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 persistance

THÈSE

présentée et soutenue publiquement le

pour l'obtention du

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

par

Clément Béra

Composition du jury

Président :	???	
Rapporteur :		
Examinateur :		
Directeur de thèse :	Stéphane Ducasse	(Directeur de recherche – INRIA Lille Nord-Europe)
Co-Encadreur de thèse :	Marcus Denker	(Chargé de recherche – INRIA Lille Nord-Europe)

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 X, Y and Z for kindly reviewing my thesis and providing me valuable feedback.

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

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

For remarks on earlier versions of this thesis I thank X and Y.

Abstract

Résumé

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 the platform-independency: 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 the whole thesis, the term VM is used to dissertate about virtual machines for high-level programming languages, by opposition to operating system VMs which are not discussed at all.

Virtual machines for high-level programming languages. Most high-level languages pursue a strict separation between language-side and VM-side. VMs for instance provide automatic memory management or use platform independent instructions such as bytecodes. These properties allow a programming language to develop independently from the underlying hardware.

High performance VMs, such as Java HotSpot or current Javascript VMs achieve high performance through just-in-time compilation techniques: once the VM has detected that a portion of code is frequently used (called *hot spot*), it recompiles it on-the-fly with speculative optimizations based on previous runs of the code. If

usage patterns change and the code is not executed as previously speculated anymore, the VM dynamically deoptimizes 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 speculate correctly about frequently used patterns. This warm-up time can be problematic for different use-cases.

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 get written in higher-level languages as an attempt to control complexity. Such VMs got 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].

Pharo programming language and community. In this thesis the focus is on a specific high-level object-oriented programming language, the Smalltalk dialect named Pharo [Black 2009], a fork of another Smalltalk dialect named Squeak [Black 2007] made by the original Smalltalk-80 implementors. 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 part of high performance VMs. As the Pharo community has a limited amount of ressources 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. To attract open-source contributors from the community, the optimising JIT has to be written in Smalltalk.

Existing design. A first optimising JIT design emerged in the early 2000s for another Smalltalk VM, a precursor of the current Pharo VM. 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 going to be reasonnable and to attract contributors from the community.

As most of the community has high skills in Smalltalk but little skills in low-level programming, one of the idea 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. 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. This way, members of the Pharo community can contribute to a project in Smalltalk doing Smalltalk-specific optimisations, improving performance while staying away from low-level details and machine-specific optimisations.

After discussing with other VM implementors, it seems that language-specific optimisations (in our case Smalltalk-specific optimisations) are more important than machine-specific optimisations for performance. As the community has a smaller number of low-level contributors, another idea is then to reuse the existing baseline JIT, already present in the VM, 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 was conceived as an extended bytecode set (the existing bytecode set with the

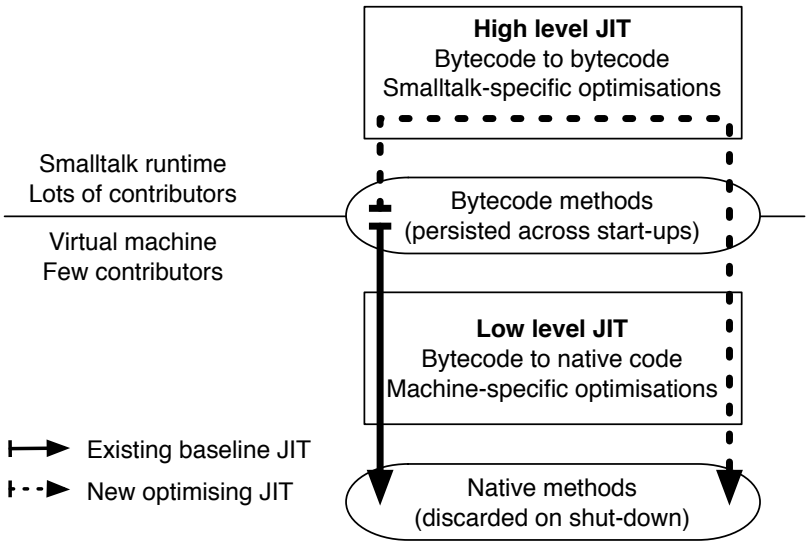


Figure 1.1: JIT compilation model design

addition of new operations used only by optimised code). This way, the existing baseline JIT already supporting the existing bytecode set would "just" need to be slightly extended to support the new operations.

Lastly, the high-level part of the JIT is generating platform-independent optimised bytecode methods. Bytecode methods can already be persisted across multiple start-ups. The last idea is 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.

Some aspects of the design were considered, analyzed and discussed very carefully by part of the Smalltalk community, making the design attractive and interesting. However, the overall design was incomplete so it was unclear how multiple parts of the system would work.

The thesis started from this proposal, in an attempt to complete the design and propose an implementation. Multiple aspects of the design are different from existing VMs. The advantages and issues of these differences are discussed in the thesis.

1.2 Problem

During the thesis, the main research direction was the following: *How to build an optimising JIT for Pharo, written in Pharo itself, running in the same runtime than the optimised application on top of the existing runtime environment ?*

By design, the optimising compiler has to run 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. In this case, as Java is multithreaded, it runs in the same runtime than the running application but in different native threads. In Graal though, 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.

As the attempt to implement the proposal was started to execute code, 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. The Smalltalk VM normally 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. In most existing VMs, the optimised code is not persisted across multiple start-ups, making difficult the persistance of green threads unless all stack frames present in their execution stack are deoptimised. With the bytecode to bytecode optimisation design, the persistance of running green threads, including the persistance of optimised code they refer to, may be possible across multiple start-ups.

Research subproblems. The thesis focuses on two aspects in the context of the main problem:

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

1.3 Contributions

The main contributions of this thesis are, in the context of the Pharo programming language:

- An optimising JIT running on top of the existing production virtual-machine, showing improved performance in execution time.

- 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.

1.4 Overview

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

1.5 Terminology

Functions. In the thesis we use the term *function* to discuss 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 function*, or n-function, the native code version of a function executed by a specific processor.

Frames. We discuss VMs using an 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.

1.6 Outline

- Chapter 2 presents existing production and research virtual machines relevant in the context of the thesis.
- Chapter 3 detail the existing Pharo runtime as Sista is built on top of it.
- Chapter 4 details the Sista architecture 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 the two subproblems of the thesis.
- Chapter 8 evaluates the Sista architecture by comparing the performance of the runtime in multiple contexts.

CHAPTER 2

State of the art

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 than the optimised application on top of the existing VM. The following few paragraphs introduce briefly the need of an optimising JIT for performance and how an optimising JIT improve performance of the running programs. Concrete examples and references are present in the context of the two most popular optimising JIT architecture in section 2.1.2 and 2.2.2.

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. The same code snippet can therefore be executed differently depending on how frequently it is run and how

many times it has been executed since the last start-up. 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 is faster. 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 executing instructions present in v-functions. The interpreter 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 start-up, the optimising JIT kicks in and generates optimised native instructions using runtime information collected during the interpreter runs. The run thousand and one 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.

