arnold 2000] optimising the code more aggressively than the baseline JIT, but it does not seem to use runtime information to direct its optimisations.

Another Java VM written in Java was implemented in the late 2000s at Oracle, called Maxine VM [Wimmer 2013]. Maxine had multiple working optimising JITs. The most popular was extracted from the Maxine VM and is now known as the Graal compiler [Oracle 2013,Duboscq 2013]. In the case of Maxine, the optimising JIT is written in Java and is able to optimise its own code.

There were other attempts to implement metacircular VMs for other languages than Java. The Klein VM [Ungar 2005] is a Self VM written in Self and reportedly, in 2009, there was some work in the direction of an optimising JIT. The project does not seem however to be very active today and the optimising JIT is definitely not fully working. There were also several attempts to write a Smalltalk VM in Smalltalk. The last attempt, still active today, is Bee Smalltalk [Pimás 2014]. Unfortunately Bee Smalltalk is currently not open-source and it is not clear if an optimising JIT is present or not. Past attempts included Pinocchio [Verwaest 2010] and the Mushroom runtime [Wolczko 1987].

The last but not least project is the Truffle framework [Würthinger 2013]. Truffle is a framework allowing to write efficiently VMs for different programming languages. The Truffle runtime is built on top of Java's Hotspot VM, but the Graal

compiler is used as the optimising JIT instead of the Hotspot optimising compiler. Multiple VMs using the Truffle framework were implemented for different programming language in the past years. For each of them, the Graal compiler in the Truffle runtime can optimise both Java code and the programming language run. As Graal is written in Java, it can optimise its own code.

2.4.2 Optimising Just-in-time compiler optimising itself

Overall, very few optimising JITs are written in a programming language they can optimise. Even when they could optimise themselves, the VM development team may choose to compile the JIT code to native code ahead-of-time and the optimising JIT does not optimise itself in production. The main existing case where the optimising JIT is optimising its own code is the Graal optimising JIT. Graal can be used in different contexts. It was built as the Maxine VM optimising JIT. It is now mainly used as the optimising JIT of the Truffle runtime, as an alternative optimising JIT for Java Hotspot.

We detail here briefly how the Graal compiler optimises its own code when it is running as an alternative optimising JIT in the Java Hotspot VM. In this case, the Graal optimising JIT is written in Java while the rest of the VM, originally from the Java Hotspot VM, is written in C++. The application is running using multiple native threads and the Graal compiler is running in other native threads, concurrently.

When a frequently used portion of code is detected, the Hotspot VM chooses a function to optimise based on the current stack. The VM then puts the function to optimise in a thread-safe compilation queue. The Graal compiler native threads, running concurrently to the application, take functions to optimise from the compilation queue and generate an optimised function for each function in the queue. Hotspot provides APIs to extract runtime information from each unoptimised function to direct the compiler optimisation. Once the optimisation finished, the Graal compiler provides to Hotspot an optimised n-function with deoptimisation metadata. The Hotspot VM installs the optimised n-function. If one of the compilation-time assumption is invalid at runtime, the Hotspot VM is able to deoptimise the stack based on the deoptimisation metadata provided by the Graal compiler.

2.5 Runtime state persistence

In Sista, we persist the runtime state across multiple start-ups, including the optimised code but also the running green threads using optimised code. Persistence of running green threads with optimised code has not been done before to the best of our knowledge. In our case, we need to persist green threads as the normal

Smalltalk developer workflow requires it. It seems no other programming language with an optimising JIT has the same requirement so the running green threads are not persisted across start-ups. For this reason, we focus in this section on the persistence of optimised code between multiple start-ups.

One of the main problems with optimising JITs, compared to ahead-of-time compiler, is the start-up performance. As the optimising JIT needs runtime information to optimise code, usually thousands of unoptimised runs of a code snippet are required before reaching peak performance. This warm-up time can cause significant problems in specific short-lived applications, where most of the execution time is spent before reaching peak performance.

Because of these constraints, some object-oriented languages are compiled with an ahead-of-time compiler. Static analysis is performed over the code to guess what function is called at each virtual call. Applications for the iPhone are a good example where static analysis is used to pre-optimise the Objective-C application. The peak performance is lower than with a JIT compiler if the program uses a lot of virtual calls, as static analysis is not as precise as runtime information on highly dynamic languages. However, if the program uses few dynamic features (for example most of the calls are not virtual) and is running on top of a high-performance language kernel like the Objective-C kernel, the result can be satisfying.

Most object-oriented languages still choose to run on top of a VM with an optimising JIT. The section describes four existing techniques to improve start-up performance, including techniques related to optimised code persistence across start-ups.

Many tiers architecture. One solution to decrease warm-up time is to have many tiers in the function-based architecture. The idea is that code would be executed slowly the few first iterations, a bit faster the next iterations, faster after an certain number of optimisations, and so on. Instead of being slow for many iterations before being fast, the VM can this way have a very good trade off between compilation time, runtime information quality and code performance.

The best example is the Javascript Webkit VM [Webkit 2015]. A code snippet is:

1. Interpreted by a bytecode interpreter the first 6 executions.
2. Compiled to machine code at 7th execution, with a non-optimising compiler, and executed as machine code up to 66 executions.
3. Recompiled to more optimised machine code at 67th execution, with an optimizing compiler doing some but not all optimisations, up to 666 executions.
4. Recompiled to heavily optimised machine code at 667th execution, with all the optimisations.

At each step, the compilation time is greater but the execution time decreases. The many tiers approach (four tiers in the case of Webkit), allows the VM to have decent performance during start-up, while reaching high performance for long running code. However, this technique has a severe drawback: the VM team needs to maintain and evolve many different tiers.

Persisting runtime information. To quickly reach peak performance, one way is to save the runtime information, especially inlining decisions made by the optimising JIT. In [Sun Microsystems 2006], it is possible to save the inlining decisions of the optimising compiler in a separate file. The optimising compiler can then reuse this file to take the right inlining decision in subsequent start-ups. In [Arnold 2005], the profiling information of unoptimised runs is persisted in a repository shared by multiple VMs, so new runs of the VM can re-use the information to direct compiler optimisations.

Persisting machine code. In the Azul VM Zing [Systems 2002], available for Java, the official web site claims that "operations teams can save accumulated optimizations from one day or set of market conditions for later reuse" thanks to the technology called *Ready Now!*. In addition, the website precises that the Azul VM provides an API for the developer to help the JIT to make the right optimisation decisions.

As Azul is closed source, implementation details are not entirely known. However, word has been that the Azul VM reduces the warm-up time by saving machine code across multiple start-ups. If the application is started on another processor, then the saved machine code is simply discarded. It is very difficult to persist optimised native code across multiple start-ups due to position dependent code and low-level details, but with the example of Azul, we know it is possible.

Aside from Azul, the work of Reddi and all [Reddi 2007] details how they persist the machine code generated by the optimising JIT across multiple start-ups of the VM. JRockit [Oracle 2007], an Oracle product, is a production Java VM allowing to persist the machine code generated by the optimising JIT across multiple start-ups.

Preheating through snapshots. This paragraph discusses the persistence of the runtime state in snapshots in Dart and in a specifc Java VM.

Dart snapshots. The Dart programming language features snapshots for fast application start-up. In Dart, the programmer can generate different kind of snapshots [Annamalai 2013]. The Dart team added in 2016 two new kind of snapshots, specialized for iOS and Android application deployment.

Android. A Dart snapshot for an Android application is a complete representation of the application code and the heap once the application code has been loaded but before the execution of the application. The Android snapshots are taken after a warm-up phase to be able to record call site caches in the snapshot. The call site cache is a regular heap object accessed from machine code, and its presence in the snapshot allows one to persist type feedback and call site frequency.

iOS. For iOS, the Dart snapshot is slightly different as the platform does not allow JIT compilers. All reachable functions from the iOS application are compiled ahead-of-time, using only the features of the Dart optimising compiler that don't require dynamic deoptimisation. A shared library is generated, including all the instructions, and a snapshot that includes all the classes, functions, literal pools, call site caches, etc.

Cloneable VMs. In Java, snapshots are not available and used by default. However, Kawachiya and all describe in their work [Kawachiya 2007] extensions to a Java VM to be able to clone the state of a running Java VM in a similar way to snapshots. In this work, the cloned VM duplicates the heap but also the machine code generated by the different JIT tiers.

Conclusion

This chapter detailed existing solutions for our research problems, including the existing optimising JIT architectures, their implementation languages and how some VMs persist optimised code across multiple start-ups. The following chapter describes the existing Pharo runtime which was used as a starting point for our implementation.

CHAPTER 3

Existing Pharo Runtime

Contents

This chapter describes the Smalltalk dialect Pharo [Black 2009] and part of its implementation. The Sista architecture was originally designed to improve the performance of the Pharo VM by adding an optimising JIT. Some existing features and implementation details already present in Pharo impacted our design decisions. They are detailed in this chapter to help the reader understanding the design decisions explained in further chapters. The chapter is not meant to explain the whole existing implementation, but only the most relevant points for the thesis.

Pharo is a pure object-oriented language. Everything is an object, including green threads, classes, method dictionaries or virtual functions. It is dynamically-typed and every call is a virtual call. The virtual machine relies on a virtual function interpreter and a baseline JIT named *Cogit* to gain performance. Pharo directly inherits from Smalltalk-80 [Goldberg 1983] but has additional features such as real closures, exceptions or continuations.

The chapter successively describes some aspects of the Pharo VM, the interface with the language and the Pharo programming language.

3.1 Virtual machine

The Pharo VM is a variant of the Cog VM [Miranda 2008]. It relies on a v-function interpreter and Cogit, the baseline JIT, to gain performance.

Executable generation. Most of the existing VM, inheriting from the original Squeak VM [Ingalls 1997], is written in Slang, a subset of Smalltalk. Slang is compiled to C and then to native code through standard C compilers. The execution engine (the memory manager, the interpreter and the baseline JIT) are entirely written in Slang.

In addition to providing some abstractions over machine-specific-details, the slang code has two main advantages over plain C:

- *Specifying inlining and code duplication:* To keep the interpreter code efficient, one has to be very careful on what code is inlined in the main interpreter loop and what code is not. In addition, for performance, specific code may need to be duplicated. For example, the interpreter code to push a temporary variable on stack is duplicated 17 times. The 16 first versions are dedicated versions for temporary numbers 0 to 15, the most common cases, and are more efficient because of the use of constants. The 17th version is the generic version, which could be used on any temporary variable but is used in practice for temporary variable 16 and over. Slang allows one to annotate functions to direct Slang to C compilation, by duplicating or inlining specific functions. This way, when a VM developer writes a function, he writes it once and chooses the number of times the function is going to be duplicated or chooses to force the inlining of the function. This feature is very important for uncommon processors where available C compilers are often not as good at optimising C code as on mainstream processors.

- *Simulation:* As Slang is a subset of Smalltalk, it can be executed as normal Smalltalk code. This is used to simulate the interpreter and garbage collector behavior. The JIT runtime is simulated using both Slang execution and external processor simulators. Simulation is very convenient to debug the VM as all the Smalltalk debugging tools are available. In addition, the simulator state can be saved and duplicated, which is very convenient to reproduce quickly and many times the same bug happening from a specific runtime state.

The executable is generated in two steps as shown on Figure 3.1, similarly to the RPython toolchain [Rigo 2006]. The first step is to generate the two C files representing the whole execution engine written in Slang using the Slang-to-C compiler. During the second step, a C compiler is called to compile the execution engine and the platform-specific code written directly in C to the executable VM.

Baseline JIT. Cogit is currently used as the baseline JIT. It takes a v-function as input, generates a n-function and installs it. Cogit performs three main kinds of optimisation:

1. *Stack-to-register mapping:* As the v-functions are encoded using a stack-based bytecode set, values are constantly pushed and popped off the stack. To avoid this behavior, Cogit simulates the stack state during compilation. When reaching an instruction using values on stack, Cogit uses a dynamic

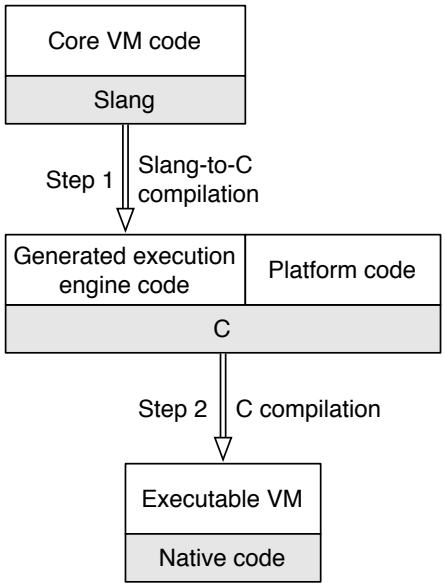


Figure 3.1: VM executable generation

template scheme to generate the native instructions. The simulated stack provides information such as which values are constants or already in registers. Based on this information, Cogit picks one of the available templates for the instruction, uses a linear scan algorithm to allocate registers that do not need to be fixed into specific concrete registers and generates the native instructions.

2. *Inline caches:* Each virtual call is compiled to an unlinked inline cache. During execution, the inline cache is relinked to a monomorphic, polymorphic or megamorphic inline cache [Deutsch 1984, Hölzle 1991] when new receiver types are met. The inline caches improve performance but also allows, through n-function introspection, to determine which types were met during previous runs of each virtual call site.

3. *Calling convention:* Cogit defines specific calling conventions for calls in-between n-functions. Typically, the receiver of the virtual call is always passed by register, and the arguments may or may not be passed by registers depending on how many there are. This is especially efficient to speed-up the inline cache logic and for primitive methods that have an assembly template available as they can directly use the values in registers.

Cogit provides abstractions over the different memory managers supported by the VM (including 32-bits and 64-bits abstractions) and the different assembly

back-ends. Most of the optimisations performed are platform-independent, though specific parts, such as inline cache relinking, need to be implemented differently in each back-end. Cogit currently supports four different back-ends in production: x86, x64, ARMv6 and MIPSEL.

Stack frame reification. The current VM evolved from the VM specified in the Smalltalk blue book [Goldberg 1983]. The original specification relied on a spaghetti stack: the execution stack was represented as a linked list of *contexts*, a context being a v-frame in the form of an object. Each context was represented as an object that could, as any other object, be read or written by the program.

Over the years, Deutsch and Schiffman [Deutsch 1984] changed the VM internal representation of the stack to improve performance. The new stack representation consists of a linked list of stack pages, where each stack page have stack frames next to each other. Most calls and returns, inside a stack page, can use efficient call and return instructions. Only uncommon calls and returns, across stack pages, need to use slower code. With the current production settings, each stack page has enough size to hold around 50 stack frames and different heuristics are used to make calls and returns across stack pages as uncommon as possible. Figure 3.2 shows the representation of the stack. In the figure, stack pages hold around 5 stack frames to make it easier to read, but in practice stack pages holding less than 40 frames induce considerable overhead.

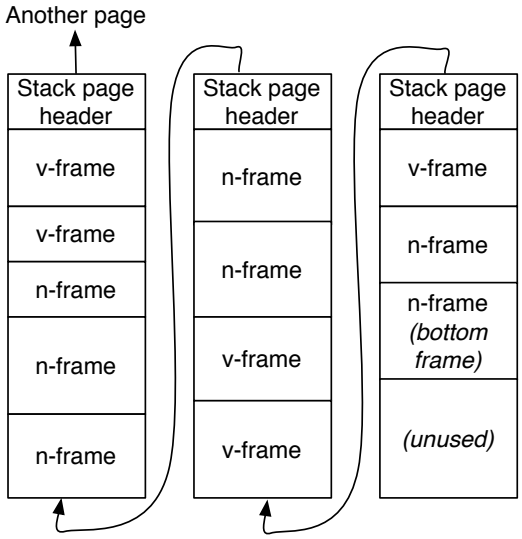


Figure 3.2: Stack representation

The VM however still provides the ability for the Smalltalk developer to read

and write the reified stack as if it was a linked list of contexts according to the original specification. To do so, the stack is reified as a linked list of contexts on demand.

The reification of the stack is used in three main places: the debugger, exceptions and continuations. For the latter two, they are implemented in Smalltalk on top of the stack reification, without any special VM support. From Smalltalk, any program can use this feature to instrospect and edit the stack.

Contexts abstract away from low-level details. A context is exactly the same if the VM is started with the interpreter only or with the hybrid interpreter plus baseline JIT runtime. Conceptually, for the Smalltalk developer, the code is interpreted and the contexts always look identical. The VM is responsible to intercept context accesses to read and write concrete v-frames and n-frames.

In this thesis, we suppose that a context is the same thing as a v-frame as the mapping between both has no impact in the design and brings no interesting additional concepts or side-effects. In practice, there are three differences:

1. A context is an object while a v-frame uses a low-level representation compliant with the stack.

2. V-frames have to care about some low-level details, such as calls and returns from v-frames to n-frames and n-frames to v-frames, or the access to the v-function arguments by reading values in the caller frame.

3. Contexts have a reference to the caller, as conceptually there is a linked list of contexts, while v-frames are below the caller on stack.

However, both v-frames and contexts use the virtual instruction pointer[1] and never the native instruction pointer and both refer to the v-function and never to the n-function. Both representations abstract away from machine-specific state as all the values used by the execution are always on stack and never in registers.

To read and write contexts, the VM intercepts all the accesses to the context objects. To do so, contexts can be in two forms. They can be "married" to a v-frame or n-frame, in which case they act as proxies to the frame. The VM then maps reads and writes to read and write the correct field in the frame representation. Alternatively, they can be "single" (for example when instantiated from Smalltalk), which means there is no stack frame on stack representing the context. In this case, the VM can modify the context as a normal object. Upon activation, the VM lazily recreates a v-frame for a single context to execute it (the single context is remarried to a new v-frame). Returns to single contexts are necessarily across stack page boundaries, hence the overhead to test if the caller is a context on heap or a

[1] A virtual instruction pointer is a pointer to an instruction in a v-function, by opposition to a native instruction pointer which points to an instruction in a n-function.

stack frame on stack is required only in the uncommon case of return across stack pages.

Aggressive stack manipulation (instruction pointer modification, caller modification) may lead the VM to crash. The program performing such operations needs to guarantee it won't happen, this is not the VM responsibility. In addition, these operations require a married context to "divorce" the frame, killing the frame in the process. Upon divorce, the stack page is split in two different stack pages (one part is copied to another memory location). One stack page returns to the single divorced context while the context returns to the other stack page, as shown on Figure 3.3. In normal execution the stack is composed exclusively of stack pages, but after stack manipulation from Smalltalk, the stack can be a linked list of stack pages and contexts.

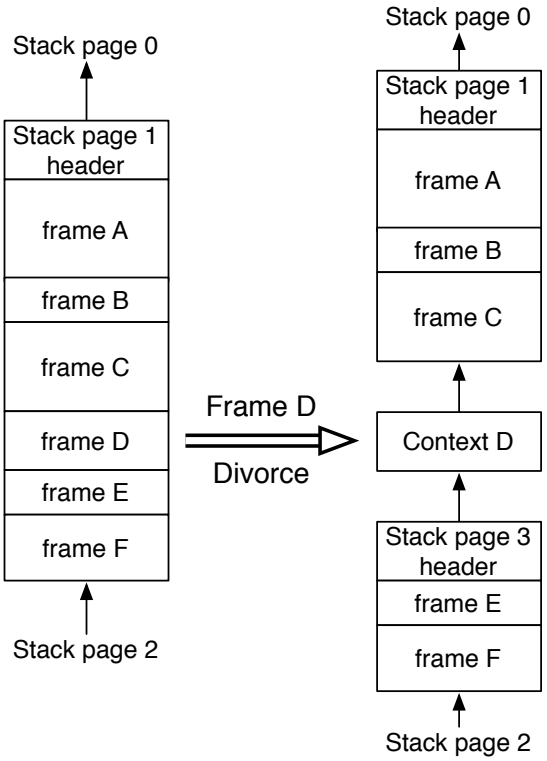


Figure 3.3: Divorce of stack frame D to context D

Marriages and divorces between stack frames and contexts are not specific to aggressive stack manipulations. They are also used for other features such as snapshots and stack page overflow. In the latter case, as there is a limited number of stack pages (currently 50 in production environments), when a stack page is required and none is available, the VM needs to free a page. To do so, the VM

marries then divorces all frames on the least recently used stack page to persist the page in the form of single contexts and re-uses the stack page for the execution of new code.

3.2 Language-VM interface

Pharo interfaces with the VM in two different ways:

1. Pharo can instantiate v-functions and install them for execution.

2. A small list of objects is registered in the VM for direct access.

Virtual function representation. As everything is an object in Pharo, virtual functions are objects. A v-function is composed of a function's header, a list of literals, a list of bytecodes and a reference to its source code as shown on Figure 3.4. The function's header contains information required for the function's execution such as the number of temporary variables or the size of the frame required. The last two literals, in the case of a method, are the selector and the method class, *i.e.,* the class where the method is installed.

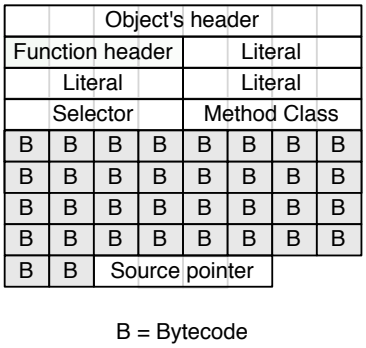


Figure 3.4: Virtual function representation

The bytecode set is stack-based. Most operations are pushing and popping values of the stack. All the operations are untyped and work with any object. One of the main instructions is the virtual call instruction, popping the receiver and arguments from the stack and pushing back the result. The bytecode set also includes conditional and unconditional branches to encode conditions and loops, as well as specific bytecodes to create efficiently closures.

Virtual function installation. Classes and method dictionaries are normal objects in Pharo. Hence, the installation of a method uses the normal dictionary API, inserting the selector as the key and the method as a value. Method dictionaries, upon modification, request the VM to flush look-up caches for the installed selector. As Pharo is dynamically-typed and through uncommon behavior (stack frame modification, exotic primitives) any method can be called by any object, flushing all the methods matching the selector is easier and safer to implement. Closure's functions are installed inside the method that instantiate them as a literal.

Primitive methods. Virtual methods can be annotated with a primitive number at creation time. A primitive method can be executed through a virtual call, like any other method, but upon activation a low-level function (either a Slang function or the native code generated from an assembly code template, depending on the current state of the runtime) is executed. The low-level code can fail if the receiver and arguments of the primitive method do not meet specific constraints.

Although primitive methods can be used for performance, most of them provides essential features that could not be implemented otherwise. For example, the addition between two integers is implemented as a primitive, forwarding the operation to the processor's implementation of the addition.

Smalltalk features a set of unconventional primitives, non present in most other programming languages. A notable example is become:, a primitive which swaps the references of two objects. If a become: b, then all references to a now refer to b, and all the references to b now refer to a. This primitive is implemented efficiently based on specific strategies [Miranda 2015].

Registered objects. An array of registered objects[2] is accessible both to the Pharo VM and from the language. This array contains multiple objects that need to be accessed directly by the VM, for example the objects nil, true and false. Any new object can be registered in the array and the array can grow on demand.

Among registered objects are specific selectors, such as the #doesNotUnderstand: selector. When a look-up performed by the VM does not find any method to activate (the selector is not implemented for the given receiver), the VM instead performs a virtual call, using the same receiver, the registered #doesNotUnderstand: selector and reifies the virtual call as an object (which class is also registered) containing the original selector, the arguments and the look-up class in case of a super send.

The registered objects allows some flexibility in the VM: each entry can be modified from the language, changing the VM behavior, without recompiling the VM.

[2] Smalltalk developers use the term special object array for this array

3.3 Language relevant features

This section details two main aspects of the programming language: native thread management and snapshots.

Native thread management. Pharo features a global interpreter lock, similarly to Python. Only calls to external libraries through the foreign function interface and specific virtual machine extensions have access to the other native threads. Smalltalk execution, including bytecode interpretation, machine code execution, just-in-time compilation and garbage collection are not done concurrently. Being single-threaded has a impact on design decisions because several other VMs implement the optimising JIT in concurrent native threads to the application [Arnold 2000].

Snapshots. In the context of Smalltalk, a snapshot[3] is a sequence of bytes that represents a serialized form of all the objects present at a precise moment in the runtime. As everything is an object in Smalltalk, including green threads, classes and v-functions, the virtual machine starts-up by loading all the objects from a snapshot and resumes the execution based on the green thread that was active at snapshot time. In fact, this is the normal way of launching a Smalltalk runtime.

One interesting problem in snapshots is how to save the execution stack, *i.e.,* the green threads. To perform a snapshot, each stack frame is reified into a context and only objects are saved in the snapshot. When the snapshot is restarted, the VM recreates a stack frame for each context lazily.

In any case, snapshots are platform-independent so they cannot save n-frames. In the Pharo VM for example, a snapshot can be taken on a laptop using a x86 processor and restarted on a raspberry pi using an ARMv6 processor.

Conclusion

The chapter described the aspects and features of Pharo relevant for the thesis. The following chapter describes the overall architecture discussed in the thesis.

[3] Smalltalk developers use the term *image* instead of snapshot.

CHAPTER 4

Sista Architecture

Contents

The overall thesis focuses on the design and implementation of an optimising JIT for Pharo, written in Pharo itself, running in the same runtime as the optimised application on top of the existing runtime environment. This chapter explains the overall designed and implemented architecture, called Sista. The first section gives an overview of the architecture. Section 4.2 details how functions are optimised. Section 4.3 focuses on the deoptimisation of optimised stack frames when an assumption taken at optimisation time is not valid at runtime. Section 4.4 compares Sista against related work.

4.1 Overview

This section starts by describing briefly the existing runtime and evolutions required to introduce an optimising JIT. Section 4.1.2 explains the overall design. Lastly, Section 4.1.3 briefly describes how functions are optimised and deoptimised.

4.1.1 Missing components

The existing Pharo runtime relies on a v-function interpreter and a baseline JIT named Cogit to execute functions. Cogit is able to compile a single v-function to a single n-function with a limited number of optimisations such as inline caches [Deutsch 1984, Hölzle 1991]. Cogit does not perform optimisations requiring speculations based on runtime information. To load new code in Pharo, it is possible to install new v-functions at runtime. In this case, the program requests the VM to flush caches matching the new v-functions installed.

Compared to the function-based architecture described in the previous chapter, the Pharo runtime is missing entirely an optimising JIT performing speculative optimisations based on runtime information. We note two main missing features:

- **Hot spot detection and n-function introspection:** The first missing feature is that the baseline JIT is not able to detect Hotspots to trigger the optimising JIT nor to introspect the n-function it generates to provide runtime information.
- **Optimising JIT:** The second missing feature is the optimising JIT itelf, which should be able to generate an optimised n-function from v-functions and the corresponding runtime information. The optimising JIT needs to include specific components in addition to the optimisation pipeline. A deoptimiser is needed to resume execution with unoptimised code when a speculation made at optimisation time is incorrect at runtime. A dependency manager is required to discard dependant optimised functions when new code is loaded.

4.1.2 Split design: Two parts for the optimising JIT

The architecture requires both baseline JIT extensions and the addition of the optimising JIT.

Baseline JIT extensions. The existing baseline JIT, Cogit, had to be extended to detect hot spots and to provide runtime information to direct the optimising JIT decisions.

To detect hot spots, Cogit was extended to be able to generate profiling counters in generated n-functions. When a profiling counter reaches a specific threshold, a specific routine is triggered and may activate an optimising JIT. More details on the profiling counters are present in Section 4.2.2.

Cogit was already able to introspect the n-function it generates for multiple purposes, such as debugging, inline cache relinking or literals garbage collection. The introspection logic was extended with a new primitive method, which answers for a given function both the values present in inline caches and the values of profiling counters.

Optimising JIT. The optimising JIT is designed in two different parts as shown in Figure 4.1. The high-level part is a unoptimised v-functions to optimised v-function compiler called *Scorch*. The second part is a v-function to a n-function compiler and an extended version of Cogit, the baseline JIT, is used. The compilation of unoptimised v-functions to an optimised n-function through Scorch followed by Cogit forms an optimising JIT.

Scorch is written in Pharo and runs in the same runtime as the optimised application in a metacircular style. Scorch deals with Smalltalk-specific optimisations. Hence, any work performed on Scorch is done in Pharo dealing with Smalltalk-specific optimisations. Such work can be performed with little knowledge on low-level or machine-specific details. The optimised v-functions generated by Scorch may use unsafe instructions in addition to the unoptimised v-function instructions. Unsafe instructions are faster to execute but require the compiler to guarantee specific invariants. For example, an unsafe array access is faster to execute than a normal array access as it does not perform bound checks, but Scorch needs to guarantee that the array access is always in-bounds.

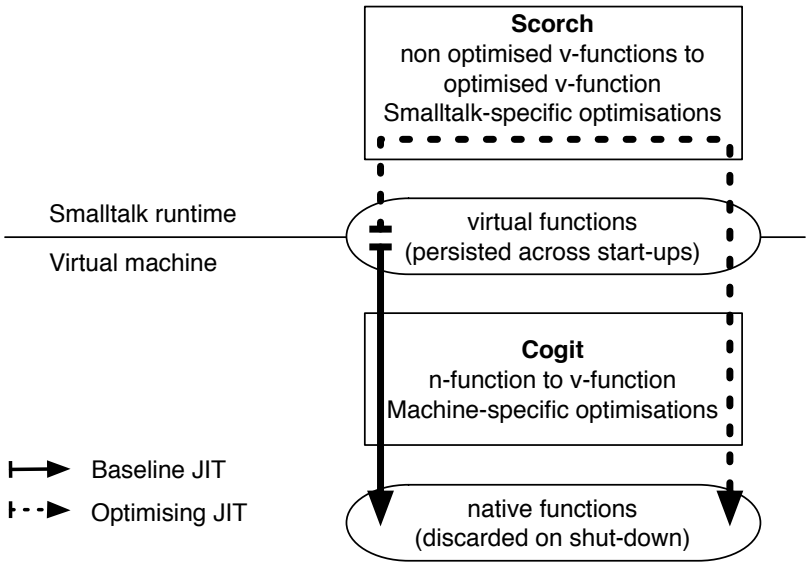


Figure 4.1: Scorch critical and background modes

For the low-level part, the existing baseline JIT Cogit is reused. Cogit may perform machine-specific optimisations. To be used as a back-end for the optimised v-functions, Cogit was extended to support new unsafe instructions, including a specific VM call-back to trigger deoptimisation when an assumption speculated at optimisation time is invalid at runtime.

4.1.3 Optimisation and deoptimisation

Cogit was extended to detect hot spots through profiling counters in unoptimised n-functions. When a hot spot is detected, Cogit immediately calls Scorch in Pharo. Scorch then looks for the best v-function to optimise based on the current stack, optimises it and installs the optimised version. To perform optimisations, Scorch

asks Cogit to introspect specific n-functions to extract type information and basic block usage from previous runs. Once installed, the VM can execute the optimised v-function at the next call to the function.

As the VM runtime uses both an interpreter and Cogit, the optimised v-function may conceptually be interpreted or compiled by Cogit and then executed as an optimised n-function. In practice, new heuristics were introduced for optimised v-functions to execute them as optimised n-functions from the first run. The details of the function optimisation logic is written in Section 4.2.

Due to speculative optimisations, optimised v-functions may contain guards to ensure optimisation-time assumptions are valid at runtime. When a guard fails, the execution stack needs to be deoptimised to resume execution with unoptimised code. When an optimised n-frame needs to be deoptimised, Cogit maps the optimised n-frame to a single optimised v-frame. Cogit then provides the optimised v-frame to Scorch, which maps the optimised v-frame to multiple unoptimised v-frames. Scorch may discard the optimised v-function if guards are failing too often in it. The execution can then resume using unoptimised v-functions. The deoptimisation logic briefly described here is explained in detail in Section 4.3.

4.2 Function optimisation

Cogit was extended to detect hot spots based on profiling counters. When a hot spot is detected, Cogit triggers a call-back to request Scorch to optimise a v-function-based on the current stack. As Pharo is currently single-threaded, the application green thread has to be interrupted to optimise a function. The overall design is then the following: after interrupting the application, Scorch finds a v-function to optimise based on the current stack, optimises it, installs the optimised version, and resumes the application. The installed optimised v-function will be executed at the next call of the function.

4.2.1 Optimiser critical and background modes

Scorch optimiser may however require a significant amount of time to optimise a v-function. Optimising a v-function can take a long time in slow machines or when a pathological function [1] is optimised. This can lead to the interruption of the application for an amount time long enough to be noticed by the user. To experiment with Sista, we worked with the development environment of Pharo (which is written in Pharo). In the case of a user-interface application, it is *very*

[1] Many compiler algorithms have a good average complexity based on heurisitics but poor worst complexity. A pathological function is a function not matching any heuristic leading to long optimisation time.

annoying to see the application interrupted during half a second or more when multiple v-functions long to optimise are optimised in a row. The user interface feels slow, lagging and unresponsive even though the overall code takes less time to execute.

To avoid the problem, we limited the optimisation time to a fixed small time period, configured from the language. For user interface application, we limit it to 40 ms. The limitation is enforced by a high-priority green thread, set to stop the optimiser after the given time period. As the current user interface is refreshing at 50Hz, the optimiser, in the worst case, forces the system not to refresh the screen twice. In practice, most v-functions are optimised in less than 40ms. However, some v-functions are still too long to optimise, so an alternative solution is required to optimise them.

Upon profiling the development tools, as one would expect, we noticed that the application spends a significant amount of time idle[2]. We show for example in Figure 4.2 that the application is successfully executing code, then idle, then executing code again, etc. In this case, each time an event happen (key stroke, mouse click, etc.), some code is executed, but when no event happens, for example when the developer is reading code, the application is in idle.

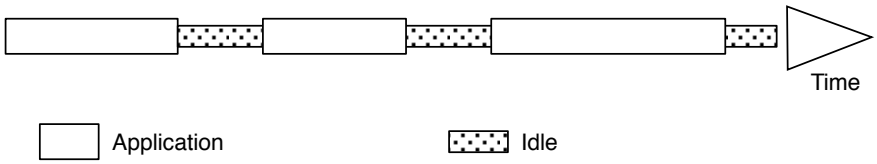


Figure 4.2: User interface application idle times

Based on the profiling result, we introduced a background green thread responsible for optimising the functions too long to optimise in the limited time period allowed. This way, when the application would normally become idle, it starts by optimising such functions and becomes idle when no functions to optimise are remaining. As the background green thread is running at low priority, if the application restarts while an optimisation is being performed, the application green thread preempts the optimisation green thread and no pauses are seen by the user.

The optimiser can therefore be run in two modes. When a hot spot is detected, the optimiser is started in *critical mode*. It has a limited time period to optimise a function-based on the current stack. If the optimisation takes too long, the function to optimise is added to the background compilation queue. When the application

[2]An application in idle means it has nothing to do, it is typically waiting for an event to do anything.

becomes idle, if the background compilation queue is not empty, the optimiser is started in *background mode*. In background mode, the optimiser is run in a low-priority green thread and is preempted by any application green thread. When the optimiser has optimised all the functions in the compilation queue, it stops and the application becomes idle. Scorch optimiser critical and background modes are represented on Figure 4.3.

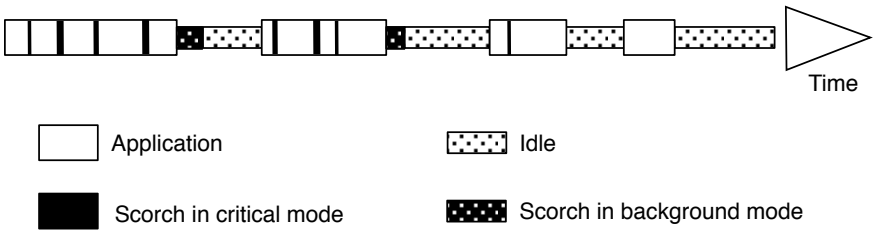


Figure 4.3: Scorch critical and background modes

Conclusion. Scorch optimiser can be run in two modes. In critical mode, it interrupts the application green thread and has a limited time period to optimise a function. If the time period is not enough, the function's optimisation is postponed to the background mode. In background mode, Scorch optimises code only when the application is idle but has no time limit. This design allows all the application's code to be optimised in a single-threaded environment without the system loosing too much responsiveness.

4.2.2 Hot spot management

Cogit was extended to be able to generate n-functions with profiling counters. Profiling counters allow one to detect hot spots and provide information about basic block usage at the cost of a small overhead, detailed in the validation chapter of the thesis. Because of the overhead, Cogit was extended to support conditional compilation. Based on a specific bit in the v-function's header, Cogit compiles the v-function with or without profiling counters. Typically, unoptimised v-functions, produced by the source code to v-function bytecode compiler [Béra 2013], are by default compiled to n-functions with profiling counters, while optimised v-functions are compiled without profiling counters. Profiling counters are generated so that the counter is increased by one when the execution flow reaches it and a specific hot spot detection routine is called when the counter reaches a threshold.

Based on [Arnold 2002], we added counters by extending the way the baseline JIT generates conditional jumps. Counters are added just before and just after a

branch. In several other VMs, the counters are added at the beginning of each function. The technique we used allows us to reduce the counter overhead as branches are 6 times less frequent that virtual calls in Smalltalk. In addition, the counters provide information about basic block usage. Every finite loop requires a branch to stop the loop iteration and most recursive code requires a branch to stop the recursion, so the main cases where hot spots are present are detected.

When a hot spot is detected, a specific Slang routine is called. The routine makes sure the n-frame where the hot spot is detected is reified so it can be introspected from Pharo. Then, the routine performs a virtual call with a selector registered from Pharo, the reified stack frame as the receiver and the boolean the conditional jump was branching on as a parameter. The method activated by the call-back, in Pharo, calls Scorch optimiser.

During optimisation, the bottom frames of the execution stack are used by Scorch optimiser. The frame above is the call-back frame, followed by the application frame holding the n-function with the hot spot, as shown on Figure 4.4.

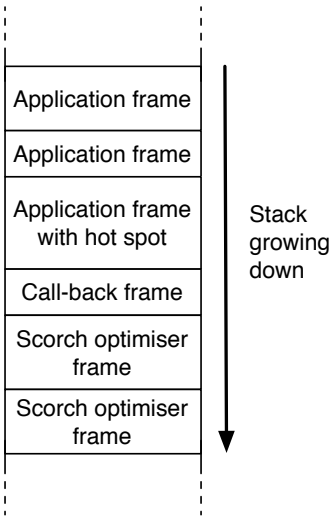


Figure 4.4: Stack state during critical mode optimisation

4.2.3 Scorch optimiser

Scorch optimiser is activated by the call-back and has access to the reified stack. Scorch firstly analyses the stack and finds the best function to optimise. Then, it generates, either directly in critical mode or indirectly through background mode an optimised v-function and installs it for further uses.

Stack search. When a hot spot is detected, Scorch is activated and has access to the reified stack. A naive approach would be to always optimise the function where the hot spot is detected and not to search the stack at all. Unfortunately, this heuristic would be terrible for a Smalltalk program. An important part of the execution time is due to the extensive use of closures. More specifically, most loops in the executed code, assuming the code respects standard Smalltalk coding conventions, are using closures. To efficiently remove the closure overhead, the closure needs to be inlined up to its enclosing environment to remove both the closure creation and the closure activation. If the function where the hot spot is detected is either activating closures or a closure activation itself, then optimising it won't gain that much performance because the closure creation and activation execution time will remain.

Another approach, a bit less naive, would be to optimise the function where the hot spot is detected if it is a method, and the enclosing environment's function if it is a closure, in an attempt to remove closure overhead. Yet, this heuristic still does not solve the most common case of the problem. To illustrate the problem, let's look at a simple example with a loop over an array.

In the code sample in Figure 4.6, exampleArrayLoop is a method installed in the class ApplicationClass. Its method body consists of a loop over an array, the array being an instance variable. To loop over an array, Smalltalk provides high-level iterator methods such as do:. In this case, do: is very similar to foreach in other languages and allows one to iterate over the array while providing at each iteration of the loop the array's element in the variable element. The do: method, installed in Array, takes a single argument, a closure, which is evaluated using value: at each iteration of the loop. The parameter of the closure activation is self at: i, which represents the access to the element i of the array. During the closure evaluation, the bottom three frames are the closure activation, the frame for Array » do: and the frame for ApplicationClass » exampleArrayLoop as shown on Figure 4.5.

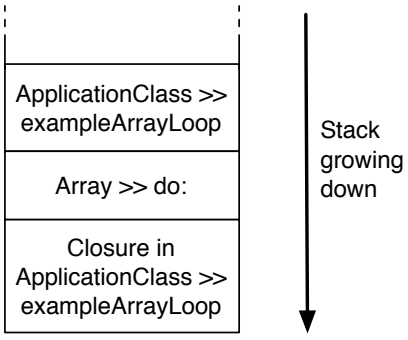


Figure 4.5: Example stack during closure execution

Method	ApplicationClass » exampleArrayLoop	Array » do: aClosure
Source Code	array do: [:elem | FileStream stdout nextPutAll: elem printString].	1 to: self size do: [:index | aClosure value: (self at: index)].
Bytecode	17 <00> pushRcvr: 0 18 <F9 00 00> createClosure 25 <10> pushLit: FileStream 26 <81> send: stdout 27 <40> pushTemp: 0 28 <82> send: printString 29 <93> send: nextPutAll: 30 <5E> blockReturn 21 <7B> send: do: 22 <D8> pop 23 <58> returnSelf	17 <4C> self 18 <72> send: size 19 <D1> popIntoTemp: 1 20 <51> pushConstant: 1 21 <D2> popIntoTemp: 2 22 <42> pushTemp: 2 23 <41> pushTemp: 1 24 <64> send: <= 25 <EF 0E> jumpFalse: 41 27 <40> pushTemp: 0 28 <4C> self 29 <42> pushTemp: 2 30 <70> send: at: 31 <7A> send: value: 32 <D8> pop 33 <42> pushTemp: 2 34 <51> pushConstant: 1 35 <60> send: + 36 <D2> popIntoTemp: 2 37 <E1 FF ED ED> jumpTo: 22 41 <58> returnSelf

Figure 4.6: Example code

The method Array » do: is using a special selector, to:do:, which is compiled by the bytecode compiler to a loop, in a similar way to for constructs in other programming languages. In fact, the Array » do: method body is a loop from 1 to the size of the array, evaluating the closure at each iteration for each element in the array. At each iteration, the current value of index is tested against the size of the array, and when that value is reached the loop is exited.

As discussed in the previous section, profiling counters detect frequent portion of code on branches. Each finite loop has a branch to either keep iterating over the loop or exit the loop. In the example, it means that the method Array » do: has a profiling counter on the branch testing the value of the index against the size of the array. The rest of the code, in the two methods and in the closure, have no other profiling counters.

The hot spot is going to be detected on the profiling counter, hence in the method Array»do:. If Scorch optimised the Array»do: method, it cannot know what closure will be executed as the closure is an argument, while an important part of the execution time is spent creating and evaluating the closure. However, if Array»do: gets inlined into ApplicationClass»exampleArrayLoop, the closure evaluated would be known to be the closure [:element | FileStream stdout nextPutAll: element printString]. Hence, to gain maximum performance, the optimiser should decide to optimise ApplicationClass»exampleArrayLoop and inline both the Array»do: method and the closure evaluation (value:).

In this case, the hot spot is detected in Array»do:. The hot spot is therefore detected in a method, not a closure. Naive heuritics would have chosen to optimise Array»do:, while it is better to select the caller stack frame's function.

Overall, because of the extensive use of closures, the optimiser almost never chooses to optimise the function where the hot spot is detected. It usually walks up a few frames to find the best function to optimise based on multiple heuristics.

Optimised v-function generation. Once Scorch has selected the v-function to optimise, it generates an optimised v-function. It attempts to do it immediately, within a limited amount of time. If it fails to do it, it postpones the optimisation to background mode. The function's optimisation and installation is using the same code in both cases, so we will discuss only the critical mode optimisation in the following paragraphs.

Scorch optimiser is implemented with traditional compiler optimisation techniques. It decompiles the v-function to optimise into a single static assignment (SSA) intermediate representation, represented in the form of a control flow graph. Specific instructions that may require deoptimisation of the optimised frame have deoptimisation metadata attached which is updated during the optimisation passes to still refer to the correct values. During decompilation, Scorch asks Cogit to introspect the n-function corresponding to the decompiled v-function. If such a n-function exists, Cogit provides type information for each virtual call based on the data present in each inline cache and provide basic block usage based on the profiling counter values. The intermediate representation is annotated with this runtime information.