Run Number	Tier	Compilation time	Execution time
1 to 1000	1 interpreter	0 ms	.5 ms / run
1001	2 optimising JIT	5 ms	.1 ms / run
1002 +		0 ms	

Figure 2.1: Execution of a frequently used code snippet

Optimising JIT architectures. Two main architectures are widely used to design an optimising JIT. The first optimising JIT architecture [Hölzle 1994], historically the first one invented, attempts to boost performance by optimising frequently used functions. The native code generated by such optimising JITs are optimised n-functions. We call this architecture the *Function based architecture* and we describe it in section 2.1. The second architecture focuses on the optimisation of linear sequences of frequently used instructions. We call this architecture the *Meta-tracing architecture*. Typically, meta-tracing JITs optimise the common execution

path of one iteration of each frequently used loop. This second architecture is detailled in section 2.2. As the Sista architecture is more similar to the function based architecture, section 2.1 is more detailled 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 2.3.

- *Runtime state persistance:* Most modern VMs always start-up an application with only non optimised 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 detailled in section 2.4.

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 specifc aspect of the VM.

Smalltalk VMs. As Pharo is a Smalltalk dialect, it may be interesting 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 1994] 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 as far as we know these VMs are not actively maintained nor used in production.

2.1 Function based architecture

The first optimising JIT architecture invented [Hölzle 1994] 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 2.1.2. The last sections discuss specific aspects of the architecture.

2.1.1 Architecture overview

In most VMs following this architecture, three tiers are present. The following three paragraphs detail each tier, including how virtual calls is 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 [1] but the execution of the v-function by the interpreter is not very fast. Virtual calls are usually implemented using a global look-up cache to avoid computing the function to activate at each call. The interpreter tier does not necessarily collect runtime information.

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 wasted 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 will be instrospected to collect runtime information if the function is executed enough times to be optimised by the next tier. The goal of the baseline JIT code generation is therefore to provide reliable runtime information with limited performance overhead over generating 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 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-optimise it differently. 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.

[1] 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).

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.

Terminology. We note that in this architecture, which is one of the most common architecture for high-performance VMs today [Google 2008, Webkit 2015](CITE HOTSPOT), two JIT tiers are present but are very different from each other. To disambiguate the two, in this thesis, we use the term *baseline JIT* to discuss the JIT tier translating quickly v-functions to n-functions with a limited number of optimisations. We call the other tier the *optimising JIT*. This tier translates multiple v-functions to a highly optimised n-function with speculative optimisations, based on the runtime information collected from runs in previours tiers.

2.1.2 Existing virtual machines

The first VM featuring this function based architecture was the Self VM [Hölzle 1994]. The Self VM had only two tiers, the baseline JIT and the optimising JIT. The Self programming language has never really become popular so the VM is not really used anymore.

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 (CITE) 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 (CITE). 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.

2.1.3 Just-in-time compiler tiers

Most VMs featuring this function based architecture 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 gen-

erate optimised code. Hence, a VM with the 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. Most VMs features 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.

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 nto 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.

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 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.

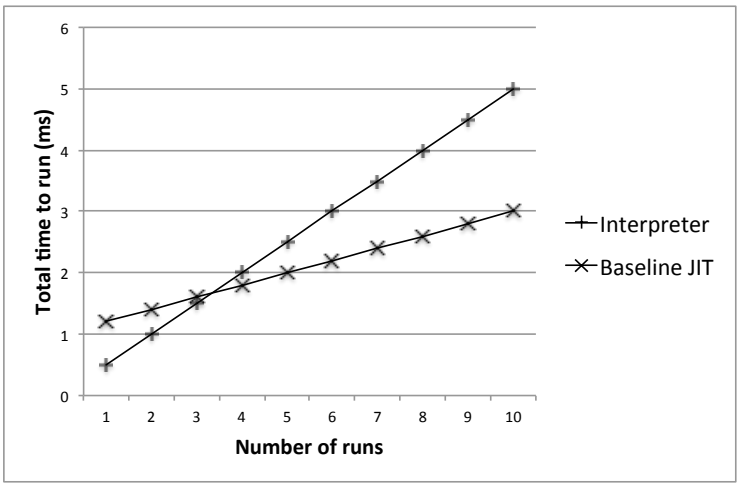


Figure 2.3: Time to execute a v-function in non-optimising tiers

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.

Independant compiler tiers. In most VMs, the baseline JIT and the optimising JIT are completely independant 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-function it generates needs 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 can know 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 compilation time reasonnable, but potentially much higher than the baseline JIT. The n-functions generated by the optimising JIT are in most VMs not introspected, 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 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 instruction, where machine-specific optimisations are performed. Lastly it generates native code.

Sharing code between compiler tiers. Because of the fondamental 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 of 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.

2.1.4 Concurrent compilation.

The first optimising JIT, designed and implemented in Self [Hölzle 1994], was done in 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 environment has 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 than 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. However, 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 Meta-tracing architecture

The main alternative to the function based architecture is the meta-tracing architecture. Meta-tracing JITs do not optimise entire functions but instead focus on optimising linear sequences of instructions. As most meta-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 2.2.2.

2.2.1 Architecture overview

VMs with meta-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 loop, 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 conditionnal branches or virtual calls. When generating the 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.

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

2.3 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.

2.3.1 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 1994, 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.

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 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 meta-circular 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 is even an attempt to write a Smalltalk VM in Smalltalk [Pimás 2014], called Bee Smalltalk. Unfortunately, as of today, Bee Smalltalk is not open-source and it is not clear if an optimising JIT is present or not.

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.

2.3.2 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 optimise 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, are 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 in the compilation queue and generate an optimised function for function in the queue. Hotspot provides APIs to extract runtime information from each non optimised 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 may install 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.

2.4 Runtime state persistance

In our work, we will attempt to persist the runtime state across multiple start-ups, including the optimised code but also the running green threads using optimised code. It does not seem that the persistance of running green thread with optimised code has been done before. 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 have the same requirement so the running green threads are not persisted across start-ups. For this reason, we focus in this section on the persistance 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 non optimised runs of a code snippet are required before reaching peak performance. This warm-up time can cause significant problems in specific applications.

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-optimize 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 precised 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.

Most object-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 persistance across start-ups.

Many tier 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 optimizations, 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 optimizing compiler, and executed as machine code up to 66 executions.
3. Recompiled to more optimized machine code at 67th execution, with an optimizing compiler doing some but not all optimisations, up to 666 executions.
4. Recompiled to heavily optimized 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 metadata. To reach quickly 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 decision of the optimising compiler in a separate file. The optimizing compiler can then reuse this file to take the right inlining decision in subsequent start-ups.

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 optimization 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.

Preheating through snapshots in Dart 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 to persist type feedback and call site frequency.

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 optimizing compiler that don't require dynamic deoptimization. A shared library is generated, including all the instructions, and a snapshot that includes all the classes, functions, literal pools, call site caches, etc.

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 design decisions. They are detailled 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 additionnal 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 flavor 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 to annotate functions to direct Slang to C compilation, by duplicating or inlining specific functions. 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 happing from a specific runtime state.

The executable is generated in two steps as shown on figure 3.1. 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 kind of optimisations:

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 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 template

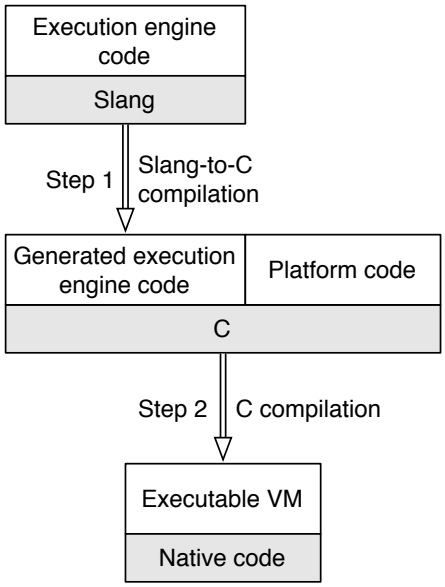


Figure 3.1: VM executable generation

for the instruction, use a linear scan algorithm to allocate registers that do not need to be fixed into specific concrete registers and generate 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, through specific parts, such as inline cache relinking, needs 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 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-function activation 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.

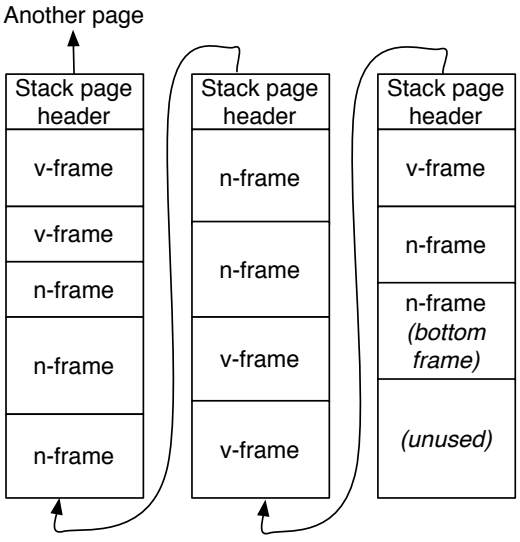


Figure 3.2: Stack representation

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 two latter, 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 the thesis, we will 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 additionnal concepts or side-effects. In practice, there are three differences:

1. A context is an object while a v-frame use 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 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. 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 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, such

operations can require a married context to "divorce" the frame, killing the frame in the process. Upon divorce, the stack page is split in two parts and one part if copied to another stack page. 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 linkedlist of stack pages and contexts.

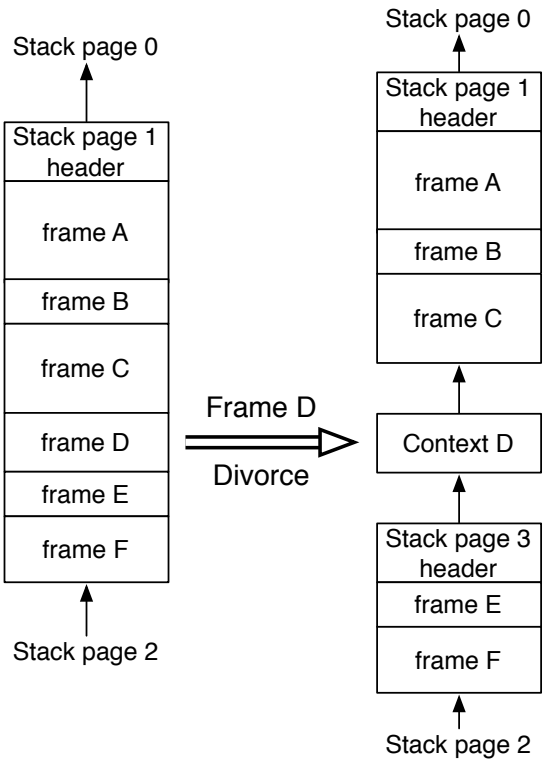


Figure 3.3: Stack frame divorce

Marriage 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-use the stack page for the execution of new code.

3.2 Language-VM interface

Pharo interfaces with its 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.

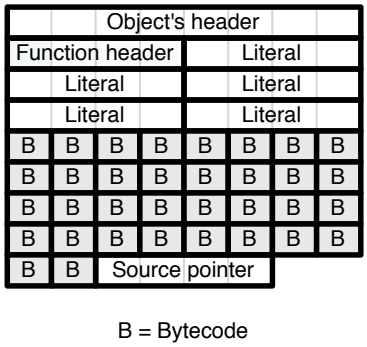


Figure 3.4: Virtual function representation

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 instruction 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 to implement and safer.

Closure's functions are installed inside the method that instantiate them as a literal. Changing the literal (hence the closure's function) is normally not done in the current runtime.

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 its set of exotic 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 [1] can be accessed in the Pharo runtime. 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. For example, the #doesNotUnderstand: selector is registered. 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.

3.3 Pharo programming language

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

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

Native threads 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 (CITE Adaptive Optimization in the Jalapeno JVM: The Controller's Analytical Model or maybe somethign with truffle graal).

Snapshots. In the context of Smalltalk, a snapshot [2] 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, the virtual machine can, at start-up, load all the objects from a snapshot and resume 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 cannot save n-frames because they are platform-independent. In the Pharo VM for example, a snapshot can be taken on a laptop using a x86 processor and restarted on a raspberry pie using a ARMv6 processor.

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

[2] 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 compiler for Pharo, written in Pharo itself, running in the same runtime than the optimised application on top of the existing runtime environment. The chapter explains the overall architecture designed and implemented, called the Sista architecture. 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 the Sista architecture 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 explain the overall design. As the architecture is different from classical architectures, the terminology is clarified in section 4.1.3. Lastly, section 4.1.4 briefly describe how functions are optimised and deoptimised.

4.1.1 Existing runtime

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 Smalltalk, 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. 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. 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 non optimised 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

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 to provide a new primitive method in Smalltalk, which answers for a given function both the values present in inline caches and the values of profiling counters.

Optimising JIT. The optimising JIT was designed in two different parts as shown in figure 4.1. The high-level part is a non optimised 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 non optimised 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 optimisa-

tions. Hence, any work performed on Scorch is done in Smalltalk 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 non optimised v-function instructions. Unsafe instructions are faster to execute but requires the compiler to guarantee specific invariants.

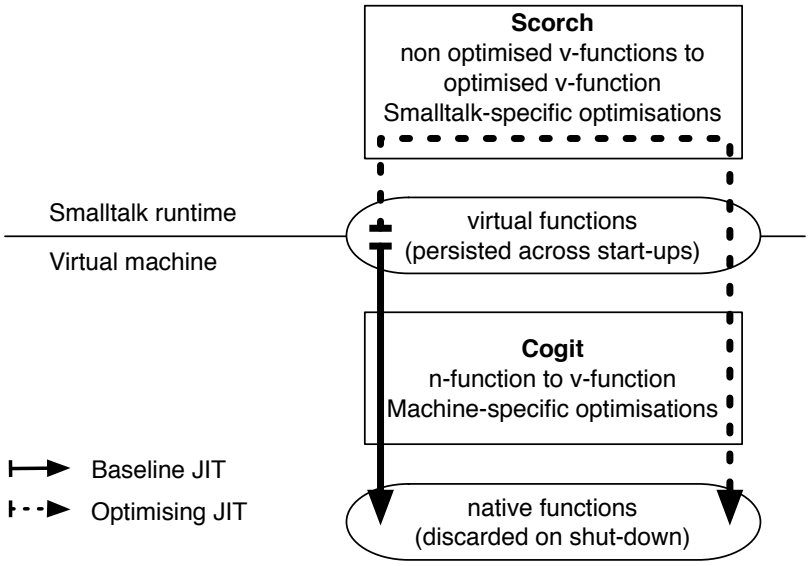


Figure 4.1: Scorch critical and background modes

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 the 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 Terminology

Sista is the named of the overall architecture detailled 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 later 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.