Scorch optimiser performs Smalltalk specific optimisations, very similar to the object-oriented specific optimisations present in other optimising JITs (speculative inlining, array bounds check elimination, etc.). Guided by the information provided by Cogit, Scorch speculates on receiver types for each virtual call to determine what v-function to inline. Each inlined v-function is decompiled to the same intermediate representation, annotated with runtime information the same way and merged into the same control flow graph. Each inlined function requires the insertion of a deoptimisation guard, to stop using the optimised code at runtime if the

receiver type assumptions are not valid anymore.

Once the inlining phase is finished, Scorch optimiser performs standard optimisations such as array-bounds check elimination with the ABCD algorithm [Bodík 2000] or global value numbering. Scorch optimiser also postpones the allocation of objects not escaping the optimised v-function from runtime to deoptimisation time (or completely removes the allocation if the object is never required for deoptimisation) [3].

The back-end is the only non-conventional part of the optimiser. Scorch generates an optimised v-function and not an optimised n-function. Most intermediate representation instructions are translated to a single bytecode instruction from our extended bytecode set. However, as the bytecode set is stack-based, Scorch back-end needs to map each used intermediate representation instruction value either to a value on stack or a temporary variable. The deoptimisation metadata needs to be updated accordingly.

Lastly, the back-end generates an optimised v-function. For each point where deoptimisation could be requested (typically, failing guards, but also each virtual call for debugger support), the optimised v-function has metadata attached to reconstruct the stack with unoptimised v-functions.

Installation. If the optimisation has been done, the optimised v-function is installed. It is installed in the method dictionary of a class if this is a method, or in a method if it's a closure. If an optimised method is installed, the installation explicitely requests the VM to flush the caches dependent of this installation so it can be used at next call (the global look-up cache for the interpreter and the inline caches).

In addition, the dependencies are installed in the dependency manager so that if new code is loaded, code that may be dependent is discarded. Indeed, if a new version of a v-function is installed while the previous version was inlined in optimised v-functions, all optimised v-functions having inlined the previous version of the function need to be discarded. Frames on stack using discarded functions are lazily deoptimised when the execution flow returns to them.

Once installed, conceptually, the VM runs the optimised v-function as a normal v-function. The first few runs can be interpreted and the subsequent runs use the n-function produced by Cogit. The only difference is that optimised v-functions have access to additional operations, but those operations are supported both by the interpreter and by Cogit. Our first version worked exactly this way.

We then added a cheap heuristic to encourage the execution of optimised v-

[3] Objects used only inside optimised functions are not allocated unless deoptimisation is triggered. The state of such objects is moved from a heap location to the stack to allow one to optimise read-write using the single-static assignment property.

functions as n-functions. Optimised v-functions have a bit set in their header to tell Cogit not to compile profiling counters when generating their corresponding n-function. If this bit is set, the interpreter asks Cogit to compile it at the first run and immediately uses the n-function instead of doing so after a couple of interpretations.

In any case, the VM still needs to support the interpretation of optimised v-functions. Indeed, in very rare cases, Cogit cannot temporarily generate a n-function for the given v-function. For example, as the native code zone for n-functions has a fixed size of a few megabytes, it can happen that Cogit tries to compile a v-function while relinking a polymorphic inline cache [Hölzle 1991] of another n-function. If there is not enough room in the machine code zone, a compaction of the machine code zone has to happen while relinking. It is not easy to compact the machine code zone at this point as it can happen that the polymorphic inline cache or the n-function holding it is discarded. To keep things simple, in this situation, we postpone the machine code zone compaction to the next interrupt point and interpret the v-function once. The interpretation of optimised v-functions, even if uncommon, is required for the VM to execute code correctly.

4.3 Function deoptimisation

The deoptimisation of the execution stack is similar to other VMs [Fink 2003, Hölzle 1992]. An optimised frame is on stack and is in most cases a n-frame [4]. The optimised frame cannot be used any more because an optimisation time assumption is invalid at runtime (a deoptimisation guard has failed) or the optimised function was discarded (by the debugger or because of new code loading). Deoptimising the stack requires the mapping of the optimised frame to multiple unoptimised v-frames.

In our architecture, deoptimisation is done in two steps as shown on Figure 4.7. Firstly, Cogit deoptimises the optimised n-frame to a single optimised v-frame. This step is not performed in the uncommon case where deoptimisation happens already from an optimised v-frame. For the rest of the section, we assume that the optimised frame is a n-frame, the other case being uncommon and being implemented simply by ignoring this first step. Secondly, Scorch deoptimises the optimised v-frames to multiple unoptimised v-frames.

When discussing deoptimisation, we deal only with stack recovery (deoptimisation of the optimised frame to the unoptimised v-frames). The unoptimised v-function is always present and never discarded, so the deoptimiser does not need to recreate it when restoring v-frames. There is no such thing in our de-

[4] In very uncommon cases, the VM may decide to interpret an optimised v-function, leading to the presence of an optimised v-frame.

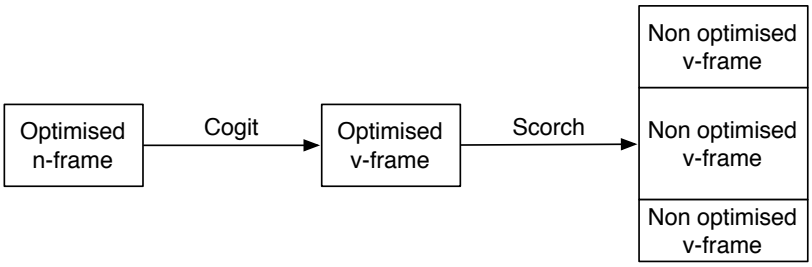


Figure 4.7: Stack frame deoptimisation in two steps

sign and implementation as reconstructing unoptimised v-function from optimised v-function-based on deoptimisation metadata. As far as we know, modern VMs such as V8 [Google 2008] always keep an unoptimised version of each function, so we believe the memory footprint impact of keeping them is acceptable.

4.3.1 Deoptimisation of a frame

Deoptimisation can happen in two main cases. On the one hand, Scorch optimiser inserts guards [Hölzle 1992] to ensure assumptions speculated at optimisation time are valid at runtime, such as the speculation on types. Such guards can fail, requiring deoptimisation of the execution stack to keep executing the application correctly. On the other hand, Smalltalk code can request deoptimisation of the code when manipulating the stack (typically the debugger's code does it).

Guard failure. When a guard fails, the VM reifies the optimised frame so it can be introspected from Smalltalk. It then calls Scorch deoptimiser, switching from the VM executable code to Smalltalk, to restore the unoptimised stack

During deoptimisation, the bottom frames are used by the deoptimiser. Just above is a frame for the activation of the virtual call performed by the routine followed by the optimised frame requesting deoptimisation, as shown on Figure 4.8.

Smalltalk code deoptimisation. The Smalltalk code can request deoptimisation of specific frames to perform specific operations such as debugging. In this case, the situation is different because:

1. The frame to deoptimise is in the middle of the application frames. Instead of having the call-back and deoptimiser frames below the frame to deoptimise on stack, other application frames are present.

2. The instruction pointer is not on a guard instruction.

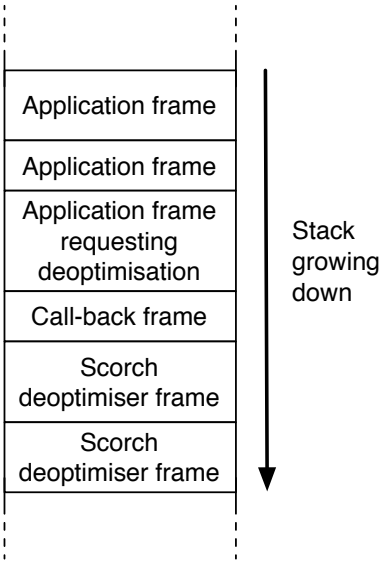


Figure 4.8: Stack state during guard deoptimisation

Deoptimisation metadata. To restore unoptimised frames, each instruction where deoptimisation can happen in optimised v-functions is annotated with deoptimisation metadata. The deoptimisation metadata is composed of the description of objects to recreate at deoptimisation time to resume execution with unoptimised functions at the corresponding annotated instruction.

Some objects to recreate are unoptimised v-frames (reified as objects as detailed in Section 3.1). These unoptimised v-frames, once recreated, will replace the optimised v-frame requesting deoptimisation on stack. Other objects to recreate are objects which allocation has been postponed from runtime to deoptimisation time, because the allocation was not required in the optimised function and ignoring the allocation speeds-up code execution.

The description of objects to recreate specifies what value the deoptimiser has to set in each field of each object recreated. The value can be a constant, a value to fetch from the optimised frame or another object to recreate.

4.3.2 Scorch deoptimiser

Cogit has reified the optimised n-frame to an optimised v-frame, hence, Scorch deoptimiser accesses the optimised v-frame. The frame is necessarily activated on a virtual instruction which is annotated with deoptimisation metadata. The metadata consists of a list of objects to materialize, which allocation has been postponed from runtime to deoptimisation time. As v-frames and closures are reified as ob-

jects in Smalltalk, part of those objects to rematerialize are unoptimised v-frames and closures. For each field of each object, the metadata specifies if the value is a specific constant, a value to fetch in the optimised v-frame or a reference to one of the other rematerialized object.

Stack recovery. Once all the unoptimised v-frames are reconstructed, the execution stack needs to be edited to use them. This is done using the stack manipulation APIs. Basically, the VM splits the current stack page in two, copying one part on another stack page. Deoptimised frames are present in the middle in their single context form, in a similar way to frame divorces described in the previous chapter and as shown in Figure 4.9.

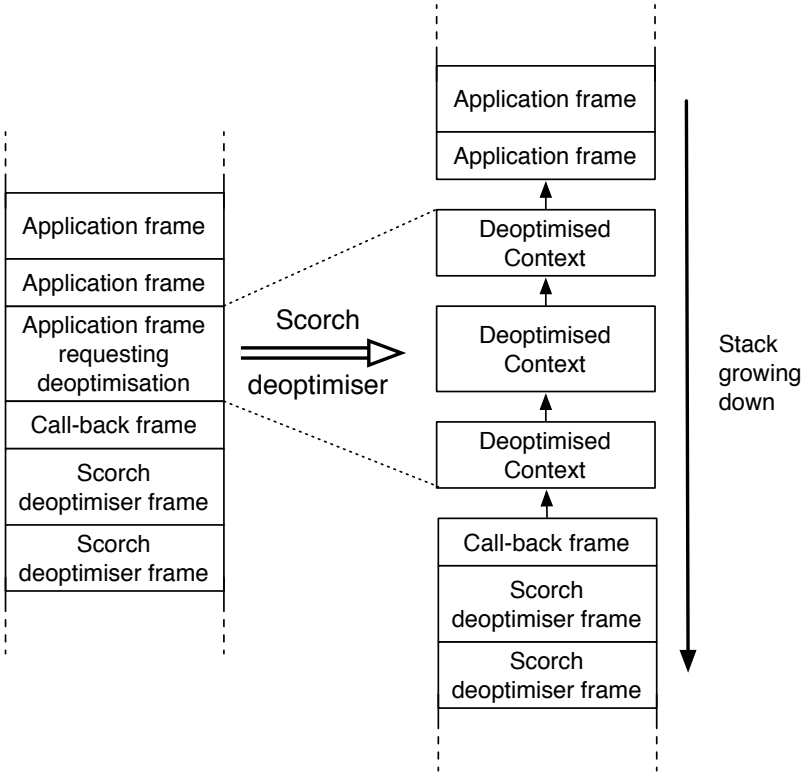


Figure 4.9: Stack recovery

Discard functions. When new unoptimised functions are installed (for example when a library is loaded), optimisation assumptions may be invalidated because some look-up results cannot be guaranteed anymore. In this case, we cannot really iterate over the whole stack and deoptimise all the frames holding invalidated code

as it would take a long time (especially multiple stack manipulations may take a while). Instead, Scorch sets all the virtual instructions of the discarded optimised v-function to guard failures (virtual instruction calling the deoptimiser), so optimised v-frames trigger deoptimisation when returned to. In addition, Scorch requests Cogit to patch all return native instruction pointers of all optimised n-frames representing discarded function activations to a pointer to a specific routine which will trigger deoptimisation upon return.

Execution restart. If the deoptimisation happened due to a guard failure (or due to a return to a discarded function), the application needs to resume execution once deoptimisation is performed. In this case, the application frame is just above the call-back frame so returning resumes execution.

4.4 Related work

This section compares our architecture against existing work. The first section compares the overall architecture versus the tracing and function-based architectures. Section 4.4.2 compares our optimised v-function representation against representations existing in other VMs. Section 4.4.3 discusses how different JIT tiers may or may not share portions of code.

The two research subproblems and their related work are detailed later in the thesis. Chapter 6 discusses how Scorch is able, under specific constraints, to optimise its own code. Chapter 7 details how the runtime state, including optimised v-functions, can be persisted across multiple start-ups.

4.4.1 Architecture

Compared to the classical three tier function-based architecture described in Chapter 2, the existing Pharo runtime featured only the first two tiers. Sista is an implementation of the third tier, the optimising JIT tier. The implementation is however quite different from other VMs. One of the main difference is the split between the high and low-level part of the optimising JIT as well as the optimised v-function representation used to communicate between both parts. As far as we know, no other VM splits its optimising JIT this way. The other main difference is Cogit, which in one code base, can be used both as a baseline JIT and as a back-end for the optimising JIT.

Sista was not designed as a tracing JIT. The main reason is that improving the performance of loop bodies did not seem the right thing to do in the case of Pharo. As explained in the example of Section 4.2.3, if the Smalltalk code executed respects standard Smalltalk coding conventions, most loops just activate a closure.

The optimiser needs to optimise code outside of the loop body to be able to inline the closure activation and improve performance. Hence, it is not clear how to design an efficient tracing JIT for Pharo: the naive strategy of focusing the optimisations on loop bodies is not good enough in our case. The project RSqueak [Felgentreff 2016b] implements a Squeak VM using the RPython toolchain [Rigo 2006] and the problem described is present.

4.4.2 Optimised virtual function representation

One important difference in Sista compared to most optimising JITs lies with the optimised v-function representation. Most optimising JIT do not generate optimised v-function but generate only optimised n-functions. As far as we know, no existing baseline JIT is used to compile v-functions optimised using runtime information to optimised n-functions.

Threaded code. One of the first works in the late 80s to speed up object-oriented language VMs was in the direction of threaded code interpreters. Such virtual machine would feature a threaded code interpreter instead of a v-function interpreter. A threaded code interpreter is faster to execute code, but requires a small compilation time to translate the v-function to its threaded code representation. The threaded code representation is platform-independent and can be considered as an optimised v-function. The main advantage of the threaded code interpreter over a JIT is that it provides speed-up while being platform-independent. Indeed, a JIT requires a different back-end for each native code supported.

Work in this direction has mostly vanished for two reasons. Firstly, the execution of v-function through JITs is more efficient than the execution through a threaded interpreter. Secondly, the use of threaded jumps in v-function interpreters allowed to massively reduce the performance difference between threaded code and v-function interpreters.

Recently, one problem for VM implementors was to implement JITs for iOS. Apple's policy was forbidding, until very recently, to have a memory page both writable and executable. Such a page is required by a JIT to store the native code it generates so it was not possible to build a JIT running in iOS. One experiment in the Dart VM was to keep the three tier strategy of the function-based architecture, but the baseline and optimising JIT would generate an abstract assembly, similar to threaded code. The abstract assembly was then executed quickly by a low-level interpreter, mapping almost one-to-one abstract assembly instructions to native instructions. This solution was not as fast as regular JITs, but it allowed the VM to perform decently under Apple's policy constraints.

WebAssembly. Recent Javascript VMs have support for WebAssembly [Group 2015], an abstract assembly code representation for functions. In this case, the VM can take as input two different representations of v-functions. One form is the v-function representation for the language supported, in the case of Javascript the source code of v-functions. The other form, the WebAssembly form, allows the VM to execute v-functions optimised ahead-of-time.

It would be interesting for Scorch to target an optimised v-function representation such as WebAssembly instead of our extended bytecode set. We implemented Scorch to target the extended bytecode set as we could make the architecture work with a limited number of evolutions in Cogit. Being able to compile efficiently a representation such as WebAssembly would require us to implement larger extensions to Cogit. One of the goal of our architecture was to limit evolutions on low-level parts of the VM, so the direction we took looked more interesting. Recently, an abstract assembly started to be supported in the Pharo VM, called LowCode [Salgado 2016]. It would be interesting, as future work, to investigate if Scorch targeting LowCode is a valuable option.

Hotspot Graal runtime. The VM design the most similar to ours is certainly the Hotspot VM using the Graal compiler [Oracle 2013, Duboscq 2013] as an alternative optimising JIT. In both cases, the baseline JIT and the interpreter are compiled to machine code ahead-of-time (in our case, from Slang to native code, in Hotspot, from C++ to native code). The optimising JITs, Scorch and Graal, are however written in the language run by the VM and run in the same runtime as the optimised application.

The main difference still lies with the optimised function representation. Graal generates optimised n-functions in the form of a data structure including the native code and metadata. Graal provides these data structures to the Hotspot VM so it can install them [Grimmer 2013]. Scorch generates however optimised v-functions, which requires Cogit to compile them.

With the Graal strategy, it may be possible to produce optimised n-functions slightly faster to execute as the Graal back-end may perform low-level optimisations more aggressively than Cogit. The compilation time may also be slightly better as no optimised v-function representation needs to be created. However, our optimised v-function representation allows us to have a single back-end to maintain for the two JIT tiers in the form of Cogit and as discussed later in the thesis to persist optimised n-functions across multiple start-ups.

4.4.3 Sharing code between compiler tiers

Most optimising JITs have no code in common with the baseline JIT they extract runtime information from. In our context, Cogit is used both as the baseline JIT

and as a back-end for the optimising JIT. Most VM teams keep the back-ends of different JIT tiers independent as each back-end has different constraints. On the one hand, the optimising JIT back-end needs to generate high-performance code in a reasonnable amount of time. On the other hand, the baseline JIT back-end needs to generate code that can be instrospected later as quickly as possible.

We believe that with Cogit performing a limited number of low-level optimisations while providing n-function instrospection, we can reduce the maintenance cost of the low-level parts of our VM compared to other architecture with a reasonnable performance loss.

There are two main cases in the litterature where JIT tiers are sharing portions of code.

Webkit VM. The Javascript Webkit VM [Webkit 2015] is one of the only VMs where important parts of code are shared between multiple JIT tiers. The Webkit VM is different from other VMs as it features four tiers, including two optimising compiler tiers. In the webkit VM, the two optimising JIT tiers are sharing the high-level part of the compiler, but not the back-end. There is some code shared for n-function introspection, but each back-end of each JIT tier is mostly independent from the other tiers.

In our case, our two compiler tiers share the same back-end is used. In addition, the portion of code is shared between a baseline and an optimising JIT and not two optimising JIT tiers.

WebAssembly. Modern Javascript VMs support WebAssembly [Group 2015], an abstract assembly representation for functions. In the V8 Javascript engine [Google 2008], the back-end of the optimising compiler Turbofan is shared between the WebAssembly compiler and the optimising JIT. There are some similarities with our work as WebAssembly is a representation that could be used to represent optimised v-functions.

In this case, the back-end is shared between two optimising runtime compilers. There is no code shared between the optimising JIT and the baseline JIT.

Conclusion

The chapter provided an overview of our architecture and detailed the optimisation of functions and deoptimisation of stack frames. The following chapter discusses the evolutions of the Pharo runtime implemented in the context of the thesis.

CHAPTER 5

Runtime evolutions

Contents

To support the architecture described in the previous chapter, the Pharo runtime had to evolve. We distinguish two kind of evolutions. Some evolutions were required to support the architecture, the Sista runtime could not work without those features. Such evolutions are detailed in Section 5.1. Other evolutions were not mandatory, Sista could have worked without these features, but each of them were important to improve the overall performance. Those evolutions are described in Section 5.2. To implement Sista, I did not work alone. Section 5.3 details the distribution of work.

5.1 Required language evolutions

Five major evolutions were required to have Sista up and running:

1. Cogit was extended to detect hot spots through profiling counters.

2. The interpreter and Cogit were extended to be able to execute and compile the additional instructions of the extended bytecode set.

3. Two VM call-backs were added to trigger Scorch when a hot spot is detected or a guard fails.

4. A new primitive was introduced to provide runtime information in Smalltalk for v-functions having a corresponding n-function generated by Cogit.

5. Scorch, including the optimiser, the deoptimiser and the dependency manager were introduced.

5.1.1 Profiling counters

To detect hot spots, Cogit was extended to be able to generate profiling counters when generating a n-function. When the execution flow reaches a counter, it increases its value by one. If the counter reaches a threshold, the VM triggers a special call-back to activate Scorch. Profiling counters induce overhead, which can be significant enough to be seen in some benchmarks (the overhead is detailed on a set of benchmarks in the validation chapter, Chapter 8). To avoid the overhead in optimised code, Cogit was extended to support conditional compilation. Based on a bit in a v-function's header, Cogit generates a n-function with or without profiling counters.

Based on [Arnold 2002], we added profiling counters on conditional branches, with one counter just before and one counter just after the conditional branch. This strategy allows the VM to provide basic block usage information in addition to the detection of hot spots. Every finite loop requires a branch to stop the loop iteration and most recursive code requires a branch to stop the recursion, so the main cases for which we wanted to detect hot spots for are covered. Each time the execution flow reaches a conditional branch in a n-function, it increases the profiling counter by one, compares the counter value to a threshold and jumps to the hot spot detection routine if the threshold is reached. If the threshold is not reached, the conditional branch is performed. If the branch is not taken, a second counter is incremented by one to provide later basic block usage information.

The main issue we had to deal with when implementing profiling counters is the location of the counters and the access to the counters. Indeed, in our first naive implementation, the counter values were directly inlined in the native code. That was a terrible idea as every write near executable code flushes part of the processor instruction cache, leading to horrible performance. In the end, we changed the logic to allocate a pinned unsigned 32-bits integer array[1] for each n-function requiring counters. The pinned array is on heap, far from executable code, and contains all the counter values. As the array is pinned, the native code can access the array and each of its fields (each counter) through a constant address. This is very nice as the native code can be efficient by using constant addresses and the n-function does not require any metadata[2].

Figure 5.1 shows a n-function with two profiling counters. The n-function is present in the n-function zone, which is readable, writable and executable. The n-function zone is exclusively modified by Cogit. The pinned array is allocated on heap, where all objects are present, which is a readable and writable (but not

[1] A pinned object is an object that can never be moved in memory. For example, the garbage collector cannot move it.