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 non optimised v-function when needed. Similarly, for frames, we say v-frame to discuss v-frames in general, and specify optimised v-frame and non optimised v-frame when discussing a v-frame respectively representing the execution state of an optimised v-function or a non optimised v-function. The same terminology is used with native (*n*-) than with virtual (*v*-).

Both runtime compilers can deoptimise stack frames. Cogit can deoptimise any n-frame to a single v-frame. Scorch can deoptimise an optimised v-frame to multiple non optimised v-frames.

4.1.4 Optimisation and deoptimisation

Cogit was extended to detect hot spot through profiling counters in non optimised 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 optimisation, Scorch may ask 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 is hybrid between the 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. If a guard fails, the execution stack needs to be deoptimised to resume execution with non optimised 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 non optimised v-frames. Scorch may discard the optimised v-function if guards are failing too often in it. The execution can then resume using non optimised 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 optimised v-function installed will be executed at the next call of the function.

4.2.1 Optimiser critical and background modes

Scorch's 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 an interruption of the application for an amount time long enough to be noticed by the user. To experiment with the Sista architecture, we worked with the development environment of Pharo (which is written in Pharo). In the case of a user-interface application like this one, it is *very* 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, force 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 in 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

[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.

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

when the developer is reading code, the application is in idle.

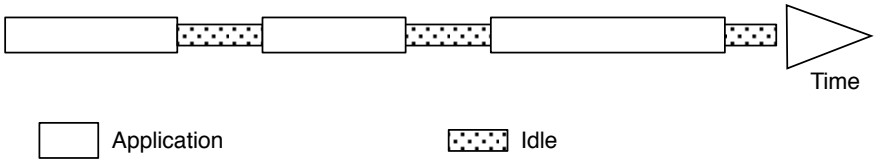


Figure 4.2: User interface application idle times

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 mode. 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 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's optimiser critical and background modes are represented on figure 4.3.

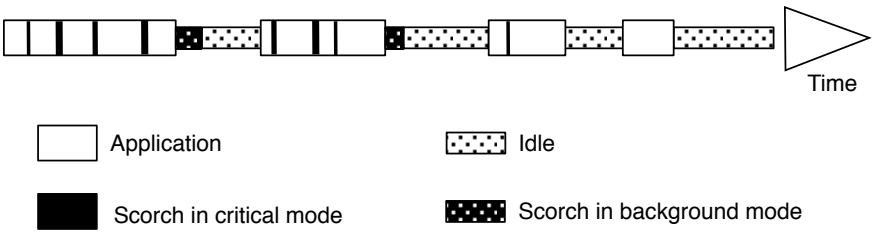


Figure 4.3: Scorch critical and background modes

Conclusion. The 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 limit 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 allow 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 to detect hot spots and provide information about basic block usage at the cost of a small overhead, detailled in the validation chapter of the thesis. Because of the overhead, Cogit was extended to support conditionnal compilation. Based on a specific bit in the v-function's header, Cogit compiles the v-function with or without profiling counters. Typically, non optimised 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 the branch. In several other VMs, the counters are added at the beginning of each function. The technique we used allowed 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 Smalltalk. Then, the routine performs a virtual call with a selector registered from Smalltalk, the reified stack frame as the receiver and the boolean the conditionnal jump was branching on as a parameter. The method activated by the call-back, in Smalltalk, calls the Scorch optimiser.

During optimisation, the bottom frames of the execution stack are used by the 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.

The profiling counter machine code is generated so that upon return to the n-frame with the profiling counter, the conditionnal branch is performed again on the value returned. This way, if the Smalltalk code resets the profiling counters and returns the boolean passed by argument to the call-back, the application execution is resumed by performing one more time the branch with the same boolean.

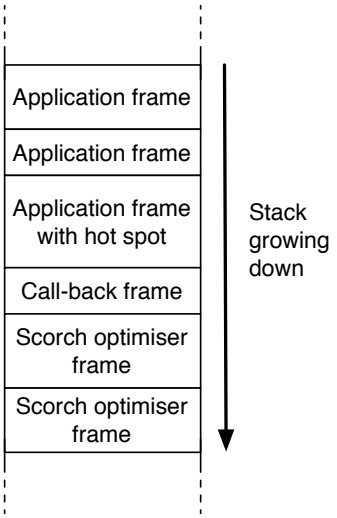


Figure 4.4: Stack state during critical mode optimisation

4.2.3 Scorch optimiser

The Scorch optimiser is activated by the call-back and has access to the reified stack. Scorch firstly analyse 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 Smalltalk. 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 4.5, 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 construct such as do:. In this case, do: is very similar to foreach in other languages and allows 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 represent 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 4.6.

```
ApplicationClass >> exampleArrayLoop
    array do: [ :element | FileStream stdout nextPutAll: element printString ].

Array >> do: aClosure
    1 to: self size do: [:index | aClosure value: (self at: index)].
```

Figure 4.5: Example code

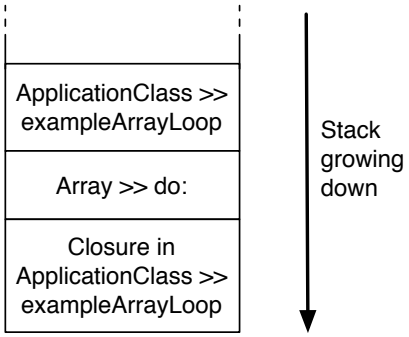


Figure 4.6: Example stack during closure execution

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 chose to optimiser the function where the hot spot is detected. It usually walks up a few frames to find the best function to optimise.

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's optimiser is implemented with traditional compiler optimisation techniques. It decompiles the v-function to optimise into a single static assignment 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 v-function decompiled. 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's optimiser performs Smalltalk specific optimisations, very similar to the object-oriented specific optimisations present in other optimising JITs. 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 phasis is finished, Scorch's optimiser performs standard optimisations such as array-bounds check elimination with the ABCD algorithm [Bodík 2000] or global value numbering. Scorch's 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).

The back-end is the only non conventional part of the optimiser. Scorch needs to generate an optimised v-function and not an optimised n-function. Most intermediate representation instructions map one to one to the extended bytecode set instructions. However, as the bytecode set is stack-based, the 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 non optimised v-functions.

Installation. If the optimisation has been done in time in critical mode or has simply been done in background mode, the optimised v-function can be 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.

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. The first version we had running was working exactly this way.

We then added a cheap heuristic to encourage the execution of optimised v-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, we added the heuristic that 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]. The optimised frame is on stack and is in most case a n-frame [3]. 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 non optimised 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 non optimised v-frames.

When discussing deoptimisation, we deal only with stack recovery (deoptimisation of the optimised frame to the non optimised v-frames). The non optimised v-function is always present and never discarded, so it can be directly set in the non optimised v-frames restored. There is no such thing in our design and implementation as reconstructing non optimised v-function from optimised v-function based on deoptimisation metadata. As far as we know, modern VMs such as V8 [Google 2008] always keep a non-optimised version of each function, so we

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

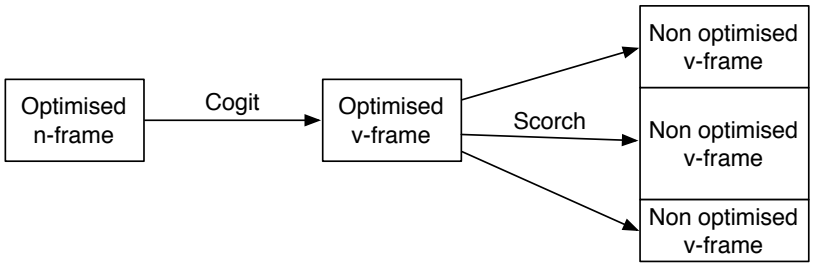


Figure 4.7: Stack frame deoptimisation in two steps

believe the memory footprint impact of keeping them is acceptable.