[2] References to non pinned objects from n-function normally require metadata to update the reference when the object is moved in memory, typically by the garbage collector.

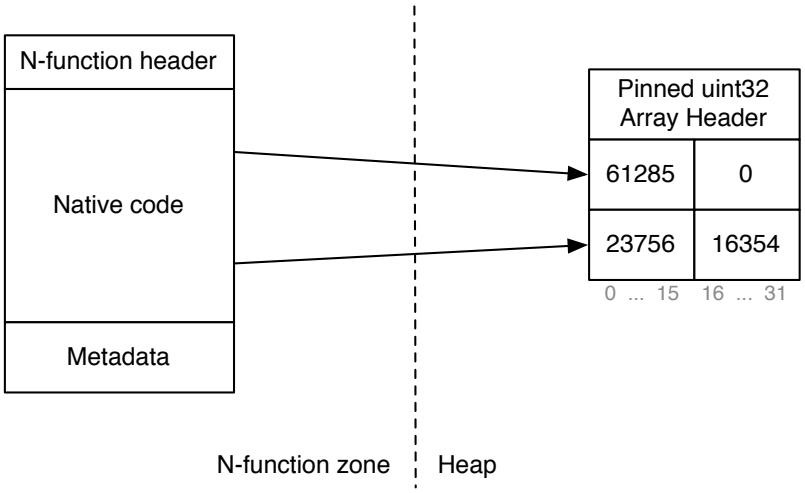


Figure 5.1: Unoptimised n-function with two profiling counters

executable) zone. The n-function is composed of a header, which encodes different properties of the n-function such as its size, the n-function native code and metadata to be able to introspect the n-function.

As the n-function from Figure 5.1 requires two counters, an array with two 32 bits wide fields is allocated on heap. Each 32 bits counter field is split in two. The high 16 bits are used for the counter just before the conditional branch, precising how many times the branch has been reached. The low 16 bits are used for the counter just after the branch, precising how many times the branch was not taken. The native code has direct references to the counter addresses. The n-function header has also a reference to the pinned array, not shown on the figure to avoid confusion, which is used to reclaim the pinned array memory when the n-function is garbage collected. In the example, we can see that the first branch is always taken (the counter increased when the branch is not taken is at 0 while the branch has been reached 61285 times).

5.1.2 Extended bytecode set

Our architecture requires an extended bytecode set to support all the new operations permitted only in optimised v-functions. The new operations were described in one of our paper [Béra 2014]. The extended bytecode set design relies on the assumption that only a small number of new instructions are needed for Cogit to produce efficient machine code. Four main kind of instructions were introduced:

- **Guards**: Guards are used to ensure an optimisation-time assumption is valid

at runtime. If the guard fails at runtime, the dynamic deoptimisation routine is triggered.

- **Object unchecked accesses**: Normally variable-sized objects such as arrays or byte arrays require bound checks to allow a program to access their fields. Unchecked access directly reads the field of an object without any checks. Instructions to access the size of a variable-sized object without any checks are included.

- **Unchecked arithmetics**: Arithmetic operations need to check for the operand types to know what arithmetic operation to execute (integer operation, double operation, etc.). Unchecked operations are typed and do not need these checks. In addition, unchecked operations do not perform any overflow check.

- **Unchecked object allocation and stores**: Normal object allocations do many different things in addition to memory allocation, such as the initialization of all fields to nil which is not needed if all fields are set immediately after to other values. Normal stores into objects go through a write barrier to make sure that the store does not break any garbage collector invariant. Such a write barrier can be ignored in specific cases.

As discussed in the previous chapter, optimised v-functions may be interpreted but we made sure that their interpretation is very uncommon. We designed the unsafe operations to be efficient when an optimised n-function is generated and executed. We did not design the unsafe operations for efficient interpretation.

Designing the operations for efficient native code generation is quite different from designing them for efficient interpretation. For efficient native code generation, we encoded the unsafe operations in multiple bytes to be able to provide extra information to Cogit on how to produce good native code for the instruction. This strategy is not very good for the interpreter as it needs additional time to decode the multiple bytes. A good design for efficient interpretation would have been to encode performance critical instructions in a single byte.

5.1.3 Call-backs

As Scorch optimiser and deoptimiser are written in Pharo while the rest of the VM is written in Slang, the VM needs a special way to activate them.

Pharo has an array of registered objects which can be accessed both from the VM and the language, as described in Section 3.2. We registered two new selectors. One is activated by the VM when a hot spot is detected in an unoptimised n-function. The other one is activated when a guard failed in an optimised n-function.

A method is implemented in Smalltalk for each selector in the class used for reified stack frames. In our case, these methods activate respectively Scorch optimiser or Scorch deoptimiser.

5.1.4 Machine code introspection

To extract runtime information from a n-function, we added a new primitive method called *sendAndBranchData*. SendAndBranchData is activated with no arguments and fails if the receiver is not a v-function. If the v-function was compiled by Cogit and therefore has an associated n-function, the primitive answers runtime information present for the function, if the v-function was not compiled, the primitive fails. The runtime information includes types and functions met in each inline cache and the profiling counter values. This information can be then used by Scorch to speculate on types and basic block usage.

Cogit had an introspection API used for multiple features such as inline cache relinking or debugging. The cache and counter values are read by the implementation of a small extension on top of Cogit API for introspection. In unoptimised n-functions, Cogit generates an inline cache for each virtual call [Deutsch 1984, Hölzle 1991], relinked at runtime each time a new receiver type is met for the call. The caches can be read using low-level platform-specific code, similar to the code used for relinking. Cogit generates for each profiling counter a call to the hot spot detection Slang routine, used to trigger the hot spot detection routine. This call is annotated with the corresponding virtual instruction pointer, allowing Scorch to know to which conditional branch in the v-function each profiling counters correspond to.

The new primitive iterates over the n-function, collecting for each virtual call and each conditional branch the virtual instruction pointer as well as respectively type and profiling information. Because the data is collected from Slang, it's not convenient to build complex data structures using multiple different kind of objects (Each object's class internal representation would need to be specifically known by both the VM and Scorch). To keep things simple, the primitive answers an array of arrays, each inner array containing the virtual program counter of the instruction, and a list of types and v-functions targetted by the inline cache or the number of times each branch has been taken.

5.1.5 Scorch

In the implementation of our architecture, the bulk of the work is the design and the implementation of Scorch optimiser. The thesis is centered around Sista in general and there are no big innovative features in Scorch.

Scorch optimiser is implemented as a traditional optimising compiler. It translates the v-functions to optimise into a single static assignment intermediate representation, represented as a control flow graph. Conditional branch and send instructions are annotated with the runtime information provided by Cogit to speculate on what method to inline and on basic block usage.

The optimisations performed are very similar to the ones performed in other optimising JITs such as V8 Crankshaft optimising JIT [Google 2008]. Scorch starts by speculating on types to inline other functions. Guards are inserted to ensure speculations are valid at runtime. Once inlining is done, Scorch performs multiple standard optimisations such as array-bounds check elimination with the ABCD algorithm [Bodík 2000] or global value numbering. Scorch also attempts to postpone allocation of objects not escaping the optimised v-function from runtime to deoptimisation time (or completely removes the allocation if the object is never required for deoptimisation).

Scorch back-end generates an optimised v-function from the intermediate representation. The phi instructions from single static assignment are removed, using instead temporary locations assigned multiple times. For each instruction, Scorch determines if the computed value needs to be stored to be reused, and if so, if it can be stored as a spilled value on stack or as a temporary variable. The deoptimisation metadata is attached to the optimised v-function to be able to restore multiple frames.

Scorch deoptimiser is much simpler than the optimiser. It reads the deoptimisation metadata for a given program counter in the optimised v-function. The metadata consists of a list of objects to reconstruct, including closures and reified stack frames. The objects are reconstructed by reading constant values in the metadata or reading the optimised stack frame values. Once the objects are reconstructed, execution can resume in the restored bottom frame.

Scorch dependency manager is also much simpler than the optimiser. It keeps track for each optimised function of the list of selectors it depends on. If a method with one of these selectors is installed, all the optimised functions dependant are discarded.

5.2 Optional language evolutions

Five major evolutions were introduced in the language in addition to the required evolutions to allow Scorch to produce more efficient optimised v-functions:

1. A new memory manager for efficient n-function generation.
2. A new bytecode set to leverage encoding limitations.
3. A register allocation algorithm in Cogit.

4. A write barrier feature to be able to mark objects as read-only.

5. A new closure implementation to be able to optimise closure more efficiently.

5.2.1 New memory manager

The first version of Sista was built on the existing VM with a minimum number of modifications. One of the main issues met was related to the memory representation of objects. The existing memory manager was designed and implemented before the implementation of Cogit for a pure interpreter VM [Ingalls 1997]. The representation of objects did not allow Cogit to produce efficient accesses to object fields in native code.

One problem for example was the encoding of the class field in an object. It could be encoded in three different ways:

- *Immediate classes:* A very limited subset of classes, included SmallInteger, have their instances encoded in the pointer to the object itself. As all objects are aligned in memory for efficient access to their fields, the last few bits (the exact number depends on the alignment) of a pointer to an object are never set. By setting some of these last few bits, the memory manager can encode a class identifier. SmallInteger for example are encoded by setting the last bit of the pointer.

- *Compact classes:* A limited set of classes, up to 15 classes, had their instances encoding their classes as an index in a 4-bits field in the first word of the object's header. The memory manager had access to an array mapping the indexes to the actual classes.

- *Other classes:* All the other instances encoded their class as a pointer to the class object, encoded in an extra pointer-sized field in the header of the object.

In practice, Cogit compiles many type-checks. In unoptimised n-functions, type-checks are generated mainly in inline caches. In optimised functions, type-checks are generated for deoptimisation guards. For each type-check, the native code generated needed three paths to find out which one of the three encodings was used for the instance which was type-checked, to finally compare it against the expected type. In addition, as many instances encoded their class as a pointer to the class object while class objects can be moved by the garbage collector in memory, cogit needed to annotate the expected type to correctly update the pointer value during garbage collection. Overall both the generated n-function and the garbage collector were slowed by the memory representation.

To solve this problem, a new memory manager was implemented and deployed in production [Miranda 2015]. The new representation of objects in memory allows the generation of very efficient n-functions. For example, there are now only two ways for an instance to access its class, the class is either immediate or compact. Compact class indexes are stored in the instances in a 22-bit fields, allowing over four millions different concrete classes. Cogit does not need any more to annotate type-checks when generating the n-function as an indirection index is referred instead of the class object.

In addition to the generation of efficient n-functions, other problems non directly related to the thesis were present in the existing memory manager (poor support for large heaps, slow scavenges, etc.) which were solved with the new memory manager.

5.2.2 New bytecode set

The existing Pharo bytecode set had multiple encoding limitations [Béra 2014]. For example, jumps (forward, backward and conditionnal) were able to jump over 1024 bytes at most. Such limitations are very rarely a problem while compiling normal Smalltalk code due to coding convention encouraging developers to write small functions. However, the optimised function produced by Scorch includes many inlined functions and in some case the limitations were a problem. As the bytecode set already needed to be extended to support the new unsafe operations, we designed a complete new bytecode set instead of just adding the new operations to leverage encoding limitations.

5.2.3 Register allocation

To allocate registers, Cogit simulates the stack state during compilation. When reaching an instruction using values on stack, Cogit uses a dynamic template scheme to generate native instructions. The simulated stack provides information such as which values are constants or already in registers. Based on this information, Cogit picks one of the available templates for the instruction, uses a linear scan algorithm to allocate registers that do not need to be fixed into specific concrete registers and generates native instructions.

The existing linear scan algorithm was very naive and limited. It was very efficient because registers are not live across certain instructions that are very common in unoptimised code. Specifically, registers cannot live across these three instructions:

1. *virtual calls:* All registers are caller-saved.

2. *backjumps:* Backjumps are interrupt points.

3. *Conditional branches:* If the branch is on a non-boolean, a slow path is taken to handle the case requiring to spill the registers.

However, these instructions are not that common in optimised code. Most virtual calls are inlined. Some backjumps are annotated not to require an interrupt check. Some conditional branches are removed because one branch has never been used and others are annotated as branching on a value which is guaranteed to be a boolean. Registers can therefore stay live across many more instructions and the register allocation algorithm has more impact on native code quality.

We wrote a new linear scan register algorithm, performing better under register pressure. The most difficult part is to correctly keep registers live across conditional branches. At each control flow merge point, the register state has to be the same in both branches or Cogit needs to generate additional instructions to spill or move registers.

5.2.4 Read-only objects

One of the main problem encountered while trying to improve the performance of the optimised v-functions generated by Scorch was literal mutability. In most programming languages, if the program executes a simple double[3] addition between two double constants the compiler can compute at compile-time the result. In Pharo, as literals are mutable, one of the double constants may be accessed through reflective APIs and mutated into another double, invalidating the compile-time result.

To solve the problem, we introduced read-only objects. With this feature, a program can mark any object as read-only. Such read-only objects cannot be modified unless the program explicitly reverts them to a writable state. Any attempt to modify a read-only object triggers a specific call-back in Smalltalk, similarly to the hot spot detection and guard failure call-backs. The modification failure routine can, for example, revert the object to a writable state, perform the modification and notify a list of subscribers that the modification happened. This feature was introduced with limited overhead to the existing runtime [Béra 2016a].

In Sista, literals are read-only objects by default. Any attempt to modify a read-only literal is caught by the runtime and Scorch is notified. If the literal was used for compile-time computation, corresponding optimised v-functions are discarded. Thanks to this technique, traditional compiler optimisations can be applied to Smalltalk.

[3] We use *double* to discuss double-precision floating-point in this paragraph.

5.2.5 Closure implementation

Another important problem encountered when implementing Scorch was related to the implementation of closures. The existing closures were implemented in a way that the closure's v-functions were inlined into their enclosing v-function. This led to multiple problems as it was difficult to optimise a v-function without having to rewrite the v-functions of all the closures that could be instantiated inside the v-function. This was increasing the complexity of the optimiser and required very expensive object manipulation at deoptimisation time to correctly remap all the v-functions of the closures created inside an optimised function. The closure implementation was also complicating the code of Cogit: Cogit could not compile a virtual method without compiling all the virtual functions of the closures the virtual method could compile. This induced significant complexity also to introspect n-functions and to activate closures.

To solve these issues, we designed a new closure implementation. In the new implementation, closures have v-functions separated from their enclosing environment v-functions. Scorch can optimise independently methods and closures. We were able to reduce the complexity of Cogit, both when it is used as a baseline JIT and as a back-end for Scorch.

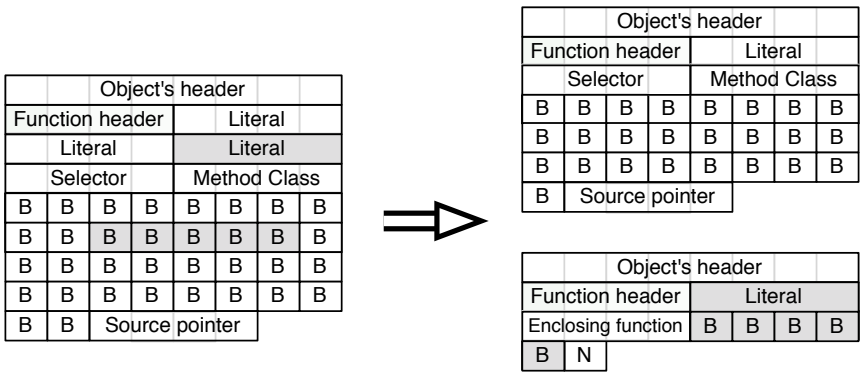


Figure 5.2: Old and new closure representation

Figure 5.2 shows on the left the old closure function representation and on the right the new representation. In the old representation, the closure function literals and bytecodes are present inside the enclosing function. The creation of a closure in the v-function interpreter requires to jump over the closure instructions. All the function metadata, normally present in the function header, can be computed by disassembling the closure creation instruction or the closure bytecodes.

Any change on the closure function impacts the enclosing function and any change on the enclosing function impacts the closure function. When Cogit compiles the v-function to n-function, it compiles both the function and all the inner closure functions at the same time. In the new representation, the two functions are independent. Each function has its own function header. Both functions can be changed independently. Cogit compiles separatedly each v-function to n-function. The closure function has no source pointer as it can be fetched from the enclosing function (the last literal of the compiled block).

5.3 Work distribution

To implement all these language evolutions, I did not work alone. I did some evolutions alone, but others were done by Eliot Miranda and the remaining evolutions were implemented by both of us. Tim Felgentreff deserves also a mention as he implemented the Squeak speed center, allowing to benchmark Sista. During the performance evaluation, he tuned Scorch to get better performance.

Task	Estimated Complexity	% done by Eliot Miranda	% done by me
01. Hot spot detection in Cogit	10	99	1
02. Interpreter and Cogit extensions to support new instructions	15	20	80
03. VM call-backs to trigger Scorch optimiser and deoptimiser	2	50	50
04. *sendAndBranchData* primitive	5	99	1
05. Scorch	1000	1	99
06. Spur memory manager	800	99	1
07. New bytecode set	10	40	60
08. Register allocation in Cogit	20	75	25
09. Read-only objects	10	20	80
10. New closure implementation	20	50	50

Figure 5.3: Work distribution of each language evolution

Figure 5.3 describes the work distribution of each language evolution implemented. Only the distribution between Eliot Miranda and I is shown as, excluding

the validation infrastructure, it represents more than 99% of the total work done to get Sista working. Sista was implemented on top of the existing production Pharo VM and Figure 5.3 does not show the work done to get that existing VM working.

The biggest evolution to support Sista is the implementation of Scorch, which I did mostly alone. The other very large contribution is the Spur memory manager, which was optional to have Sista working but important for the overall VM performance. The Spur memory manager was mostly done by Eliot Miranda.

Conclusion

This chapter described the runtime evolutions required in the Pharo runtime to support Sista. The next chapter discusses how Scorch is able to optimise its own code and under which constraints.

CHAPTER 6

Metacircular optimising JIT

Contents

By design, Scorch optimiser and deoptimiser are written in Smalltalk and are running in the same runtime as the optimised application. This design leads to recursion problems similar to the ones existing in metacircular virtual machines, detailed in this chapter.

As Pharo is currently single-threaded, it is not possible to run Scorch in a concurrent native thread. To optimise code, Scorch requires either to temporarily interrupt the application green thread or to postpone the optimisation to a background-priority green thread as described in Section 4.2.1. The deoptimiser cannot however postpone the deoptimisation of a frame as it would block completely the running application. The deoptimiser has necessarily to interrupt the application green thread to deoptimise the stack.

Hot spots can be detected in any Smalltalk code using conditional branches, including Scorch optimiser code itself (as Scorch is written in Smalltalk). When a hot spot is detected in the optimiser code, the optimiser interrupts itself and starts to optimise one of its own functions. While doing so, the same hot spot may be detected again before the optimised function is installed, leading the optimiser to interrupt itself repeatedly. This problem is discussed in Section 6.1.

A similar problem exists for the deoptimiser. As Scorch deoptimiser is written in Smalltalk, its code base may get optimised. One of the optimisation-time speculation may be incorrect at runtime, leading the deoptimiser to require the deoptimisation of one of its own frames. In this case, the deoptimiser calls itself on one of its own frames, which may require to deoptimise another frame for the same function, leading the deoptimiser to call itself repeatedly. This second problem is detailed in Section 6.2.

We call this issue where the optimiser or the deoptimiser calls itself repeatedly as the *meta-recursion* problem. The expression meta-recursion comes from other works [Chiba 1996, Denker 2008] where similar problems are present.

The optimiser and deoptimiser have different constraints. It is possible to disable temporarily the optimiser while the application is running. In the worst case, a disabled optimiser leads some functions not to be optimised, but the application keeps running correctly. However, the deoptimiser cannot be disabled temporarily while the application is running. Indeed, an application requiring deoptimisation cannot continue to execute code until the deoptimisation is performed. As the optimiser and the deoptimiser have different constraints, they need different solutions for the meta-recursion problem.

This chapter explains the design used to avoid the meta-recursion issue in both the optimiser and the deoptimiser. Section 6.1 describes how the problem is solved for the optimiser by temporarily disabling it in specific circumstances. Section 6.2 shows how the deoptimiser solves the problem by using a code base completely independent from the rest of the system that cannot be optimised, to never require to be deoptimised. Section 6.3 discusses similar designs in other VMs and compares our solution to other solutions when relevant.

6.1 Scorch optimiser

Scorch optimiser is activated by the VM when a hot spot is detected. As Pharo is single-threaded, the optimiser is activated by interrupting one of the application green threads. The optimiser chooses, based on the current stack, a v-function to optimise. Once the v-function to optimise is chosen, the optimiser gets started in critical mode: it attempts to generate an optimised v-function in a limited time period. If it succeeds, the optimised v-function is installed and used by further calls on the function. If the optimiser fails to generate the optimised v-function in the limited time period, it adds the v-function to a background compilation queue. In any case, the application is then resumed. When the application becomes idle, if the background compilation queue is not empty, Scorch gets activated in background mode. It produces and installs optimised v-functions for each function in the compilation queue without any time limit.

6.1.1 Meta-recursion issue

As Scorch optimiser is written in Smalltalk, it can theoretically optimise its own code. In practice, if it happens, it may lead to a meta-recursion. Indeed, each time Scorch tries to optimise a function, before reaching the point where it can install the optimised function, it may interrupt itself to start optimising one of its own functions. If a hot spot is detected in the optimiser code each time it attempts to optimise anything, then the optimiser never reaches the point where it can install an optimised function.

Figure 6.1 shows the problem. On the left, in the normal optimisation flow, the application is interrupted when a hot spot is detected. The optimiser generates an optimised v-function, installs it and the application resumes. On the right, in the meta-recursion issue, the application is also interrupted when a hot spot is detected. While the optimiser is generating an optimised function, a hot spot is detected in the optimiser code. The optimiser then restarts to optimise one of its own functions, but another hot spot (potentially the same one) is detected in the optimiser code: the optimiser keeps restarting the optimisation of one of its own function.