4.3.1 Deoptimisation management

Deoptimisation can happen in two main cases. On the one hand, Scorch's optimiser inserts guards(CITE URS OU Dubois) to ensure assumption 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 native code generated by Cogit calls a specific routine in Slang to perform the deoptimisation of the stack. The routine makes sure the frame with the tripping counter is reified so it can be introspected from Smalltalk. Then, the routine performs a virtual call with a selector registered from Smalltalk and the reified stack frame as the receiver. The Smalltalk code lastly calls Scorch deoptimiser's to restore the non optimised 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.

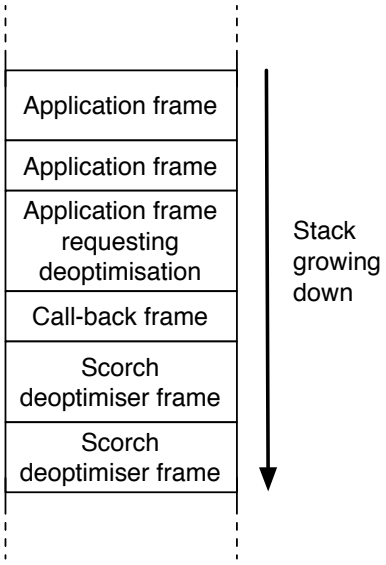


Figure 4.8: Stack state during guard deoptimisation

2. The instruction pointer is not on a guard instruction. Although, all instructions where deoptimisation can happen are annotated with deoptimisation metadata, so having the instruction pointer on a non guard instruction does not really change much.

4.3.2 Scorch deoptimiser

Cogit has reified the optimised n-frame to an optimised v-frame, hence, Scorch's deoptimiser access 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 have been postponed from runtime to deoptimisation time. As v-frames and closures are reified as objects in Smalltalk, part of those objects to rematerialize are non optimised 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 edition. Once all the non optimised 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 split 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.

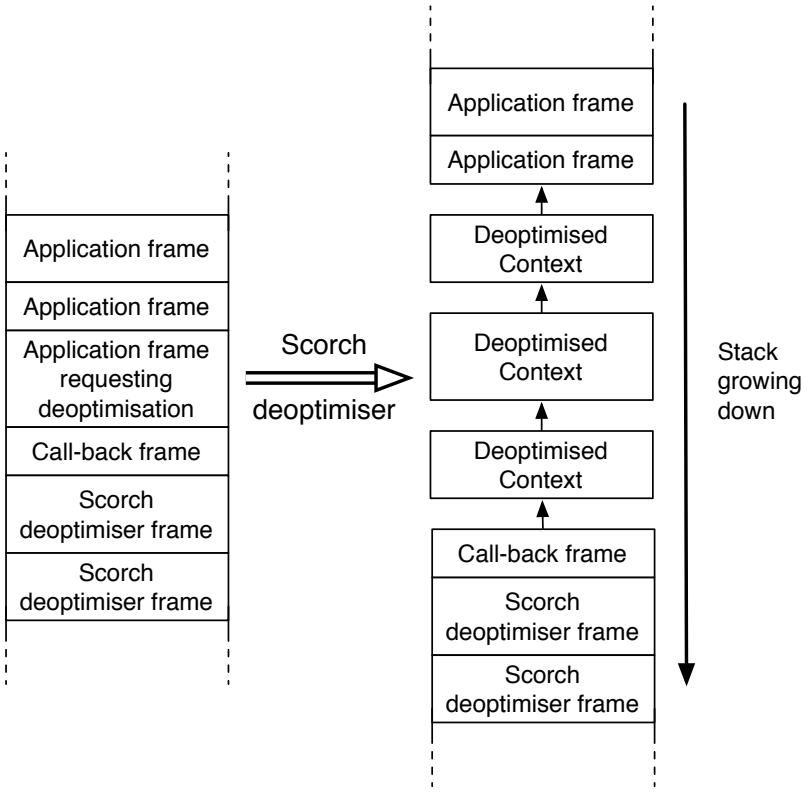


Figure 4.9: Stack edition

Discard functions. When new non optimised functions are installed (for example when a library is loaded), optimisation assumptions may be invalidated because some look-up results cannot be guaranteed any more. 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 mutates the discarded optimised v-function to hold only guard failures virtual instructions, so optimised v-frames would always trigger deoptimisation upon return. 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 should resume execution. There is one little detail, a return instruction in Smalltalk pushes the returned value on the caller stack. The deoptimisation call-back returns the top value of the bottom deoptimised frame to resume execution. The bottom deoptimised frame is guaranteed not to be empty as deoptimisation can be only in position in the code where there is necessarily something on stack.

4.4 Related work

This section compares our architecture against existing work. The first section compares the overall Sista architecture versus the meta-tracing and function based architecture. Section 4.4.2 compares our optimised v-function representation against other representation 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. The Sista architecture 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.

The Sista architecture was not designed as a meta-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 activates 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 implement an efficient meta-tracing JIT for Pharo.

4.4.2 Optimised virtual function representation

One important difference in the Sista architecture 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. (CITE ???) One of the first work in the late 80s to speed up object language virtual machines 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 require a small compilation time to translate the v-function to its threaded code representation. The threaded code represenation is platform-independent and can be considered as an optimised v-function. The main advantage of the threaded code interpreter is to provide speed-up while keeping the VM platform-independent. Indeed, JIT require 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, new implementation techniques for v-function interpreters using threaded jumps allowed to massively reduce the performance difference between threaded code interpreter and v-function interpreters.

(CITE ???) 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.

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.

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 add 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 would be a valuable option.

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 independant 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 VM where important parts of code are shared between multiple JIT tier. 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, we share the back-ends of our two compiler tiers. 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 supports WebAssembly [Group 2015], a 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. This are some similarities with our work as WebAssembly is a representation that could be used to represent optimised v-function.

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 the Sista 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. Other evolutions were not mandatory, the Sista architecture could have worked without these features, but each of them were important to improve the overall performance.

5.1 Required language evolutions

Five major evolutions were required to have the Sista architecture 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. The Scorch framework, 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 in-

creases its value by one. If the counter reaches a threshold, the VM trigger 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 in the validation chapter on a set of benchmarks). To avoid the overhead in optimised code, Cogit was extended to support conditionnal 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], profiling counters were added 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, with 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 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 this n-function requires two counters, an array with two 32 bits wide fields is allocated on heap. Each 32 bits counter field

[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.