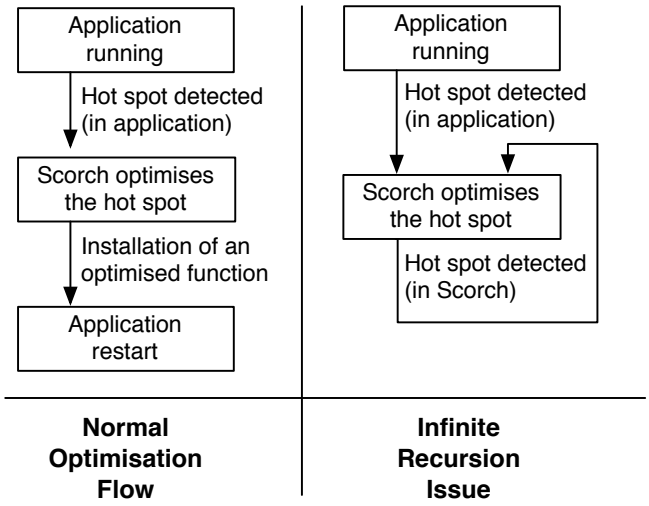
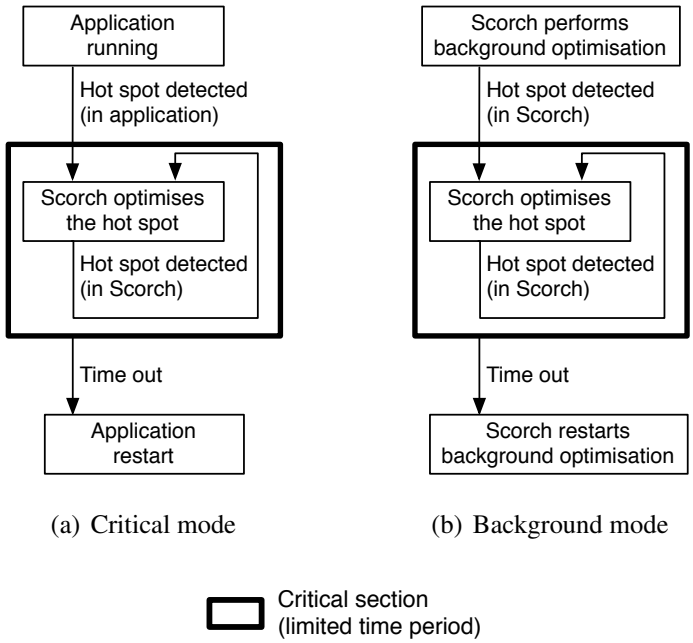


Figure 6.1: Meta-recursion problem during optimisation

This problem has different consequences depending if the optimiser is started in critical or in background mode. In practice, the meta-recursion issue leads to a massive performance loss that we detail in the following paragraphs.

Critical mode. In critical mode, the optimiser has a limited time period to optimise code. If the meta-recursion issue happens, the optimiser spins until the time period ends as shown in Figure 6.2(a). The application is then resumed without any optimised function installed. The application gets drastically slower as it gets interrupted for the full critical mode time period without gaining any performance from those interruptions.

Background mode. When the application becomes idle, the optimiser is started in background mode to optimise functions in the background compilation queue. In this case, the optimiser always successfully generates and installs optimised functions. However, the optimisation of a function is very slow. Indeed, while the



(a) Critical mode (b) Background mode

Figure 6.2: Meta-recursion problem in the two optimisation modes

optimiser is running in background mode, it activates itself multiple times in critical mode when detecting hot spots in its own code. Each time it happens, the optimiser spins in critical mode for the time period allowed as shown on Figure 6.2(b). If the optimiser is started many times on itself during background optimisation, the optimisation of the function may take a significant amount of time.

Eventually, the optimiser optimises most of its own code correctly through the background mode. Once done, both modes can work correctly as the optimiser cannot be triggered on already optimised code.

The problem is therefore that the application executed gets really slow at start-up because of time wasted spinning in critical mode. Peak performance takes a long time to reach because the optimiser successfully installs code only in background mode or when the meta-recursion issue does not happen in critical mode.

6.1.2 Current solution

The solution we implemented is to disable the optimiser when it is running in critical mode. This way, no meta-recursion can happen. This design has a significant advantage: it is quite simple both conceptually and implementation-wise, while it completely avoids the meta-recursion problem. It has however a major drawback: the optimiser code cannot be optimised when run in critical mode any more, which

is the most common way the optimiser is run. Of course, the optimiser may use core libraries that can be optimised. For example, the optimiser uses collections such as arrays. If the application optimised is also using the same collections, and it is very likely that an application would use arrays, the array code base may get optimised. Then, the optimiser ends up using an optimised version of arrays. However, the code specific to the optimiser is not optimised.

To implement the solution, we changed the VM call-back activating the optimiser to uninstall itself upon activation. When a hot spot is detected but the call-back is not installed, the VM resets the profiling counters which has reached the hot spot threshold and restarts immediately the execution of the application. Resetting the counters avoids the hot spot to be detected repeatedly each time the execution flow reaches it, which slows the runtime without any purpose. Then, we changed the optimiser to install back the call-back when it resumes the application, after installing optimised code or adding a function to the background queue.

Our implementation effectively disables the optimiser only when it is running on critical mode. When hot spots are detected, they are optimised or postponed without any issue as the optimiser is disabled in critical mode and the application resumes correctly. However, when the optimiser is started in background mode, it is not disabled. Hence, in this case, hot spots are detected in the optimiser code and the optimiser is sometimes interrupting itself (the optimiser in critical mode interrupts the optimiser in background mode) to optimise its own code.

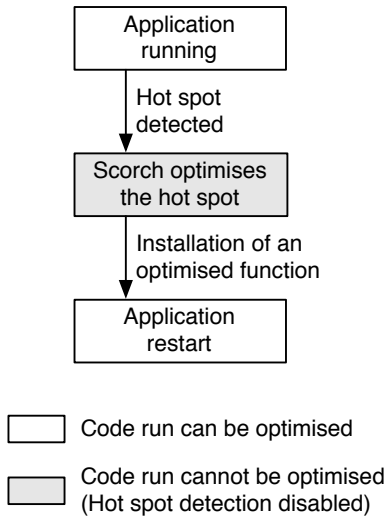


Figure 6.3: Hot spot detection disabled in critical mode.

Figure 6.3 shows the solution. The application code can be optimised by Scorch, but Scorch cannot optimise its own code when run in critical mode as hot

spot detection is disabled. When Scorch is started in background mode, then hot spots are detected (as in any application) and the optimiser code can get optimised.

This solution is implemented, stable and works fine. Multiple benchmarks run with significant speed-up over the normal VM (This will be detailed in Chapter 8). In general, in our production VM, simplicity is really important to keep the code base relatively easy to maintain. For each added complexity in the VM we evaluate if the complexity is worth the benefit. This solution is very valuable to us because it is fairly simple to understand and to maintain. Hence, the optimising JIT may move to production with this design. The next section discusses alternative solutions, which are more complex but allow, at least partially, the optimiser to optimise its own code when run in critical mode.

6.1.3 Discussion and advanced solutions

The solution described in the previous section is working but has one major drawback: Scorch optimiser cannot optimise itself in critical mode. Indeed, hot spots detected inside the optimiser in critical mode are completely ignored and the corresponding profiling counters are reset. If the optimiser attempts to optimise code later, it may get confused by some counter values which were reset (basic block usage is incorrectly inferred in this case, in the worst case, a branch may be speculated as unused whereas it is frequently used).

Decay strategy. Instead of resetting entirely the counters, we could implement a decay strategy, by for example dividing the current counter values by two. We did not go in this direction because the counters are currently encoded in 16 bits while the hot spot threshold is set to the maximum value. Due to the 16 bits encoding limitation, not completely resetting the counters leads to the detection of many hot spots, repeatedly, without any optimisation happening slowing down the optimiser at start-up. Further analysis in this direction are required to conclude anything.

Partial disabling. Another naive approach is to postpone the optimisation to background mode when the meta-recursion issue happens in critical mode. In our design, it is quite difficult to do so. Indeed, when the VM call-back starts the optimiser, it provides only a reification of the current stack as detailed in Section 4.2.3. The optimiser then needs to search the stack to select a function to optimise, and only then it can add a function to optimise to the background compilation queue.

The stack is modified upon execution and may reference a very large graph of objects, so it is very difficult to save it efficiently for the optimiser to search it later in the background green thread. In addition, as discussed in Section 4.2.3, there is no simple and quick heuristic to figure out the best function to optimise based on the current stack.

It is however possible, once the optimiser has found what function to optimise, to add it to the background compilation queue. Therefore, we believe that instead of disabling the entire optimiser while it is running in critical mode, we could instead disable it only during the stack searching phase in critical mode and postpone the optimisation to the background green thread instead if the function to optimise has already been found. This way, only hot spots detected during the stack search would be ignored, while the rest of the optimiser would be optimised at the next idle pause. As the stack search phase represents less than 1% of the optimiser execution time, this approach looks very promising.

Figure 6.4 shows the solution proposed. When a hot spot is detected, Scorch is activated on a stack to optimise and starts by searching a function to optimise. During this phase, the optimiser is disabled to avoid the meta-recursion issue. Once the function to optimise is found, Scorch optimises it. During this phasis, if a hot spot is detected, the optimiser searches a function to optimise and directly appends it to the background compilation queue. Once the function is optimised, the optimised v-function is installed and the application can resume.

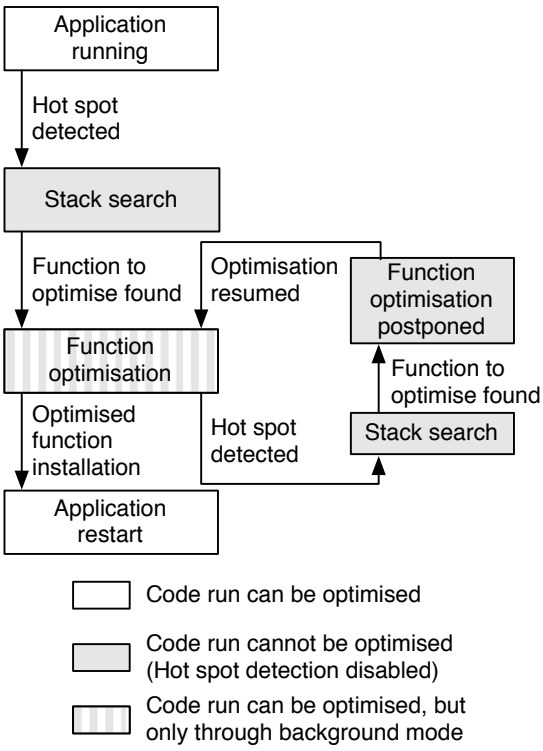


Figure 6.4: Partial disabling of the optimiser.

Ahead-of-time optimisation. Alternatively, we could consider optimising Scorch optimiser code ahead-of-time.

As Sista allows one to persist optimised code (this is discussed in details in the following chapter, Chapter 7), the optimiser code could be optimised ahead-of-time. The optimised optimiser code would be shipped to production, and no runtime optimisation would happen on the optimiser code in production.

To generate the optimised code ahead-of-time, one way is to preheat the optimiser through warm-up runs. For example, the optimiser can be given a list of well-chosen functions to optimise. This way, all hot spots inside the optimiser would be detected ahead-of-time and optimised.

Alternatively, the optimiser's code could be optimised statically by calling itself on its own code, using types inferred from a static type inferencer instead of types inferred from the runtime. This solution has a significant cost in our case as we have to implement and maintain a library to infer types.

6.2 Scorch deoptimiser

The deoptimiser can be activated in multiple situations. If an optimisation time assumption is invalid at runtime, a deoptimisation guard fails and Cogit triggers a call-back to deoptimise the stack. In addition, multiple development tools in the language, such as the debugging tools, may call the deoptimiser to introspect the stack.

6.2.1 Meta-recursion issue

As Scorch deoptimiser is written in Smalltalk, its code base may get optimised. If one of the optimisation-time speculation is incorrect at runtime, the optimised frame requires the deoptimiser to restore the non-optimised stack frames to continue the execution of the program.

In the case where the deoptimiser functions are optimised, the deoptimiser may call itself on one of its own frames to be able to continue deoptimising code. If at each deoptimisation the deoptimiser calls itself on one of its own frames, the deoptimisation will never terminate because the deoptimiser needs to call itself again. The application then gets stuck in an infinite loop[1].

Figure 6.5 shows the problem in the case where the deoptimiser is triggered by a guard failure. On the left, in the normal deoptimisation flow, the application is interrupted when a guard fails. The deoptimiser recreates the unoptimised stack frames from the optimised stack frame and edits the stack. The application can

[1]The infinite loop is theoretical: in practice, a maximum number of stack frames can be allocated or the application runs out of memory.

then resume with unoptimised code. On the right, in the meta-recursion issue, the application is also interrupted when a guard fails. However, while the deoptimiser is deoptimising the stack, another guard fails in the deoptimiser code. The deoptimiser then restarts to deoptimise one of its own frame, but another guard fails in its own code. The deoptimiser keeps restarting the deoptimisation of one of its own frame and the application gets stuck in an infinite loop.

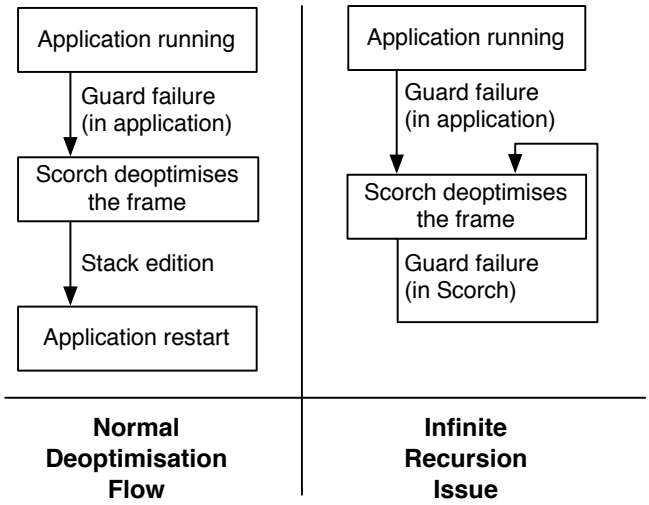


Figure 6.5: Meta-recursion problem during deoptimisation

Unlike the optimiser, the deoptimiser cannot be disabled temporarily, otherwise no application green thread requiring deoptimisation would be able to keep executing code. To solve this meta-recursion problem, we implemented two solutions.

The first solution, described in Section 6.2.2, restores the runtime in a "recovery mode" when recursive deoptimisation happens. In recovery mode, no optimised function can be used (the runtime relies entirely on the v-function interpreter and the baseline JIT). This solution was used successfully for a subset of the current set of benchmarks. However, this solution was not very good for applications and benchmarks using multiple green threads.

We then designed and implemented a second solution, detailed in Section 6.2.3, that is still in use now. The second solution consists in keeping all the deoptimiser code in a library completely independent from the rest of the system that cannot be optimised.

6.2.2 Recovery mode

As a first attempt to solve the meta-recursion issue for the deoptimiser, Scorch was modified to keep a recovery copy of each method dictionary where optimised v-functions are installed. The recovery copies include only unoptimised v-functions. We added a global flag, marking if a deoptimisation is in progress. If the deoptimiser is activated while a deoptimisation is in progress (this can be known thanks to the global flag), the deoptimiser falls back to recovery mode. To do so, the deoptimiser uses the primitive become: (described in Section 9.2.2) to swap all method dictionaries with their recovery copy and disables the optimiser not to optimise anything in the recovery copies. The deoptimiser can then deoptimise the stack without calling itself repeatedly as it now uses only unoptimised functions. Once the stack is deoptimised, the deoptimiser restores the method dictionaries with the optimised v-functions and re-enables the optimiser.

Figure 6.6 shows how the recovery mode solves the meta-recursion issue in the deoptimiser. The application is interrupted when a guard fails, and the deoptimiser recreates the unoptimised stack frames from the optimised stack frame. If another guard fails in the deoptimiser code, Scorch falls back to recovery mode, and deoptimises its stack frame using only unoptimised code. Once done, Scorch deactivates the recovery mode to resume the deoptimisation of the application optimised frame. In the worst case, the deoptimiser may switch multiple times to recovery mode to deoptimise the application frame. One the unoptimised stack frames are recreated, the application can resume with non-optimised code.

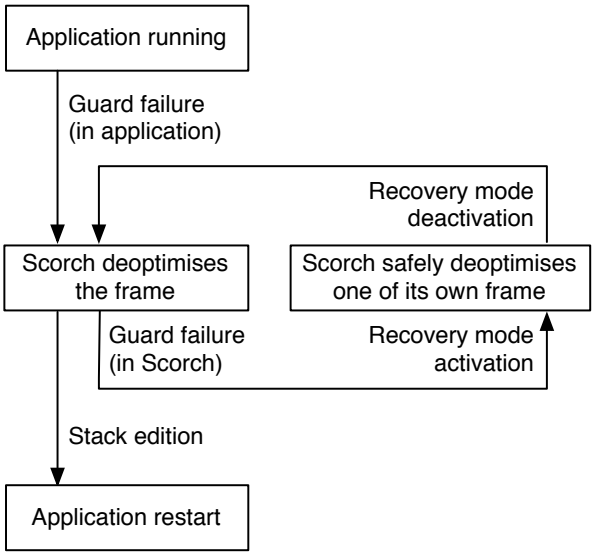


Figure 6.6: Meta-recursion problem solved with recovery mode

With this solution, most of the deoptimiser code can be optimised. Indeed, if a meta-recursion happens, Scorch is able to switch to recovery mode and executes correctly the code. Although most of the deoptimiser code can be optimised, not all the code can be. The Smalltalk code executed between the guard failure call-back and the point where recovery mode is activated cannot be optimised. Such code is not protected by the recovery mode and may suffer from the meta-recursion issue. To avoid the problem, we marked a very small list of functions so they cannot be optimised.

Issues. We were able to run most benchmarks with this solution. However, some benchmarks showed significant slow-down during deoptimisation. Moreover, other benchmarks (the ones using multiple green threads) were crashing. With this solution, we had two major problems.

The first problem is that switching to recovery mode requires to edit many look-up caches in the VM to use unoptimised functions. Each look-up cache entry referencing an optimised function needs to be edited to refer back to the unoptimised function. Once the deoptimisation in recovery mode is terminated, the caches need to be updated again to reference the optimised functions. Updating all the caches can take a significant amount of time. In addition, in our implementation, the inline caches are directly written in the native code of each n-function. Each update in the native code requires the processor to partially flush its instruction cache. The recovery mode therefore slows down the application for a short while due to the time spent for the VM to update the caches but also to the cpu instruction cache flush and miss (due to the flush).

The second and main problem is that several of our benchmarks use multiple green threads. In this case, the global flag approach to mark if a deoptimisation is in progress does not work as multiple deoptimisations may happen concurrently. In a normal application, when such a problem happens, a developer uses semaphores or other green thread management features present in the language to make the code green thread safe. In the case of Pharo, most of the code related to green thread management and scheduling is written in Pharo itself. Using this code to mark if a deoptimisation is in progress in a process would require it to be marked in the list of functions that cannot be optimised, as this is used in-between the guard failure call-back and the activation of the recovery mode. We did not want to disable optimisations on code present in the semaphores and the process scheduler as such code may be performance critical in some applications. Forbidding the optimisation of such code seemed to be too restrictive. We concluded that this approach could not work for our production environment.

6.2.3 Independent library

As a second solution, we designed the whole deoptimiser as a completely independent Smalltalk library. The deoptimiser may use primitive operations but is not allowed to use any external function. All the deoptimiser classes are marked: their functions do not have profiling counters and Scorch optimiser is aware that the optimisation of such functions is not allowed. The deoptimiser code is therefore not optimised at runtime, it is running using only the v-function interpreter and the baseline JIT. As the deoptimiser code cannot be optimised and cannot use any external function that could be optimised, the meta-recursion issue cannot happen.

Constraints. This solution has three main constraints:

1. *The deoptimiser code cannot be optimised at runtime.* All the deoptimiser classes are marked not to be optimised at runtime, forbidding the deoptimiser code to reach high performance like the rest of the code. This constraint may be considered as minor as deoptimisation is uncommon. Not optimising the deoptimiser code may therefore not be a problem as the overall time spent in executing its code is very low. In addition, this problem can be partially solved by optimising the deoptimiser code ahead-of-time using Scorch only with optimisations that do not require deoptimisation guards and type information inferred statically. As the library is quite small (500 LoC) and has a very strong invariant (it cannot call any external function), a type inferencer can be easy to implement, very precise and efficient.

2. *The deoptimiser code needs to be completely independent.* Only primitive operations can be used directly because any external function called may be optimised, potentially leading to the meta-recursion problem. To remove the dependencies to external functions, we analysed what libraries the deoptimiser depends on. Most dependencies were very small (for instance accessors to the reification of stack frames) and they could be removed by duplicating some code. However, one dependency was a problem: the deoptimiser uses arrays and dictionaries to deoptimise the stack. Those classes are used in many applications so we cannot just forbid to optimise their code base. To solve this issue, we created a minimal array and a minimal dictionary as part of the deoptimiser library that cannot be optimised. The two collections have a separate code base from the classical collection library, which needs to be maintained in parallel to the core collections. As mentioned in the previous paragraph, the overall optimiser code base is very small (500 LoC including the duplicated collections), so we believe it is possible to maintain it without too much effort.

3. *Work on the deoptimiser is very tedious.* A simple thing such as logging a string in the deoptimiser code for debugging requires to call an external function and may lead to the meta-recursion problem. Understanding and debugging the deoptimiser code is therefore quite difficult.

Although the constraints are important, we were able to run all our benchmarks with this design. We believe this implementation is good enough to move to production, at least on the short term.

6.3 Related work

To have an optimising JIT optimising its own code and encounter the meta-recursion problem we discussed in this chapter, the optimising JIT has to be written in one of the languages it can optimise and run in the same runtime than the optimised application. Such an optimising JIT is not common.

Many production VMs are entirely written in a low-level language such as C++ [Google 2008, Webkit 2015]. The optimising JIT cannot optimise its own code in such VMs. Other VMs such as the ones written with the RPython toolchain [Rigo 2006] are written in a language that the optimising JIT could optimise, but the production VMs are compiled ahead-of-time to native code, hence the optimising JIT does not optimise its own code at runtime in production. Metacircular VMs [Ungar 2005, Alpern 1999] are entirely written in a language they can run.

Many metacircular VMs, such as Klein [Ungar 2005], do not feature an optimising JIT[2]. There are two main projects where the optimising JIT optimises its own code at runtime. The first project is the Graal compiler [Oracle 2013, Duboscq 2013] which can be used both in the context of the Maxine VM [Wimmer 2013] and the Java Hotspot VM [Paleczny 2001]. The Graal compiler effectively optimises its own code at runtime as it would optimise the application code. The other project is the Jalapeño VM [Alpern 1999], now called Jikes RVM, features a runtime compiler that can optimise its own code at runtime.

6.3.1 Graal optimising JIT

The Graal runtime compiler [Oracle 2013, Duboscq 2013] is an optimising JIT written in Java, which is able to optimise its own code at runtime. Graal can be used in different contexts, with different solutions to the meta-recursion problem. Initially, the Graal runtime compiler was designed and implemented as part of the Maxine VM [Wimmer 2013], a metacircular Java VM. Graal was then extracted

[2] In 2009, Adam Spitz reported some work in the direction of an inlining JIT compiler in Klein on the project web page, but there has been no further news about it since then.

from Maxine and it can now work on top of the Java Hotspot VM [Paleczny 2001]. Graal can be used in two main ways on top of the Hotspot VM. On the one hand, it can be used as an alternative optimising JIT, replacing the Java Hotspot optimising JIT written in C++. On the other hand, it can be used as a special purpose optimising JIT, optimising only specific libraries or application while the rest of the Java runtime is optimised with Hotspot optimising JIT.

Graal-Hotspot architecture. In our context, the most relevant use-case is when the Graal compiler is used as an alternative optimising JIT on top of the Java Hotspot VM. The interpreter and baseline JIT tiers are in this case present in Java Hotspot VM, written in a low-level language (C++) and compiled ahead-of-time to native code. This is very similar to our design, where the Pharo interpreter and baseline JIT are also compiled ahead-of-time to native code. The optimising JIT are in both cases written in the language run by the VM (Graal in Java and Scorch in Smalltalk), they can optimise their own code and they need to interface with the existing VM to trigger runtime compilation and to install optimised code.

In the Graal-Hotspot runtime, when a hot spot is detected, code in the Hotspot VM (written in C++) searches the stack for a function to optimise. Once the function is chosen, Hotspot adds it to a thread-safe compilation queue. The Graal compiler is run in different native threads concurrently to the application native threads. Graal takes functions to optimise from the compilation queue, generates concurrently optimised n-function and hands them over to the Hotspot VM for installation. The optimised n-functions handed by Graal to Hotspot respect the Graal Java native interface [Grimmer 2013]. They include deoptimisation metadata that the hot spot VM is able to understand. When dynamic deoptimisation happens, code written in the Hotspot VM (in C++) is responsible for the deoptimisation of the stack using the metadata handed at installation time.

In our work, Scorch optimiser is able to optimise its own code according to different constraints. The stack search code can be optimised only if a hot spot is detected while the optimiser is running in background mode. The code responsible for the optimisation of a function can be optimised only indirectly through the background mode. Scorch deoptimiser code cannot be optimised at all.

Comparison between the architectures. Table 6.1 sumarizes the similarities and the differences between the architectures. We call *the base VM* the core elements of the VM excluding the optimising JIT: the interpreter, the baseline JIT and the GC. We then distinguish three parts in the optimising JIT:

1. *The stack search:* responsible to find a function to optimise based on a stack with a hot spot.

Table 6.1: Comparison between the Sista and the Graal-Hotspot architectures

	Sista	Graal-Hotspot
Base VM	Compiled and optimised AOT	Compiled and optimised AOT
Stack Search	Application runtime, optimised at runtime if hot spot detected in background mode	Compiled and optimised AOT
Optimisation of a Function	Application runtime, optimised at runtime through the background mode	Application runtime, optimised at runtime unconditionally
Deoptimisation of a Frame	Application runtime, not optimised at runtime	Compiled and optimised AOT

2. *The optimisation of a function:* responsible to generate an optimised function-based on a non-optimised function and runtime information. This is by far the largest and most complex part of the optimising JIT.

3. *The deoptimisation of a frame:* responsible to recreate non-optimised stack frames from an optimised frame.

In both architectures, the base VM is optimised and compiled ahead-of-time (AOT) to executable code. The stack search code is also optimised and compiled ahead-of-time in the case of the Graal-Hotspot architecture, while it is running in the same runtime as the application in the case of Scorch. The stack search code can be optimised in Scorch if the hot spot is detected while Scorch is running in background mode. The code responsible to generate the optimised function is in both cases running in the same runtime than the application optimised. In the case of Graal-Hotspot, the code can be optimised the same way than the application code. In the case of Sista, the optimisation of the function is postponed to the background compilation queue. Lastly, the code responsible for the deoptimisation of a frame is optimised and compiled ahead-of-time in the case of the Graal-Hotspot architecture, while it is running in the same runtime than the application in the case of Sista, but cannot be optimised at runtime.

Notable differences. Having the stack search and deoptimisation code in Smalltalk, even with constraints, allow us to change part of the design such as the deoptimisation metadata without having to recompile the VM. This is an advantage in our context, as we want Smalltalk developer to be able contribute to the

project without having to recompile the VM or look into low-level details. It can however be seen as a draw-back as the constraints, decreases the performance of the deoptimisation of stack frames and the start-up performance of the stack search.

The other difference is how the optimisation of a function is managed. In Sista, Scorch needs to postpone the optimisation to background mode while the Graal-Hotspot architecture allows one to optimise the function like any application function. Our constraints comes from the fact that Pharo is currently single-threaded. In Graal-Hotspot, the optimisation of a function is done in a concurrent native thread. This allows one to avoid having a critical and a background mode, as well as allowing the optimising JIT to optimise this part of the code without any constraints. The solution of Graal-Hotspot has less constraints, but it requires multithreading support.

6.3.2 Jikes RVM

Jikes RVM [Alpern 1999, Arnold 2000] optimising runtime compiler is written entirely in Java and can optimise Java code, including its own code. However, it is not currently able to use runtime information to direct its optimisations and does not generate deoptimisation guards[3], so it is not that relevant in our context.

The runtime compiler uses however an interesting technique [Arnold 2000] to choose what function to optimise. Instead of profiling counters, Jikes RVM uses an external sampling profiling native thread. Based on the profiling samples, the profiling thread detects what function should be optimised and adds it to a thread-safe compilation queue. The optimising runtime compiler can then start other native threads which take functions to optimise from the compilation queue, optimise and install them. With this technique, the functions to optimise are chosen entirely concurrently. The application is not interrupted, at any time, to search the stack for a function to optimise or to detect a hot spot. This technique therefore allows one to write the hot spot detection in the same runtime than the application optimised, while it can still be optimised the same way than the application run. We did not investigate in this direction because our VM is currently single threaded.

Conclusion

In this chapter we discussed the meta-recursion issue. If a hot spot is detected inside the optimiser code, the optimiser may call itself indefinitely to try to optimise itself. The deoptimiser has a similar issue when it needs to deoptimise its own code. The problem exists because the optimiser and the deoptimiser are implemented in

[3] "The provided AOS [Adaptive Optimisation System] models do not support Feedback-Directed Optimizations" http://www.jikesrvm.org/ProjectStatus/

Smalltalk and are running in the same runtime and the same native thread than the application they optimise and deoptimise respectively. The main issue is related to meta-recursion.

The optimiser solves this issue by disabling itself when it runs in critical mode (interrupting temporarily the application green thread to perform the optimisation). The deoptimiser has to solve the problem differently as it cannot be disabled temporarily or Smalltalk code cannot be executed any more. The deoptimiser avoids the problem by being written using a small number of classes, independent from the rest of the system, that cannot be optimised nor call any external function.

The next chapter explains how the runtime state is persisted across multiple VM start-ups, including the running green threads and the optimised functions.

CHAPTER 7

Runtime state persistence across start-ups

Contents

This chapter describes how the Sista VM persists the runtime state across multiple VM start-ups, including the running green threads and the optimised code, for the VM to reach peak performance quickly after start-up. Section 7.1 focuses on the main issue solved by the runtime state persistence: the start-up performance of many VMs today is significantly worse than their peak performance. Several cases where the start-up performance is a problem are described. Section 7.2 discusses the interactions between existing snapshots in Pharo and Sista, including how the optimised code and the running green threads are persisted across VM start-ups. Section 7.3 compares our approach to existing VMs. Few VMs attempt to persist the runtime state across multiple start-ups, but some VMs include solutions to improve start-up performance, solving the same problem.

7.1 Warm-up time problem

The most important problem solved by persisting the runtime state across start-up is the warm-up time problem, i.e., the time wasted by the VM at each start-up to reach peak performance. Depending on use-cases, the warm-up time may or may not matter. In long-running applications, the warm-up time required to reach peak performance is negligible compared to the overall uptime of the application. However, when applications are started or updated frequently and are short-lived, warm-up time matters.

We give three examples where the virtual machine start-up time matters, representative of industrial use-cases.

Short-lived slaves. Modern large distributed applications run on hundreds, if not thousands, of machines such as the slaves one can rent on Amazon Web Services. Slaves are usually rented per hour, though now some services such as Amazon Lambda allows one to rent a slave for small amount of time down to a hundred milliseconds. Depending on usage, the application automatically rents new slaves or frees used slaves. This way, the application scales up very well as thousands of slaves are rent if needed, while a few slaves are rented if the application is used very little. The server cost of the distributed application depends purely on how much computation power is needed: one pays for slaves when they are used and does not pay when they are not used.

The problem is that to reduce the cost to the minimum, the application needs to rent a slave when needed, but frees it at the 10th of second where the slave is not used to avoid paying for an unused slave. Doing so implies having potentially very short-lived slaves when the application usage varies greatly from a 10th of a second to the next 10th of a second. Slaves could have a life expectancy of a couple hundred milliseconds. Now, if the slave does not have enough time to reach peak performance in its short life-time, the money saved by not paying for unused slaves is dominated by the money wasted in computation power used in the optimising JIT to reach peak performance. To have very short-lived slaves worth it, the time between the slave start-up and the peak performance of the application used has to be as small as possible. A good VM for such kind of scenario is a VM where peak performance is reached immediately after start-up.

Mobile applications. In the case of mobile applications, the start-up performance matters because of battery consumption. During warm-up time, the optimising compiler recompiles frequently used code. All this compilation process requires time and energy, whereas the application is not run. In the example of the Android runtime, the implementation used JIT compilation with the Dalvik VM [Bornstein 2008], then switched to client-side ahead-of-time compilation (ART) to avoid that energy consumption at start-up, and is now switching back to JIT compilation because of the AOT (ahead-of-time compiler) constraints [Geoffray 2015]. These different attempts show the difficulty to build a system that requires JIT compilation for high performance but can't afford an energy consuming start-up time.

Web pages. Some web pages execute only a bit of Javascript code when opened, while other web pages use extensively Javascript in their lifetime (in this latter case, one usually talk about web application). A Javascript virtual machine has to reach peak performance as quickly as possible to perform well on web pages where only a bit of Javascript code is executed at start-up, while it has also to perform well on

long running web applications.

7.2 Snapshots and persistence

Snapshots are available in multiple object-oriented languages such as Smalltalk [Goldberg 1983] and later Dart [Annamalai 2013]. As discussed in Section 3.3, in our case we use Pharo which features snapshots. A snapshot is a serialized form of all the objects present at a precise moment in the runtime. Everything is an object in Pharo, including green threads or v-functions. To start-up Pharo, the virtual machine loads all the objects from a snapshot and resumes the execution based on the green thread that was active at snapshot time. This is how Pharo is normally launched.

Pharo development workflow. A Pharo programmer does not modify source code in files as many other programming languages. For development, Pharo is started using a snapshot which includes development tools, user interface elements and a source code to v-function compiler. When started, the programmer can open the development tools and write or edit the source code of a function. Then, the compiler generates a v-function from the source code (which is implicitely added in the heap). Then, the programmer may take a new snapshot, which includes the changes made. Further start-ups, on the new snapshot, features the changes made by the programmer. We note that in this paragraph we described the normal development flow of a Smalltalk programmer: this is not a workflow the programmer can do but does not normally do, this is how all programmers currently do it.

Application to Sista. In the context of Sista, optimised v-functions are installed at runtime by Scorch. Those functions effectively modify the current heap of objects. Hence, when a new snapshot is taken, optimised v-functions are persisted. The next start-up of Pharo will use directly the optimised v-functions.

Green threads and snapshots. To persist running green threads in a platform-independent way, to take a snapshot, the VM reifies each stack frame to a context object as explained in Section 3.1. Effectively, this means that only v-frames are persisted: n-frames are converted to v-frames to be part of the snapshot. N-frames cannot be persisted in any case as snapshots are machine-independent (or the instruction pointer would not necessarily be correct when the snapshot is started on a different architecture).

When the VM starts from a snapshot, all running green threads are executed using the v-function interpreter. However, once a function is called multiple times or a loop is interpreted a certain number of iterations, Cogit generates a n-function

for the corresponding v-function (optimised or not), and the runtime resumes with the n-function.

Conclusion. To conclude, programmers normally work on Pharo by modifying the current heap, for example by adding new v-functions to method dictionaries of classes, and then take a snapshot of the heap to save their code. V-functions are persisted in snapshot but n-functions are not.

In Sista, the optimising JIT is the combination of Scorch, which generates and installs optimised v-functions, and Cogit, which generates and installs n-functions.

Optimised v-functions generated by Scorch are, without any additional work, persisted across multiple start-ups as part of the snapshot like unoptimised v-functions. N-functions generated by Cogit are never persisted.

Most of the compilation time is currently spent in Scorch, hence, if Pharo is started using a snapshot including optimised v-functions, Pharo can reach peak performance very quickly. Green threads using optimised functions are persisted in the snapshot in the form of optimised and unoptimised v-frames (n-frames are converted to v-frames, they cannot be persisted because they refer to n-functions).

7.3 Related work

This section discusses other strategies implemented in other VMs and research projects to decrease the warm-up time.

7.3.1 Preheating through snapshots

Dart snapshots. The Dart programming language features snapshots for fast application start-up. In Dart, the programmer can generate different kind of snapshots [Annamalai 2013]. Since that publication, the Dart team have added two new kind of snapshots, specialized for iOS and Android application deployment, which are the most similar to our snapshots.

Android. A Dart snapshot for an Android application is a complete representation of the application code and the heap once the application code has been loaded but before the execution of the application. The Android snapshots are taken after a warm-up phase to be able to record call site caches in the snapshot. The call site cache is a regular heap object accessed from machine code, and its presence in the snapshot allows one to persist type feedback and call site frequency.

In this case, the code is loaded pre-optimised with inline caches prefilled values. However, optimised functions are not loaded as in our architecture. Only unoptimised code with precomputed runtime information is loaded.

iOS. For iOS, the Dart snapshot is slightly different as iOS does not allow JIT compilers. All reachable functions from the iOS application are compiled ahead-of-time, using only the features of the Dart optimising compiler that don't require dynamic deoptimisation. A shared library is generated, including all the instructions, and a snapshot that includes all the classes, functions, literal pools, call site caches, etc.

This second case is difficult to compare to our architecture: iOS forbids machine code generation, which is currently required by our architecture. A good application of our architecture to iOS is future work.

Cloneable VMs. In the work of Kawachiya and all [Kawachiya 2007], an extension of the Java VM allows one to clone a running VM to quicken the start-up. The heap is cloned, in a similar way to our snapshot, but the n-functions are also cloned. Cloning n-functions improves start-up performance over our approach, but the clone is processor-dependent: there is no way of cloning with their approach a Java runtime from an x86 machine to an ARMv6 machine. Our approach requires slightly more warm-up time to quickly compile our optimised n-functions to n-functions, but our approach is platform-independent.

7.3.2 Fast warm-up

An alternative to snapshots is to improve the JIT compiler so the peak performance can be reached as early as possible. The improvements would consists of decreasing the JIT compilation time by improving the efficiency of the JIT code, or have better heuristic so the JIT can generate optimised code with the correct speculations with little runtime information.

Tiered architecture. One solution, used the Webkit VM [Webkit 2015], is to have a tiered architecture with many tiers. In the version of Webkit in production from March 2015 to February 2016 [Webkit 2015], the code is:

- interpreted by a bytecode interpreter the first 6 executions.
- compiled to machine code at 7th execution, with a non-optimising compiler, and executed as machine code up to 66 executions.
- recompiled to more optimised machine code at 67th execution, with an optimising compiler doing some but not all optimisations, up to 666 executions.
- recompiled to heavily optimised machine code at 667th execution, with an optimising compiler using LLVM as a backend.

At each step, the compilation time is greater but the execution time decreases. This tiered approach (4 tiers in the case of Webkit), allows to have good performance from start-up, while reaching high performance for long running code. This kind of approaches has also draw-backs: the VM development team needs to maintain and evolve four different tiers.

Saving runtime information. To reach quickly peak performance, an alternative of saving optimised code is to save the runtime information. The Dart snapshot saves already the call site information in its Android snapshots. Other techniques are available.

In Strongtalk [Sun Microsystems 2006], a high-performance Smalltalk that has never reached production, it is possible to save the inlining decision of the optimising compiler in a separate file. The optimising compiler can then reuse this file to make the right inlining decision the first time a hot spot is detected.

In the work of Arnold and all [Arnold 2005], the profiling information of unoptimised runs is persisted in a repository shared by multiple VMs. This allows new VM starting-up to re-use the information of other and previous VM runs to direct compiler optimisations.

Saving runtime information decreases the warm-up time as the optimising JIT can speculate accurately on the program behavior with very few runs. However, on the contrary to our approach, time is still wasted optimising functions.

Saving machine code. In the Azul VM Zing [Systems 2002], available for Java, the official web site claims that "operations teams can save accumulated optimisations from one day or set of market conditions for later reuse" thanks to the technology called *Ready Now!*. In addition, the website precises that the Azul VM provides an API for the developer to help the JIT to make the right optimisation decisions. As Azul is closed source, implementation details are not entirely known.

However, word has been that the Azul VM reduces the warm-up time by saving machine code across multiple start-ups. If the application is started on another processor, then the saved machine code is simply discarded. We did not go in this direction to persist the optimisation in a platform-independent way (in our architecture, starting the application on x86 instead of ARMv5 does not require the saved optimised code to be discarded), but we have a small overhead due to the bytecode to machine code translation at each start-up. In addition, we believe it's very difficult to persist correctly machine code compared to persisting bytecodes.

Aside from Azul, the work of Reddi and all [Reddi 2007] details how they persist the machine code generated by the optimising JIT across multiple start-ups of the VM. JRockit [Oracle 2007], an Oracle product, is a production Java VM allowing to persist the machine code generated by the optimising JIT across

multiple start-ups.

We did not go in the direction of machine code persistence as we wanted to keep the snapshot platform-independent way: in our architecture, starting the application on x86 instead of ARMv5 does not require the saved optimized code to be discarded, while the other solutions discussed in this paragraph do. However, we have a small overhead due to the bytecode to machine code translation at each start-up. In addition, the added complexity of machine code persistence over bytecode persistence should not be underestimated: our approach is simpler to implement.

Ahead-of-time analysis. In the work of Krintz and Calder [Krintz 2001], static analysis done ahead-of-time on Java code generates annotations that are used by the optimising JIT to reduce compilation time (and hence, the warm-up time). As for the persistence of runtime information, on the contrary to our approach, time is still wasted at runtime optimising functions.

Ahead-of-time compilation. The last alternative is to pre-optimise the code ahead-of-time. This can be done by doing static analysis over the code to try to infer types. Applications for the iPhone are a good example where static analysis is used to pre-optimise the Objective-C application. The peak performance is lower than with a JIT compiler if the program uses a lot of virtual calls, as static analysis are not as precise as runtime information on highly dynamic language. However, if the program uses few dynamic features (for example most of the calls are not virtual) and is running on top of a high-performance language kernel like the Objective-C kernel, the result can be satisfying.

Conclusion

This chapter discusses how the runtime state is persisted across multiple start-ups, improving the performance during start-up. The next chapter validates Sista, mainly through performance evaluation in a set of benchmarks. The validation chapter also evaluates the Sista VM performance when the runtime state is persisted across multiple start-ups.

CHAPTER 8

Validation

Contents

To validate our architecture, we evaluated in Section 8.1 the execution time of a set of benchmarks on the Pharo VM with and without Sista. With Sista, the VM is up to five times faster than the current production VM. The VM was also evaluated in different configurations to show the overhead of profiling counters and the persistance of optimisations across multiple start-ups. Section 8.2 discusses other strategies we implemented to validate our architecture.

8.1 Benchmarks

This section first details the methodology used for benchmarking, then describes each benchmark used, analyses the results and concludes.

8.1.1 Benchmark methodology

We evaluate our architecture on a variety of benchmarks from the Squeak/Smalltalk speed center [Felgentreff 2016a] that is used to monitor the performance of the Cog VM and other compatible virtual machines for Squeak and Pharo. The benchmarks are adapted from the Computer Language Benchmarks Game suite [Gouy 2004] and contributed by the Smalltalk community. We have selected these benchmarks to give an indication of how certain combinations of operations are optimised with our architecture. Although they tend to over-emphasize the effectiveness of certain aspects of a VM, they are widely used by VM authors to give an indication of performance.

We consider the results on the Pharo VM with four different configurations:

1. *Cog* is the existing VM (interpreter and Cogit as the baseline JIT). Cog represents the baseline performance.

2. *Cog+Counters* is the existing VM with profiling counters without any additional optimisation. Cog+Counters is used to evaluate the profiling counters overhead.

3. *Sista Cold* is the Sista VM started on a snapshot without any optimised v-function.

4. *Sista Warm* is the Sista VM started on a snapshot that already contains optimised v-functions.

We iterated each benchmark for 100 iterations or 60 seconds, whichever came last, and measured the last 10 iterations. For Sista Warm, we start with an already optimised snapshot. For Sista Cold, we only enable the optimising compiler before the last 10 iterations (this way, the warm-up from Cog's baseline JIT is not included in measuring the warm-up of Sista Cold). The benchmarks were run on an otherwise idle Mac mini 7,1 with a Dual-Core Intel Core i7 running at 3GHz and 16 GB of RAM. We report the average milliseconds taken per iteration, with the confidence interval given for the 90^{th} percentile. For these measurements, we configured the VM to detect hot spots when a profiling counter reaches 65535 iterations (they are encoded as *int16*, so this is currently the maximum) and we allow the optimiser up to 0.4 seconds to produce an optimised method in critical mode (the benchmarks are run consecutively without any idle time so optimisation in background mode is not considered). We use a high counter value and allow for a long optimisation time, because as the optimisations are saved across start-ups we believe it does not matter that much if the VM takes a long time to reach peak performance.

We have found these values to produce good performance across a variety of benchmarks. Because Scorch is written in Smalltalk itself, it is possible to configure various other optimisation options depending on the application, for example, to emphasize inlining, to produce larger or smaller methods, or to spend more or less time in various optimisation steps.