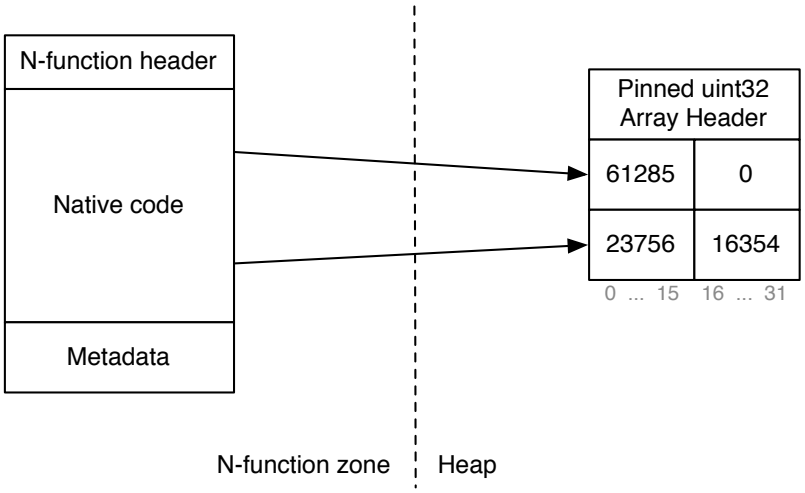


Figure 5.1: Non optimised n-function with two profiling counters

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

The Sista architecture required 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 IWST 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 bounds 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 check. 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, for example when doing a store in an object that has just been allocated, because it is guaranteed to be in the young object space.

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 to execute the optimised v-function. 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 the 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. Among registered objects are specific selectors. One example is 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.

We registered two new selectors. One is activated by the VM when a hot spot is detected in a non optimised 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 the Scorch optimiser or the 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-fucntion was not compiled, the primitive fails. The runtime information includes types and functions met at each inline cache and the profiling counters 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's API for introspection. In non optimised n-function, 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 conditionnal 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 structure using multiple different kind of objects (Each object's class internal representation would need to be specifically known by both the VM and the language). To keep things simple, the primitive answers an array of array, 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 the Sista architecture, the bulk of the work was the design and the implementation of the Scorch optimiser. The thesis is centered around the

Sista architecture in general and there is no big innovative features in Scorch.

Scorch's optimiser is implemented as a traditionnal optimising compiler. It translates the v-functions to optimise into a single static assignment intermediate representation, represented as a control flow graph. Conditionnal 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 optimisation 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 perform 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's 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 the multiple frames.

Scorch's 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 bottom frame restored.

Scorch's 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 object as read-only.
5. A new closure implementation to be able to optimiser closure more efficiently.

5.2.1 New memory manager

The first version of the Sista architecture was built on the existing VM with a minimum number of modifications. One of the main issue 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. The representation of object 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 non optimised 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 is now only two ways for an instance to access its class, the class is either immediate or compact. Compact classes indexes are stored in the instances in a 22-bit fields, allowing over four millions different concrete classes. As all references to classes are now through an indirection index, type-checks are not annotated by Cogit when generating the n-function as the garbage collector can ignore them.

In addition to allowing 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 conditonnal) 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 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 template for the instruction, use a linear scan algorithm to allocate registers that do not need to be fixed into specific concrete registers and generate the 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 non optimised code. Specifically, registers cannot be live across these three instructions:

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

2. *backjumps:* Backjumps are interrupt points.

3. *Conditionnal 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 conditionnal branches are removed because one branch has never been used and other are annotated as branching on a value which is guaranteed to be a boolean. Registers can therefore stay live across many more instructions and register allocation algorithm have 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 addition between two double constants the compiler can compute at compile-time the result. In Pharo, as literals are mutable, one of the double constant may be accessed through reflective APIs and mutated into another double, invalidating the compile-time result.

To solve the problem, we introduced a feature, called 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 revert 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].

Literals can now all be read-only objects by default. Any attempt to modify a read-only literal is caught by the runtime and Scorch may be 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.

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 implementation of closure was also complexifying 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. Cogit had significant complexity to handle n-function introspection of closures of simply to activate a closure from the closure's enclosing method v-function.

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.

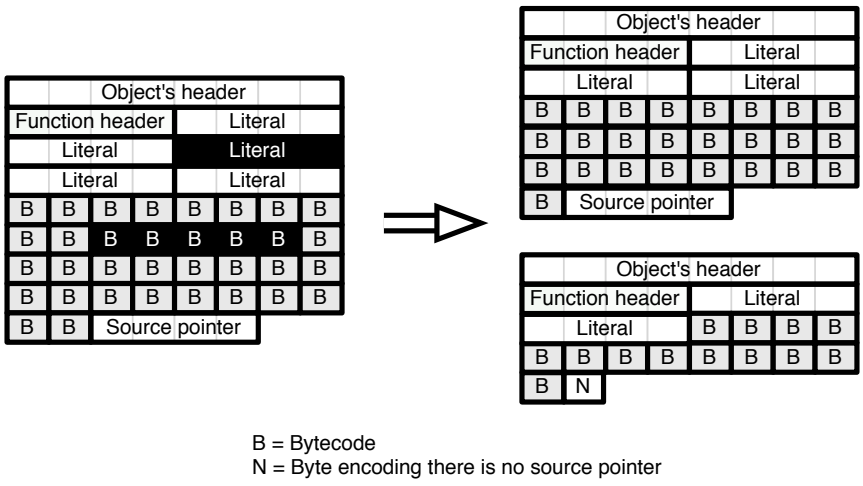


Figure 5.2: Old and new closure representation

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.

CHAPTER 6

Metacircular optimising JIT

Contents

By design, Scorch's optimiser and deoptimiser are written in Smalltalk and are running in the same runtime than the optimised application. This design leads to multiple problems similar to the ones existing in metacircular virtual machines.

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 until deoptimisation is finished.

Hot spots can be detected in any Smalltalk code using conditionnal 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 function. While doing so, the same hot spot may be detected again before any optimised function is installed, leading the optimiser to interrupt itself repeatedly.

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 frame. In this case, the deoptimiser calls itself on one of its own frame, which may require to deoptimise a frame for the same function, leading the deoptimiser to call itself repeatedly.

We call the *infinite recursion* problem this issue where the optimiser or the deoptimiser respectively calls itself. The issue is detailed for each case in the first two sections.

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 to some functions not to be optimised, but the application

keeps running correctly. However, the deoptimiser cannot be disabled temporarily while the application is running as the deoptimiser may be needed to keep executing code. As the optimiser and the deoptimiser have different constraints, they need different solutions for the infinite recursion problem.

This chapter explains the design used to avoid the infinite recursion issue in both the optimiser and the deoptimiser. The problem is solved in the optimiser by temporarily disabling it in specific circumstances. The deoptimiser solves the problem by using a code base completely independent from the rest of the system that cannot be optimised, hence it never requires to be deoptimised. The last section discuss similar design issues 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 the application green thread. 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 Infinite 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 an infinite 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 function. 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 infinite recursion issue, the application is also interrupted when a hot spot is

detected, but 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 function, but another hot spot is detected in the optimiser code. The optimiser keeps restarting the optimisation of one of its function.