8.1.2 Benchmark descriptions

This section briefly describes for each benchmark what the benchmark does and what operations are used the most.

A*. The A* benchmark is a good approximation for applications where many objects collaborate. It measures parsing of large strings that define the layout of the tree of nodes, message sending between each node, arithmetic to calculate costs, and collection operations. In the benchmark, we alternately parse and traverse

two different graphs with 2,500 and 10,000 nodes, respectively. It is also a good benchmark for inlining block closures that are used in iterations.

Binary tree. The binary tree benchmark allocates, walks and deallocates binary trees. The benchmark is parameterized with the maximum tree depth, which we have set to 10. The benchmark is focused on object allocation, tree traversal and object deallocation.

JSON parsing. We test a JSON parser written in Smalltalk as it parses a constant, minified, well-formed JSON string of 25 Kilobytes. This benchmark is heavy on nested loops and string operations, as well as a lot of parsing rules that call each other.

Richards. Richards is an OS kernel simulation benchmark that focuses on message sends between objects and block invocations. We ran this benchmark with the customary idle task, two devices, two handler tasks, and a worker, and filled the work queue of the latter three.

K-Nucleotide. This benchmark reads a 2.4 MB DNA sequence string and counts all occurrences of nucleotides of lengths 1 and 2, as well as a number of specific sequences. It is a benchmark meant to test the performance of dictionaries in different languages, but serves well to test our inlining of small methods into loops. The benchmark runs much slower than the others due to the large input, taking over 4 minutes to complete.

Thread ring. The Thread ring benchmark switches from thread to thread (green threads) passing one token between threads. Each iteration, 503 green threads are created and the token is passed around 5,000,000 times.

N-body. N-body models the orbits of Jovian planets, using a symplectic integrator. Each iteration simulates 200,000 interactions between the Jovian planets. The n-body benchmark is heavy on float operations, and ideal benchmark to highlight the inlining that Sista performs.

DeltaBlue. DeltaBlue is a constraint solver, it tests polymorphic message sends and graph traversal. Each iteration tests updating a chain of 5000 connected variables once with equality constraints and once with a simple arithmetic scale constraint.

Spectral Norm. Calculating the spectral norm of a matrix is heavy on floating point and integer arithmetic as well as large arrays. The arithmetic is expected to inline well, but since large allocations take place throughout this benchmark, the performance benefit for Sista is expected to be smaller.

Mandelbrot. This benchmark calculates the Mandelbrot set of on a 1000x1000 bitmap. It is implemented in only one method with nested loops that almost exclusively calls primitive float methods and thus is a good candidate for Sista optimisations.

Meteor. This benchmark solves the meteor puzzle by recursively trying to fit puzzle pieces together using an exhaustive search algorithm.

8.1.3 Results

Figure 8.1 shows all the measurements in the form of graphs. The y-axis of each graph is the average time in milliseconds to run one iteration of the corresponding benchmark, so the smaller the column is, the fastest the benchmark is run. The exact values of the measurements are reported in Figure 8.2. The following paragraphs describe each a benchmark and its corresponding performance measurements. We distinguish three categories of benchmarks.

Quick start-ups. A*, Binary tree, JSON parsing, Richards, and K-nucleotide reach quickly peak performance. The difference between Sista Cold and Sista Warm is minimal, as even from a cold state, the VM is able to reach peak performance during the first few runs out of the ten runs. We can however see that the error margin in the Sista Cold is greater, as the first few runs have lower performance.

Slow start-ups. Thread ring, N-body, Delta blue and Meteor require multiple runs to reach peak performance. The average performance of the ten first runs is clearly not as good in Sista Cold that in Sista Warm, as a significant amount of these runs are not done at peak performance. In fact, in the case of N-body, ten runs is not even enough to reach peak performance. The error margin in Sista Cold is very important.

Very slow start-ups. In the case of Mandelbrot and Spectral Norm, ten runs is far from enough to reach peak performance. An important part of the execution time in the ten first runs is spent in compilation, leading the benchmark to be slower than the base VM. Once peak performance has been reached, Spectral Norm is

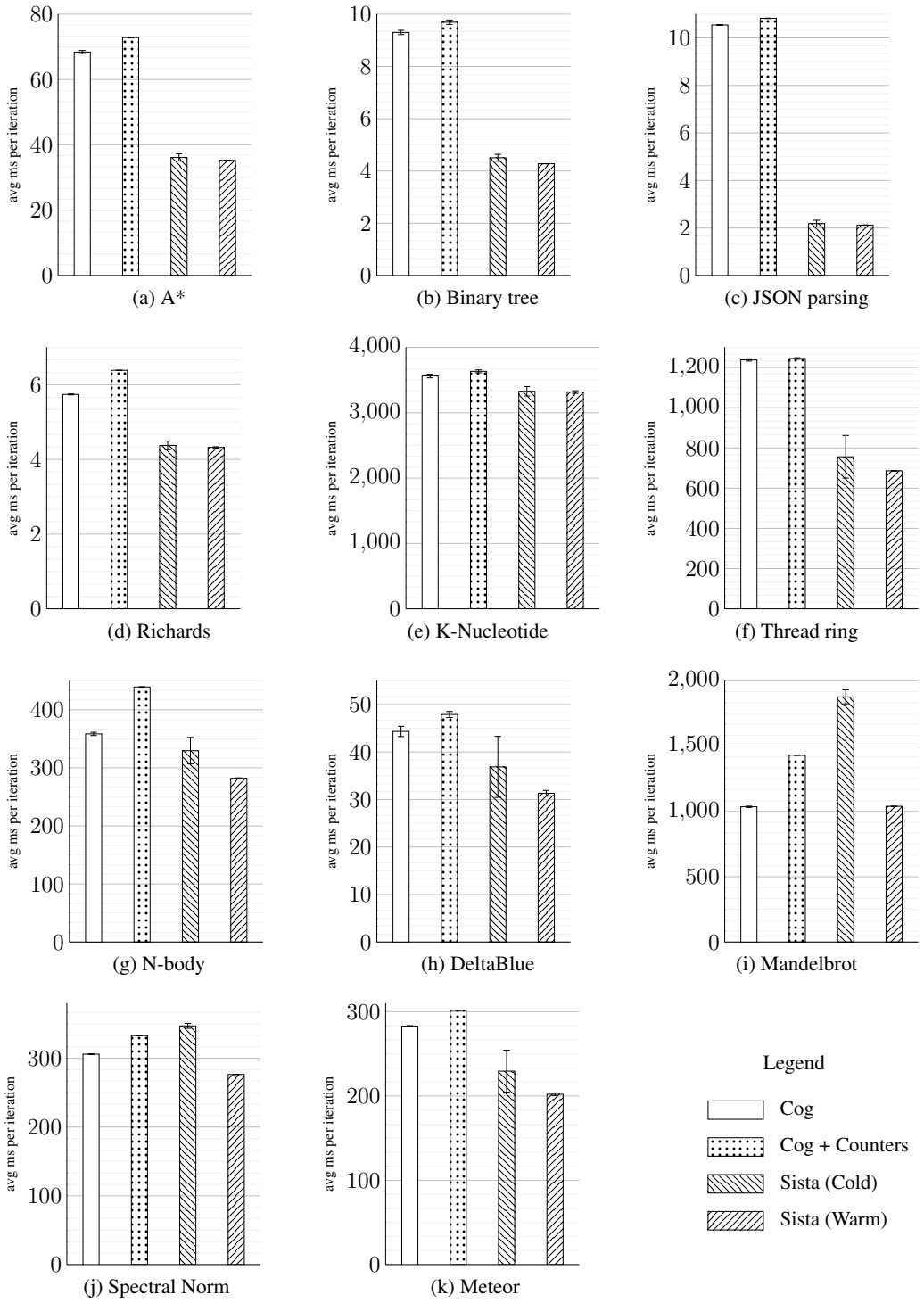


Figure 8.1: Benchmark measurements

Benchmark	Cog	Cog + Counters	Sista (Cold)	Sista (Warm)
A*	68.39 +- 0.485	72.833 +- 0.129	36.13 +- 1.12	35.252 +- 0.0479
Binary tree	9.301 +- 0.0811	9.694 +- 0.0865	4.505 +- 0.13	4.278 +- 0.0031
Delta Blue	44.33 +- 1.08	47.892 +- 0.638	36.86 +- 6.42	31.315 +- 0.601
JSON parsing	10.545 +- 0.0174	10.826 +- 0.0089	2.125 +- 0.140	2.121 +- 0.00826
Mandelbrot	1035.17 +- 4.99	1429.93 +- 1.2	1876.4 +- 53.4	1038.867 +- 0.604
Richards	5.7419 +- 0.0119	6.388 +- 0.0045	4.375 +- 0.115	4.3217 +- 0.0174
K-Nucleotide	3563.1 +- 28.6	3634.4 +- 21.8	3328.6 +- 71.8	3326.8 +- 20.0
Spectral Norm	305.983 +- 0.494	332.983 +- 0.485	347.15 +- 3.54	276.517 +- 0.347
Thread ring	1237.70 +- 5.73	1244.93 +- 3.89	756 +- 106	686.27 +- 1.56
N-body	358.42 +- 2.74	439.25 +- 0.484	329.5 +- 22.9	281.883 +- 0.836
Meteor	282.858 +- 0.658	301.60 +- 0.132	229.5 +- 24.8	202.07 +- 1.480

Figure 8.2: Benchmark results (standard errors in avg ms, 90% confidence interval)

10% faster than Cog. The peak performance of Mandelbrot is similar to Cog performance, only removing the overhead of profiling counter, because Mandelbrot is a floating-pointer intensive benchmark and we have not yet implemented floating-pointer optimisations in Sista.

8.1.4 Conclusion

For all benchmarks our approach shows significant performance improvements on the scales that we would expect given the various benchmark's properties. For these benchmarks, Sista is up to 80% faster. Since the baseline JIT compiles almost every method on second invocation, this is also the only warmup when a snapshot that was warmed up using our approach is launched. Thus, these benchmarks indicate that Sista can provide significant performance benefits without any additional warm-up time compared to the baseline compiler.

We ran our VM profiler to profile the VM C code, but as for real world application, the time spent in the baseline JIT compiler generating machine code from bytecode is less than 1% of the total execution time. As the runtime switches from interpreted code to machine code at second invocation for most functions and at first invocation for optimised functions, the time lost here is too small to be shown on our graphics. In fact, the time lost here is not significant compared to the variation so it is difficult to evaluate in our current setting. We believe that using a back-end doing many more machine low-level optimisations would increase the machine code compilation time and in this case we would be able to see a difference between the first run of pre-heated snapshot and second run as the VM still needs to produce the machine code for the optimised bytecoded functions.

Our optimiser is controlled by a number of variables that have been heuristi-

cally chosen to give good performance in a variety of cases. These include, among others, global settings for inlining depth, the allowed maximum size of optimised methods as well as methods to be inlined, as well as the time allowed for the optimiser to create an optimised method before it is aborted. We have found that for certain benchmarks, these variables can have a great impact. We are working on fine-tuning these default values, as well as enabling heuristics to dynamically adapt these values depending on the application.

8.2 Other validations

To evaluate our infrastructure, we tried two other innovative techniques in addition to measuring benchmarks. On the one hand, we built an experimental technique to validate runtime deoptimisation using partial evaluation. On the other hand, we built a type inferencer using the runtime information extracted from inline caches. The promising results of the type inferencer confirm that the runtime information is quite precise and should give Scorch valuable hints to direct compiler optimisations.

8.2.1 Experimental validation of the deoptimiser

The speculative optimisations in the Sista VM enable many performance optimisations. However, they also introduce significant complexity. The compiler optimisations themselves, as well as the deoptimisation mechanism are complex and error prone. To stabilize Scorch, we designed a new approach to validate the correctness of dynamic deoptimisation. The approach [Béra 2016b] consists of the symbolic execution of an optimised and a non optimised v-function side by side, deoptimising the abstract stack at each point where dynamic deoptimisation is possible and comparing the deoptimised and non optimised abstract stack to detect bugs.

Although this approach is interesting, the complexity required to maintain a good symbolic executor is significant compared to the time available for the maintenance of the overall VM. In other VMs such as V8 [Google 2008], dynamic deoptimisation is stabilised using a "deopt-every-n-time" approach: the program run is forced to deoptimise the stack regularly (every n deoptimisation point met). This approach is way simpler to maintain and finds in practice a similar number of bugs than the approach built. We are now using a "deopt-every-n-time" approach to validate the deoptimisation of functions.

8.2.2 Assessment of the runtime information quality

Thanks to the *sendAndBranchData* primitive method, any Pharo program, including Scorch, may request Cogit to provide the runtime information of a specific function. This runtime information is composed of the types met and functions called at each virtual call and the profiling counter values. To assess the quality of the runtime information provided for each virtual call, we built an approach called *inline-cache type inference* (ICTI) to augment the precision of fast and simple type inference algorithms [Milojković 2016].

ICTI uses type information available in the inline caches during multiple software runs, to provide a ranked-list of possible classes that most likely represent a variable's type. We evaluated ICTI through a proof-of-concept that we implemented in Pharo Smalltalk. Analyzing the top-n+2 inferred types (where n is the number of recorded runtime types for a variable) for 5486 variables from four different software systems (Glamour [Bunge 2009], Roassal2 [Araya 2013], Morphic [Fernandes 2007] and Moose [Gîrba 2010, Ducasse 2005, Ducasse 2000]) show that ICTI produces promising results for about 75% of the variables. For more than 90% of variables, the correct runtime type was present among the first ten inferred types. Our ordering shows a twofold improvement when compared with the unordered base approach [Pluquet 2009], *i.e.,* for a significant number of variables for which the base approach offered ambiguous results, ICTI was able to promote the correct type to the top of the list.

Based on these results, we believe the runtime information extracted from the first runs to be quite reliable. In any case, this information is used only to direct compiler optimisation: if the runtime information is not correct, the code is executed slower but correctly.

Conclusion

This chapter validates Sista by showing on a range of benchmarks the performance gain of the architecture (up to 5x). The next Chapter discusses the future works that could be considered based on this thesis.

CHAPTER 9

Future work

Contents

This chapter discusses future work related to Sista. The focus is on research-oriented future work, but the first two sections also mention engineering-oriented future work (for instance, moving the architecture to production).

Section 9.1 describes the evolution planned or considered for the overall architecture. Section 9.2 discusses specifically the optimisations that could be implemented in Scorch and Cogit. Section 9.3 details how the application of Sista on a real-world application requiring quick start-ups would be very valuable to validate further the architecture. Lastly, Section 9.4 discusses briefly the potentially low energy consumption of the Sista VM and how it could be valuable in specific use-cases.

9.1 Architecture evolution

This section details the evolution of the Sista architecture that are worth investigating. Section 9.1.1 describes the potential evolution of the interface VM-language: Sista is currently using an extended bytecode set to communicate between Scorch and Cogit, another representation may be better. Section 9.1.2 explains the on-going work to integrate Sista with the development tools. The evaluation of the memory footprint used by the Sista runtime is precised in Section 9.1.3. Section 9.1.4 discusses the current platform-dependencies of the persistence of the runtime state across start-ups and how these dependencies could be avoided. Section 9.1.5 briefly states the on-going work to move Sista to production.

9.1.1 Interface VM-language

Stack-based or register-based IR. The v-functions discussed in the thesis are currently encoded in a stack-based bytecode Intermediate Representation (IR). Stack-based IRs are usually considered as difficult to deal with in optimising compilers, so this can be seen as a problem. For this reason, Scorch decompiles the v-function to a register-based IR (similar to TAC[1], but some operations such as virtual calls may have more than three parameters), performs the optimisations and translates it back to the stack-based bytecode. Cogit then takes the stack-based bytecode as input, and translates it to a register-based IR to generate the n-function.

The stack-based extended bytecode sets has two main issues:

- **Loss of information:** Scorch IR has significant information about the instructions (liveness, uses) than the stack-based bytecode. This information is lost during the translation to the stack-based bytecode, while it may be valuable for Cogit to perform efficiently low-level optimisations.

- **redundant conversion:** When compiling using Sista's optimising JIT, unoptimised v-functions are compiled by Scorch to optimised v-functions and then by Cogit to optimised n-functions. It feels a bit redundant to take a stack-based IR (bytecode of the v-function), translate it to a register-based IR (Scorch IR), then translate it back to a stack-based IR (extended bytecode of the optimised v-function) and lastly translate it to a register-based IR (Cogit's IR), as shown in the left part of Figure 9.1. One could consider encoding optimised v-functions in a register-based IR, to avoid two translations.

However, the extended bytecode set was designed stack-based for two relevant reasons:

- **Compatibility:** The existing bytecode set is stack-based, and having the extended bytecode set stack-based allows us to generate optimised functions using existing instructions and not only new ones. This was very convenient to have quickly a working version of Sista (only the new unsafe operations used needed to work to have the architecture working).

- **Machine-independent:** A stack-based representation allows one to abstract away from low-level details such as the exact number of registers. To abstract away from the exact number of registers, both a stack-based IR and a register-based IR with an infinite number of registers are possible. Although at first look it seems the register-based solution is easier to deal with, experts such as the ones which designed WebAssembly [Group 2015] chose to use a stack-based IR over a register-based IR. It is not clear which solution is better.

[1] Three Address Code.

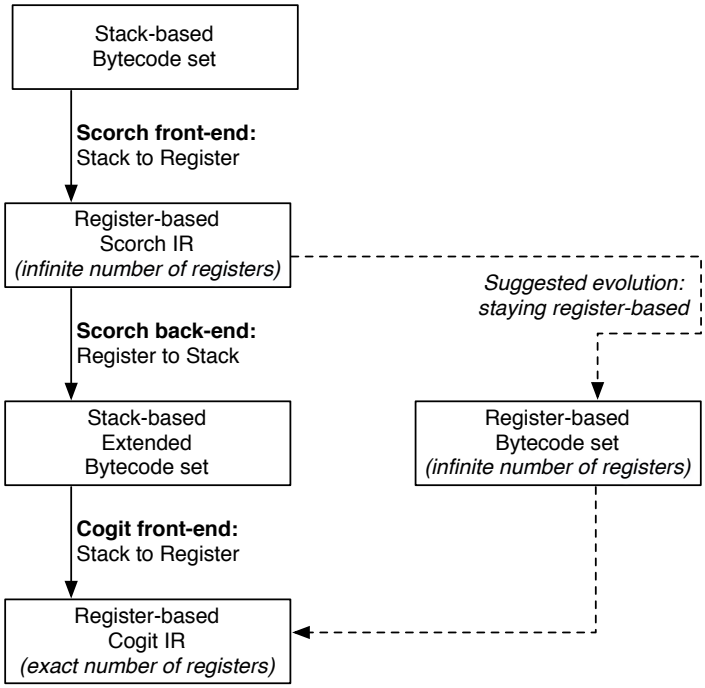


Figure 9.1: Redundant conversion

One future work is to design another extended bytecode set, register-based, and to evaluate the complexity of the representation in the back-end of Scorch and the front-end of Cogit, as shown on the right part of Figure 9.1.

Lower-level representation. The optimised v-functions are encoded in a quite high-level representation. For example, instructions like array accesses generate multiple native instructions. The extended bytecode set is quite high-level because it abstracts away from:

- the memory representation of objects.
- the processor used.

We believe abstracting away from the processor used was a good idea as it allows one to use snapshot to persist the optimised state across multiple start-ups. However, snapshots are already dependent on the memory representation of objects used, so one could implement a new representation of optimised v-functions with a lower-level representation, object representation dependent but still processor independent. This would allow Scorch to perform additional optimisations, such as better constant propagation (Some constants are currently hidden in high-level instructions), which are currently hard to support in Cogit.

Currently, only the VM code is aware of the memory representation of object. Hence, using such a lower-level representation would require to duplicate the knowledge about the memory representation of objects from the VM code-base to Pharo, so Scorch could be aware of it.

Summary. To summarize, one could change Sista so optimised v-functions would be encoded in a lower-level representation instead of the extended bytecode set. Such changes would make Scorch dependent of the memory representation of objects, while keeping it independent from the processor used. Some code would need to be duplicated from the VM code-base to Pharo to make Scorch aware of the internal memory representation of objects. Scorch would be able to produce more optimised code, performing optimisations that are currently difficult to implement in Cogit, yielding hopefully better performance.

Implementation. To implement such a solution, one could build a back-end for Scorch targeting the abstract assembly instructions set featured by the Cog VM, called Lowcode [Salgado 2016], similar to WebAssembly [Group 2015]. Lowcode features low-level instructions, compiled almost in most processors one-to-one to machine instructions. The future work is to implement such a back-end for Scorch and evaluate the complexity and the performance.

9.1.2 Development tools integration

In Sista, Scorch is an unoptimised v-functions to optimised v-functions compiler, running in the same runtime as the application. Although this design has several advantages, there is a major draw-back: when the programmer accesses the reification of a stack frame, depending on the optimisation state, an optimised frame might be shown.

In some cases, for example when the programmer is working on Scorch itself, it is relevant to show in the development tools the optimised frames. In other cases, for example when the programmer is working in an end-user application on top of Pharo, development tools should show only unoptimised frames by transparently deoptimising stack frames.

We are now adapting the debugging tools to request deoptimise stack frames when needed. To do so, we are adding a development tool setting: one may or may not want to see the stack internals, depending on what one wants to implement.

9.1.3 Memory footprint evaluation

When implementing an optimising JIT, it is relevant to evaluate the memory footprint of the optimised code and the deoptimisation metadata. In our context, such

an evaluation would be very interesting as:

- **Split architecture:** Due to the split between Scorch and Cogit, optimised functions are present both as v-functions and n-functions, and each of them has different deoptimisation metadata, potentially increasing the memory footprint.

- **Persistence:** As optimised v-functions are persisted across start-ups, it is interesting to know the size of the optimised v-functions and the corresponding deoptimisation metadata that is persisted.

Two future works are planned in this direction:

- Does the split in the optimising JIT induce memory overhead compared to a classical function-based optimising JIT, and how big is the overhead?

- What is the size of the optimised v-functions and the corresponding deoptimisation metadata that is persisted across start-up?

We did not evaluate the memory footprint because currently the deoptimisation metadata of Scorch is kept uncompressed. As Scorch deoptimiser requires to read deoptimisation metadata, compressing the metadata requires to write a decompressor which is, as all the deoptimiser code, independent from the rest of the system. This is possible but requires a certain amount of engineering work, which is the reason why we postponed it.

9.1.4 Platform-dependency

In Pharo, snapshots are independent of the processor and the OS used. It is proven as the same snapshot can be deployed on x86, ARMv6 and MIPSEL, as well as on Windows, Mac OS X, iOS, Linux, Android or RISC OS. However, the snapshots are dependent on the machine word size: 32 bit or 64 bit snapshots are not compatible. They are not compatible because the size of managed pointers is different, but also because the representation of specific objects, such as floating numbers, is different. It is however possible to convert offline a 32 bit snapshot to 64 bit and vice-versa.

As some optimisations related to number arithmetics, such as overflow checks elimination, depends on the number representations, Scorch is currently dependent on the machine word size. A fully portable solution would either need not to do optimisations on machine word specific number representations or deoptimise the affected code on start-up.

9.1.5 Productisation

With the current version, the Sista VM is able to run all our benchmark suite. We are now able to do part of our development with the development tools, written in Pharo, running on the Sista VM. We still have work to do in the integration with the development tools, especially the debugger, but it seems that the biggest part of the work has been done.

There are still some edge cases where the Sista VM is unstable, which still need to be fixed, but most code can be run on top of the Sista VM as it would be run on the production VM. We have started to integrate the dependencies of Sista in Pharo, such as the new implementation of closures, the new bytecode set or read-only objects. We are now looking forward to integrate Scorch in Pharo.

9.2 New optimisations

Another direction for future work is the implementation of new optimisations in Scorch or Cogit.

9.2.1 State-of-the-art optimisations

To compare Sista against other optimising JITs efficiently, the next thing to do is to implement all the state-of-the-art compiler optimisation in Scorch and Cogit. Scorch lacks multiple common optimisation passes, including important ones for performance such as floating-pointer related optimisations or advanced escape analysis. Cogit features a naive register allocator to set registers in its dynamic templates, but we have plan to build a more advanced one and evaluate the performance difference.

9.2.2 New optimisations

Aside from existing optimisations, one could implement new optimisations that are not present in other compilers.

One way to do so is to have new ideas on how to optimise code, to design and implement new algorithms. This is far from trivial as many experts have worked in compiler optimisations in other compilers, but, it is theoretically possible.

Alternatively, one could describe how traditional optimisations are implemented in the context of the split architecture present in Sista. For example, one could explain which optimisation should be implemented in Scorch, which one should be implemented in Cogit, which one should be implemented partly with both and which annotations are required in the optimised v-functions to communicate extra information from Scorch to Cogit.

Lastly, and most interestingly, one could work on Smalltalk-specific optimisations that are not possible or not relevant in other programming languages because they do not usually support the unconventional features present in Smalltalk.

Indeed, Smalltalk provides some unconventional operations that are not usually available in other object-oriented languages. These operations are problematic for the optimising JIT. The main operations we are talking about are become, described in Section and heavy stack manipulation APIs on reified stack frames detailed in Section 3.1.

In each case, the feature has implications in the context of an optimising JIT as at any interrupt point, there any temporary variable of the optimised stack frame, or worst, any internal state (sender frame, program counter, etc.) could be edited to any object in the heap. This would invalidate all assumptions taken at compilation time by Scorch.

Fortunately, all these operations are uncommon in a normal Smalltalk program at runtime. They are usually used for implementing the debugging functionalities of Pharo. Currently, profiling production applications does not show we could earn noticeable performance if we would optimise such cases. The current solution is therefore to always deoptimise the stack frames involved when such unconventional operations happen. In the case of become, if a temporary variable in a stack frame executing an optimised method is edited, the frame is deoptimised. In the case of the stack manipulation, if the reification of the stack is mutated from the language, we deoptimise the corresponding mutated stack frames.

However, in specific libraries or workflow, such operations may be common enough to have some impact on performance. Especially, continuations and exceptions are built in Pharo in the language on top of the stack manipulation features, and a few libraries use them extensively. It could be possible to have Scorch aware of these features and to handle them specifically. Scorch would for example mark some temporary variables as being accessed frequently from the outside of the function: such temporaries would not be optimised and the deoptimisation metadata would include information on how to access those temporaries of inlined functions directly in the optimised frame. Optimising such features would allow to have efficient continuations and exceptions, in a similar way to the optimisation of exceptions described in [Ogasawara 2001].

9.3 Application of Sista for quick start-ups

One point that is a bit unclear is how to use the Sista for quick start-up in a real-world application. It is possible to persist optimised functions across start-ups. However, how are the optimised functions generated in the first place? Are they generated from warm-up runs using a test suite or are they generated the first day

the application is running?

A good example on how to use persisted optimisations across start-up is the success story of Azul [Systems 2002]: a trading bank claims that they are able to use the optimised functions generated the day N-1 to improve the start-up performance of the day N. Another good example is the user-base of the cloneable Java VM [Kawachiya 2007].

It seems that depending on the use-case where start-up performance matters (distributed application with short-lived slaves, such as the ones on Amazon lambda, Android application or Web pages), different frameworks and solutions require to be set up to improve start-up performance using persisted optimisations.

It would be very valuable to focus on one of those use-cases and build a solid framework showing how to use the persistence of optimisations to reduce warm-up time and evaluate what is the cost for the application programmer. Does the programmer have to do something specific to persist the optimisations as part of the deployment (warm-up runs, etc.) or is it done automatically as part of a framework?

In the case of distributed application on Amazon lambda, slaves may live down to a couple seconds in case of a high-variant demand, so that there are very few slaves rent when the application is unused and a lot of slaves when the application is heavily used. Is it possible to build a framework that automatically learn from the life of previous slaves what optimised functions are worth keeping, so when a new short-lived slaved is instantiated, it can reach peak performance very quickly?

In the case of web pages, is it possible to build a global cache so all frequently used web pages would have pre-optimised code available from the runs of the previous users? How would such a design work with modern security requirements?

All these applications of the persistence of optimisations across start-ups of Sista are very interesting and could be analysed in future work.

9.4 Energy consumption evaluation

Another interesting aspect of Sista is the energy saved at start-up by re-using persisted optimised functions instead of wasting cpu cycles re-generating them.

One of the most relevant use-case is Android application. With the Dalvik VM or the Android Runtime, Google's team on the VM for Android application have switched their VM design from JIT compilation to AOT compilation then back to JIT compilation. The main problem is that JIT compilation yields better peak performance, but requires warm-up time for each Android application start-up which wastes many cpu cycles, which corresponds in practice to an important part of the smart phone battery.

Sista would be relevant in this context as the application could be shipped un-optimised, but each time the user would use the application, the optimised func-

tions generated from previous runs would be persisted so further uses won't waste battery. This way, theoretically, the application could have very good peak performance due to the JIT while not wasting too many battery at each Android application start-up.

In addition, power drain is becoming an important factor not just in mobile computing but also computers embedded in things (Internet of Things) and even on servers. Excessive power leads to increased cooling. In flash memory, excessive writes can not only shorten its life but also lead to increased power (writes draw more power than reads).

To work in this direction, one would need to evaluate the energy consumed by the VM to reach peak-performance. We did not work in this direction because we did not have expertise in energy consumption measurement, but this is definitely a relevant future work.

Conclusion

This chapter discussed the future works that may be done based on this thesis. The next chapter summarises the contents of the thesis and concludes.

CHAPTER 10

Conclusion

Contents

In this chapter we summarise the dissertation, then list the contributions and lastly discuss the impact of the thesis.

10.1 Summary

In the thesis we focused on three problems. First, we investigated how to build an optimising JIT architecture where the optimising JIT is running in the same runtime than the running application on top of an existing VM composed of an interpreter and a baseline JIT. The result is Sista, our optimising JIT architecture. Second, we studied metacircular problems related to an optimising JIT that can optimise its own code. As a result, Scorch is able to optimise its own code with metacircular problems under specific circumstances. Third, we analysed the persistence of the runtime state, including runtime optimisations, across multiple VM start-ups in the context of the warm-up time problem. As a result, Sista is able to avoid most of the warm-up time by persisting optimised v-functions across VM start-ups.

Chapter 2 defined the terminology used in the thesis. The chapter then described the two most common architectures for optimising JITs, the function-based and the meta-tracing architecture, with examples of VMs using one architecture or the other. Then, work related to Sista in the context of metacircular VMs and the persistence of the runtime state across start-ups were listed.

Chapter 3 detailed the existing Pharo runtime. The focus was on the features not present in most other VMs or relevant in the context of the dissertation. Sista is built on top of the existing runtime described in this chapter.

Chapter 4 explained Sista, including the inner details of the architecture. Especially, the split between Scorch and Cogit in the optimising JIT architecture was

described. The chapter compared Sista against existing architectures for optimising JITs.

Chapter 5 discussed the evolutions required to the existing Pharo runtime to implement Sista on top of the existing runtime. This included all the core aspects of the system which needed to be changed to support Sista, such as the implementation of closures or the ability to mark an object as read-only.

Chapter 6 was articulated around a problem happening in metacircular optimising JIT such as Scorch: under specific circumstances, Scorch ends up in an infinite recursion where it calls itself repeatedly. The problem can happen both when Scorch attempts to optimise one of its own function or when it attempts to deoptimise one of its own frame. The problem is solved by disabling temporarily the optimiser when relevant and by making the deoptimisater code completely independent from the rest of Pharo.

Chapter 7 detailed how the runtime state is persisted across multiple start-ups in the context of Sista, including optimised v-functions, allowing one to decrease warm-up time.

Chapter 8 showed benchmarks run on the existing Pharo runtime and the Sista runtime. The Sista VM is up to 5x faster at peak performance than the current production runtime and peak performance is reached almost immediately after start-up if optimised v-functions are persisted as part of the snapshot.

Chapter 9 discussed future work that are worth investigating based on the contents of this thesis.

10.2 Contributions

The main contributions of this thesis are:

- The implementation of Sista, which yields 1.5x to 5x speed-up in execution time compared to the existing Pharo runtime and allow one to reach peak performance very quickly if optimised v-functions are persisted from previous runs of the VM.
- An alternative bytecode set solving multiple existing encoding limitations.
- A language extension: each object can now be marked as read-only.
- An alternative implementation of closures, both allowing simplifications in existing code and enabling new optimisation possibilities.

10.3 Impact of the thesis

Multiple contributions of the thesis (the alternative bytecode set, the read-only objects and the alternative implementation of closures) are integrated in Pharo and got some attention from the user-base. The main impact is Sista, which is currently being integrated in Pharo and will hopefully enable new research and industrial work.

Bibliography

[Alpern 1999] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd and Mark Mergen. *Implementing Jalapeño in Java*. In Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '99, 1999. 2, 5, 23, 83, 86

[Annamalai 2013] Siva Annamalai. *Snapshots in Dart*, 2013. https://www.dartlang.org/articles/snapshots/. 4, 27, 91, 92

[Araya 2013] Vanessa Peña Araya, Alexandre Bergel, Damien Cassou, Stéphane Ducasse and Jannik Laval. *Agile Visualization with Roassal*. In Deep Into Pharo. Square Bracket Associates, 2013. 104

[Arnold 2000] Matthew Arnold, Stephen Fink, David Grove, Michael Hind and Peter F. Sweeney. *Adaptive Optimization in the Jalapeño JVM: The Controller's Analytical Model*. In Workshop on Feedback-Directed and Dynamic Optimization, FDDO-3, 2000. 20, 24, 37, 86

[Arnold 2002] Matthew Arnold, Michael Hind and Barbara G. Ryder. *Online Feedback-directed Optimization of Java*. In Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '02, 2002. 44, 60

[Arnold 2005] Matthew Arnold, Adam Welc and V. T. Rajan. *Improving Virtual Machine Performance Using a Cross-run Profile Repository*. In Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, 2005. 27, 94

[Bala 2000] Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia. *Dynamo: A Transparent Dynamic Optimization System*. In Programming Language Design and Implementation, PLDI '00, 2000. 22

[Béra 2013] Clément Béra and Markus Denker. *Towards a flexible Pharo Compiler*. In International Workshop on Smalltalk Technologies 2013, IWST '13, 2013. 44

[Béra 2014] Clément Béra and Eliot Miranda. *A bytecode set for adaptive optimizations*. In International Workshop on Smalltalk Technologies 2014, IWST '14, 2014. 61, 66

[Béra 2016a] Clément Béra. *A low Overhead Per Object Write Barrier for the Cog VM*. In International Workshop on Smalltalk Technologies IWST'16, 2016. 67

[Béra 2016b] Clément Béra, Eliot Miranda, Marcus Denker and Stéphane Ducasse. *Practical Validation of Bytecode to Bytecode JIT Compiler Dynamic Deoptimization*. Journal of Object Technology, 2016. 103

[Black 2007] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Squeak by example. Square Bracket Associates, 2007. 4

[Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Pharo by example. Square Bracket Associates, Kehrsatz, Switzerland, 2009. 4, 29

[Bodík 2000] Rastislav Bodík, Rajiv Gupta and Vivek Sarkar. *ABCD: Eliminating Array Bounds Checks on Demand*. In Programming Language Design and Implementation, PLDI '00, 2000. 49, 64

[Bolz 2009] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski and Armin Rigo. *Tracing the meta-level: PyPy's tracing JIT compiler*. In Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS'09, 2009. 22

[Bornstein 2008] Dan Bornstein. *Dalvik Virtual Machine internal talk, Google I/O*, 2008. 90

[Bunge 2009] Philipp Bunge. Scripting Browsers with Glamour. Master's thesis, University of Bern, 2009. 104

[Chiba 1996] Shigeru Chiba, Gregor Kiczales and John Lamping. *Avoiding Confusion in Metacircularity: The Meta-Helix*. In Kokichi Futatsugi and Satoshi Matsuoka, editeurs, ISOTAS'96, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996. 71

[Denker 2008] Marcus Denker, Mathieu Suen and Stéphane Ducasse. *The Meta in Meta-object Architectures*. In TOOLS EUROPE, volume 11 of *LNBIP*, pages 218–237. Springer-Verlag, 2008. 71

[Deutsch 1984] L. Peter Deutsch and Allan M. Schiffman. *Efficient Implementation of the Smalltalk-80 system*. In Principles of Programming Languages, POPL '84, 1984. 14, 17, 18, 21, 31, 32, 39, 63

[Duboscq 2013] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer and Hanspeter Mössenböck. *Graal IR: An Extensible Declarative Intermediate Representation*. In Asia-Pacific Programming Languages and Compilers Workshop, 2013. 5, 15, 24, 56, 83

[Ducasse 2000] Stéphane Ducasse, Michele Lanza and Sander Tichelaar. *Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*. In CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools), 2000. 104

[Ducasse 2005] Stéphane Ducasse, Tudor Gîrba, Michele Lanza and Serge Demeyer. *Moose: a Collaborative and Extensible Reengineering Environment*. In Tools for Software Maintenance and Reengineering, RCOST / Software Technology Series. Franco Angeli, Milano, 2005. 104

[Felgentreff 2016a] Tim Felgentreff. *Squeak VM speed center*, 2016. http://speed.squeak.org. 97

[Felgentreff 2016b] Tim Felgentreff, Tobias Pape, Patrick Rein and Robert Hirschfeld. *How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain*. In International Workshop on Smalltalk Technologies, IWST'16, pages 21:1–21:10, New York, NY, USA, 2016. ACM. 55

[Fernandes 2007] Hilaire Fernandes and Serge Stinckwich. *Morphic, les interfaces utilisateurs selon Squeak*, 2007. 104

[Fink 2003] Stephen J. Fink and Feng Qian. *Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement*. In International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '03, Washington, DC, USA, 2003. IEEE Computer Society. 50

[Gal 2006] Andreas Gal, Christian W. Probst and Michael Franz. *HotpathVM: An Effective JIT Compiler for Resource-constrained Devices*. In Virtual Execution Environments, VEE '06, 2006. 22

[Gal 2009] 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 Programming Language Design and Implementation, PLDI '09, 2009. 22

[Geoffray 2015] Nicolas Geoffray. *From Dalvik to ART: JIT! -> AOT! -> JIT! internal talk, Google compiler phd summit*, 2015. 90

[Gîrba 2010] Tudor Gîrba. *The Moose Book*, 2010. 104

[Goldberg 1983] Adele Goldberg and David Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. 2, 4, 29, 32, 91

[Google 2008] Google. *V8 source code repository*, 2008. https://github.com/v8/v8. 1, 15, 18, 20, 51, 57, 64, 83, 103

[Gouy 2004] Isaac Gouy and Fulgham Brent. *The Computer Language Benchmarks Game*, 2004. http://benchmarksgame.alioth.debian.org/. 97

[Grimmer 2013] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz and Hanspeter Mössenböck. *An Efficient Native Function Interface for Java*. In Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '13, 2013. 56, 84

[Group 2015] WebAssembly Community Group. *WebAseembly official website*, 2015. http://webassembly.org/. 20, 56, 57, 106, 108

[Hölzle 1991] Urs Hölzle, Craig Chambers and David Ungar. *Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches*. In European Conference on Object-Oriented Programming, ECOOP '91, London, UK, UK, 1991. 14, 17, 31, 39, 50, 63

[Hölzle 1992] Urs Hölzle, Craig Chambers and David Ungar. *Debugging Optimized Code with Dynamic Deoptimization*. In Programming Language Design and Implementation, PLDI '92, 1992. 14, 50, 51

[Hölzle 1994a] Urs Hölzle. *Adaptive optimization for Self: reconciling high performance with exploratory programming*. Ph.D. thesis, Stanford, 1994. 10, 11, 13, 15, 20, 21, 23

[Hölzle 1994b] Urs Hölzle and David Ungar. *Optimizing Dynamically-dispatched Calls with Run-time Type Feedback*. In Programming Language Design and Implementation, PLDI '94, pages 326–336, New York, NY, USA, 1994. 14

[Ingalls 1997] 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 Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '97, 1997. 2, 5, 29, 65

[Kawachiya 2007] Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu and Toshio Nakatani. *Cloneable JVM: A New Approach to Start Isolated Java Applications Faster*. In International Conference on Virtual Execution Environments, VEE '07, 2007. 28, 93, 112

[Krintz 2001] Chandra Krintz and Brad Calder. *Using Annotations to Reduce Dynamic Optimization Time*. In Programming Language Design and Implementation, PLDI '01, 2001. 95

[Milojković 2016] Nevena Milojković, Clément Béra, Mohammad Ghafari and Oscar Nierstrasz. *Inferring Types by Mining Class Usage Frequency from Inline Caches*. In International Workshop on Smalltalk Technologies, IWST'16, 2016. 104

[Miranda 2002] Eliot Miranda, with contributions from Paolo Bonzini, Steve Dahl, David Griswold, Urs Hölzle, Ian Piumarta and David Simmons. *A Sketch for an Adaptive Optimizer for Smalltalk written in Smalltalk*, 2002. Technical report: https://hal.inria.fr/hal-01525754/document. 2, 20

[Miranda 2008] Eliot Miranda. *Cog Blog: Speeding Up Terf, Squeak, Pharo and Croquet with a fast open-source Smalltalk VM*, 2008. http://www.mirandabanda.org/cogblog/. 4, 29

[Miranda 2015] Eliot Miranda and Clément Béra. *A Partial Read Barrier for Efficient Support of Live Object-oriented Programming*. In International Symposium on Memory Management, ISMM '15, 2015. 36, 66

[Ogasawara 2001] Takeshi Ogasawara, Hideaki Komatsu and Toshio Nakatani. *A Study of Exception Handling and Its Dynamic Optimization in Java*. In Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '01. ACM, 2001. 111

[Oracle 2007] Oracle. *JRockit*, 2007. https://docs.oracle.com/cd/E13188_01/jrockit/docs142/userguide/codecach.html. 27, 94

[Oracle 2013] Oracle. *OpenJDK: Graal project*, 2013. http://openjdk.java.net/projects/graal/. 15, 24, 56, 83

[Paleczny 2001] Michael Paleczny, Christopher Vick and Cliff Click. *The Java hotspotTM Server Compiler*. In Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1, JVM'01. USENIX Association, 2001. 1, 15, 83, 84

[Pimás 2014] Javier Pimás, Javier Burroni and Gerardo Richarte. *Design and implementation of Bee Smalltalk Runtime*. In International Workshop on Smalltalk Technologies, IWST'14, 2014. 24

[Pluquet 2009] Frédéric Pluquet, Antoine Marot and Roel Wuyts. *Fast type reconstruction for dynamically typed programming languages*. In DLS'09: Symposium on Dynamic Languages, pages 69–78, New York, NY, USA, 2009. ACM. 104

[Reddi 2007] Vijay Janapa Reddi, Dan Connors, Robert Cohn and Michael D. Smith. *Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications*. In Proceedings of the International Symposium on Code Generation and Optimization, CGO '07, 2007. 27, 94

[Rigo 2006] Armin Rigo and Samuele Pedroni. *PyPy's Approach to Virtual Machine Construction*. In Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM. 2, 22, 23, 30, 55, 83

[Salgado 2016] Ronie Salgado and Stéphane Ducasse. *Lowcode: Extending Pharo with C Types to Improve Performance*. In International Workshop on Smalltalk Technologies, IWST'16, 2016. 56, 108

[Stadler 2012] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck and Thomas Würthinger. *Compilation Queuing and Graph Caching for Dynamic Compilers*. In Virtual Machines and Intermediate Languages, VMIL '12, 2012. 20

[Sun Microsystems 2006] Inc. Sun Microsystems. *Strongtalk official website*, 2006. http://www.strongtalk.org/. 11, 15, 23, 27, 94

[Systems 2002] Azul Systems. *Azul official website*, 2002. https://www.azul.com/. 27, 94, 112

[Ungar 2005] David Ungar, Adam Spitz and Alex Ausch. *Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment*. In Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, pages 11–20, New York, NY, USA, 2005. 2, 5, 24, 83

[Verwaest 2010] Toon Verwaest, Camillo Bruni, David Gurtner, Adrian Lienhard and Oscar Niestrasz. *Pinocchio: Bringing Reflection to Life with First-class Interpreters*. In Object Oriented Programming Systems Languages and Applications, OOPSLA '10, pages 774–789, New York, NY, USA, 2010. ACM. 24

[Webkit 2015] Webkit. *Introducing the Webkit FTL JIT*, 2015. https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/. 18, 19, 23, 26, 57, 83, 93

[Wimmer 2013] 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., pages 30:1–30:24, 2013. 2, 5, 24, 83

[Wolczko 1987] Mario Wolczko. *Hardware support for Objects: The MUSHROOM Project*, 1987. http://www.wolczko.com/mushroom/. 24

[Würthinger 2013] 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 International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward2013, 2013. 24