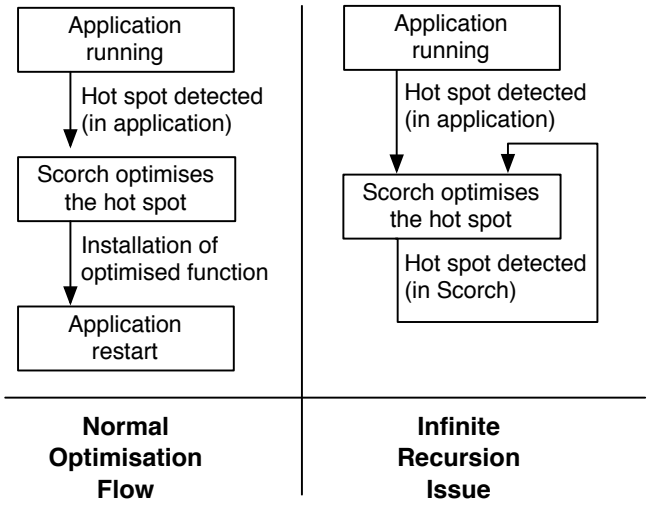


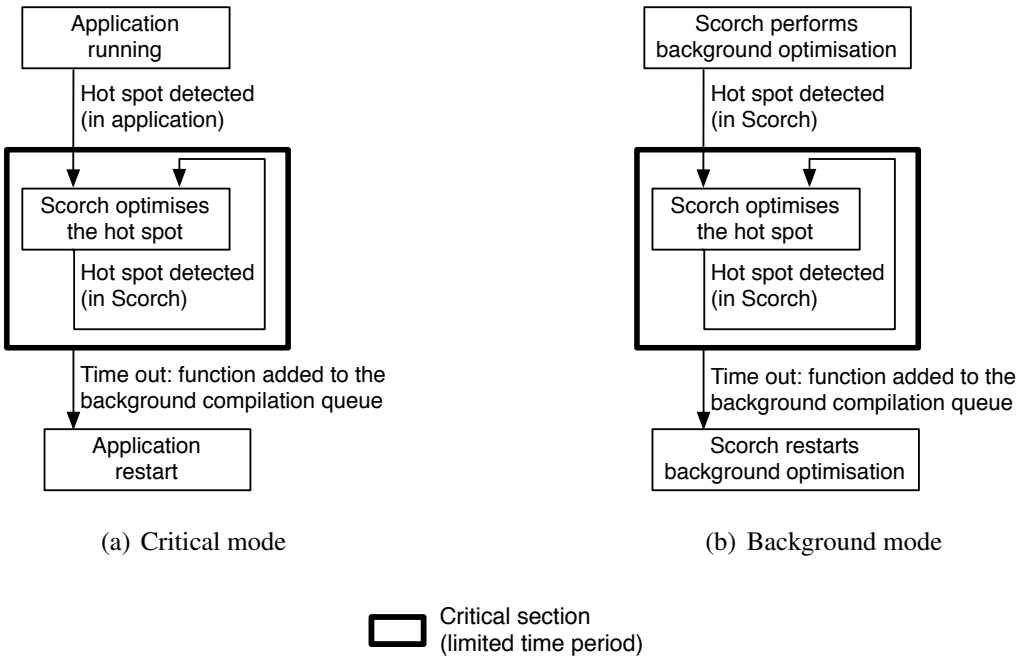
Figure 6.1: Infinite recursion problem during optimisation

In practice, the infinite recursion issue leads to a massive performance loss.

In critical mode, the optimiser has a limited time period to optimise code. If the infinite 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.

It can conceptually happen that the optimiser starts spinning while searching the stack for a function to optimise. In this case, no function can be added to the background compilation queue as the optimiser has not been able to find a function to optimise in the limited time period. In practice, in our case, the stack search code is quite simple and no hot spot can be detected repeatedly in this code.

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 verey slow. Indeed, while the optimiser is running in background mode, it activates itself 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



(a) Critical mode

(b) Background mode

Figure 6.2: Infinite recursion problem in the two optimisation modes

itself during background optimisation, the optimisation of the function may take a significant amount of time.

Eventually, the optimiser optimise most of its own code correctly through the background mode. Once done, both modes can work correctly as no hot spot can be detected in 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 infinite recursion issue does not happen in critical mode.

6.1.2 Current solution

The first solution we wanted to implement was to disable the optimiser when it is running. This first design has a significant advantage: it is quite simple both conceptually and implementation-wise, while it completely avoids the infinite recursion problem. It has however a major drawback: the optimiser cannot optimise its own code any more. Of course, the optimiser may use core libraries that can be optimised. For example, the optimiser may use 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. For this reason, it is possible that while optimising code the optimiser triggers the deoptimiser, and it works perfectly fine.

To implement our first 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. Then, we changed the optimiser to install back the call-back when it resumes the application, after adding a function to the background queue or installing optimised code. This way, we believed the optimiser would never end up in a situation where it optimises itself, solving entirely the problem.

Then, we ran our benchmarks and saw that the problem was solved but the optimiser could still optimise its own code. Our first implementation effectively disabled the optimiser, but only when it was running on critical mode. When hot spots were detected, they were optimised or postponed without any issue as the optimiser disabled itself in critical mode, and the application resumed just fine. When the optimiser was started in background mode, it was not disabled. Hence, in this case, the optimiser was sometimes interrupted by itself in critical mode to get optimised.

This first 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 the production VM, simplicity is really important to keep the code base relatively easy to maintain. For each added complexity in the VM we wonder if the complexity is worth the benefit. This first solution is nice because it is simpler, both to understand and to maintain, than the alternative ones, so the optimising JIT may move to production with this design. Alternative solutions, more complex but with less constraints, are discussed in the next section.

6.1.3 Discussion and advanced solutions

The first solution is working but there is one major drawback: Scorch optimiser cannot optimise itself in critical mode.

With the current solution, hot spots detected inside the optimiser in critical mode are completely ignored and the corresponding profiling counters are reset. If the optimiser attempts to inline code later, it may get confused by some counter values which were reset. The optimiser may speculate that a branch was not taken while in fact the branch was taken but the counters were reset.

We could implement some kind of decay strategy instead of completely resetting the counters. We did not go in this direction because the counters are encoded in 16 bits while the hot spot threshold is at 60,000. Due to the 16 bits encoding

limitation, not completely resetting the counters leads to many hot spot detected without anything happening, slowing down the optimiser code at start-up. Further analysis in this direction are required to conclude anything.

Based on advises from other people, we considered, instead of disabling the optimiser when it is running in critical mode, to postpone the optimisation to the background 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. The optimiser then needs to search the stack to select a function to optimise. As the stack is modified upon execution, it is not possible to save it efficiently so that the optimiser can search it later in the background green thread. As discussed in section 4.2.3, there is no obvious cheap heuristic to figure out what function is the best to optimise based on the current stack (especially, picking the bottom frame function is usually not a good idea). It is however possible, once the optimiser has found what function to optimise, to add it to the background compilation queue.

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. The stack search phase represents less than 1% of the optimiser execution time. The selected function optimisation may be postponed to the background green thread. 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.

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

As the Sista architecture allows to persist optimised code (This is discussed in details in the following chapter), the optimiser code could be preheated through warm-up runs, for example by giving it a list of well-chosen functions to optimise. This way, all hot spots inside the optimiser would be detected ahead of time and optimised. The optimised optimiser code would be shipped to production.

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.

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 Infinite 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 frame representing the execution of the optimised function requires the deoptimiser to restore the stack frames with non optimised functions 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 frame to be able to continue deoptimising code. If each time the deoptimiser calls itself on one of its own frame, the deoptimisation cannot terminate because the deoptimiser needs to call itself again on of its own frame, the application gets stuck in an infinite loop.

Figure 6.3 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 non optimised stack frames from the optimised stack frame and edits the stack. The application can then resume with non optimised code. On the right, in the infinite 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.

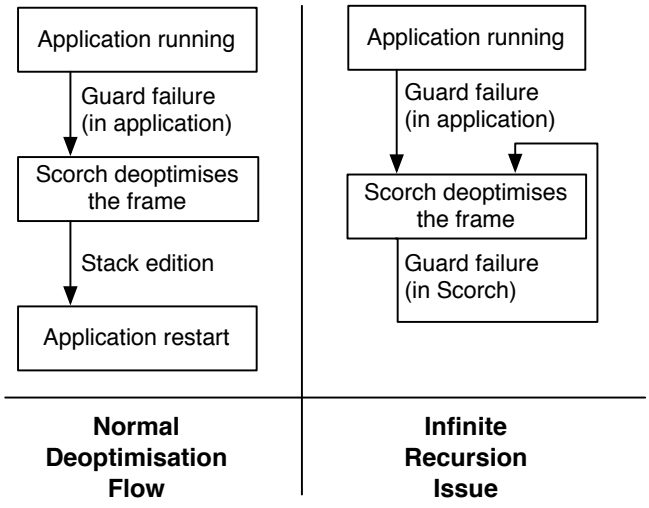


Figure 6.3: Infinite recursion problem during deoptimisation

This problem was solved in the optimiser by disabling it temporarily in specific circumstances. The optimiser could be disabled as the execution could simply fall back to non optimised code. The deoptimiser cannot however be disabled or

any application green thread requiring deoptimisation would not be able to keep executing code. As the deoptimiser cannot be disabled, it is not possible to solve the infinite recursion problem in the same way than the optimiser.

To solve this infinite recursion problem, we implemented two different solutions. The first solution attempts to restore 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 for the first benchmarks, but it did not work correctly with benchmarks creating multiple green threads and making this solution thread-safe had too many constraints. We then designed and implemented a second solution 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 infinite recursion issue for the deoptimiser, we forced Scorch to keep a recovery copy of each method dictionary where optimised v-functions are installed. The recovery copies include only non optimised 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 use the primitive become: 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 non optimised functions. Once the stack is deoptimised, the deoptimiser restores the method dictionaries with the optimised v-functions and re-enables the optimiser.

With this solution, most of the deoptimiser code can be optimised as if an infinite recursion happens, the runtime switches to recovery mode and executes correctly the code. Only most of the deoptimiser code can be optimised as all the Smalltalk code executed from the guard failure call-back to the point where recovery mode is activated cannot be optimised. Such code is not protected by the recovery mode and may suffer from the infinite recursion issue. To avoid the problem, we marked a very small list of functions so they cannot be optimised.

We were able to run most benchmarks with this solutions. 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 caches in the VM to use the non optimised version of functions. Each look-up cache entry referencing an optimised function needs to be edited to refer back to

the non optimised 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. Especially, in our implementation the inline caches are directly written in the native code and each mutation requires partial flush of the cpu instruction cache. The recovery mode therefore slowed down the application for a short while both due to the time spent for the VM to update the caches and for the cpu to restore all the instruction caches.

The second and main problem is that several of our benchmarks use multiple green threads. In this case, the global flag approach does not work as multiple deoptimisations may happen concurrently. One solution is to wrap the code switching from recovery mode to optimised mode and the other way around in semaphores to be green thread safe. However, every function called from the deoptimisation call-back to the point where recovery mode is enabled needs to be marked not to be optimised. 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 primitives but is not allowed to use any external functions. All the deoptimiser classes are marked: their functions do not have profiling counters and the 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 infinite recursion issue cannot happen.

This solution has multiple important constraints.

Firstly, the deoptimiser code cannot be optimised at runtime. This constraint is the least important as there are potential solutions. As deoptimisation is uncommon, arguably, not optimising the deoptimiser code is not a problem. 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 very precise and efficient.

Secondly, the deoptimiser code needs to be completely independent. Only primitives can be used directly because any external function may be optimised, potentially leading to the infinite recursion problem. We analysed what other li-

braires the deoptimiser depended on and remove most of the dependencies by duplicating a bit of code. However, the deoptimiser was relying both on arrays and dictionaries to deoptimise the stack. We create a minimal array and a minimal dictionary as part of the deoptimiser library. The two collection code have now to be maintained in parallel to the core collections.

Lastly, working of the deoptimiser is very tedious. A simple thing such as logging a string in the deoptimiser code require to call a function external to the deoptimiser and may lead to the infinite recursion problem.

Although the constraints are important, we were able to run all our benchmarks with this design. We believe we are going to keep this implementation to move to production.

6.3 Related work

To have an optimising JIT optimising its own code and encounter the infinite 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 the production runtime. Some VMs are metacircular [Ungar 2005](CITE JIKES RVM), which means they are entirely written in a language they can run. However, many metacircular VMs, such as Klein [Ungar 2005], do not feature an optimising JIT.

Overall, there are two main VMs where the optimising JIT optimises its own code at runtime. On the one hand, there 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 (CITE). The Graal compiler effectively optimises its own code at runtime as it would optimise the application code. On the other hand, 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] was initially designed and implemented as part of the Maxine VM [Wimmer 2013], a metacircular Java VM. Graal was then extracted from Maxine. It was then integrated in the Java

hotspot VM (CITE). Graal can be used in two different ways on top of the hotspot VM. It can be used as an alternative optimising JIT, replacing the Java hotspot optimising JIT written in C++ or 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. VMs built with Truffle [Würthinger 2013], a framework allowing to build efficient VMs by simply writting an AST interpreter in Java, are now running using the Java hotspot VM and the Graal compiler as the optimising JIT.

The most relevant use-case, in our context, 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. In our case, 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 or to install optimised code.

In the Graal-hotspot runtime, when a hotspot 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. They include deoptimisation metadata that hot spot is able to understand. When dynamic deoptimisation happens, code written in the hotspot VM (in C++) is responsible for the deoptimisation of the stack.

In our work, Scorch optimiser is able to optimise most of its own code unconditionnally, but the stack search code can be optimised only if the a hot spot is detected while the optimiser is running in background mode. Scorch deoptimiser code cannot be optimised at all. In the case of the Graal-hotspot runtime, the stack search and deoptimisation code is written in C++ and cannot be optimised by Graal. Having the stack search and deoptimisation code in Smalltalk, even if they cannot be optimised normally, allow us to change part of the design such as the deoptimisation metadata without having to recompile the VM. We therefore believe that our approach has an advantage over the Graal-hotspot runtime.

6.3.2 Jikes RVM

Jikes RVM [Alpern 1999, Arnold 2000] optimising runtime compiler is written entirely in Java and can optimise Java code. However, it is not currently able to use

runtime information to direct its optimisations and do not generate deoptimisation guards, making it not that relevant in our context. The runtime compiler uses however an interesting technique [Arnold 2000] to choose what function to optimise. Jikes RVM uses an external sampling profiling native thread. Based on the profiling samples, the profiling thread detect 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.

The n-function generated by the Jikes RVM baseline JIT do not need profiling counters as profiling is performed using the samples taken by the profiling native thread. 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.

We did not investigate in this direction because our VM is currently single threaded. We do not think it is possible to implement something similar in a single-threaded environment.

Conclusion. In this chapter we discussed the main issue existing because the optimiser and the deoptimiser are implemented in 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 infinite recursion. If a hot spot is detected inside the optimiser code, the optimiser may call itself indefinitely to try to optimise it. The deoptimiser has a similar issue when it needs to deoptimise its own code. In both cases, the program may get slow or gets completely stuck.

The optimiser solves this issue by disabling itself when it runs in critical mode (when it interrupts temporarily the application green thread to perform the optimisation). The optimiser cannot optimise itself directly while running in critical mode, it can only optimise the application, which may include libraries used both by the application and the optimiser itself. For functions taking a long time to optimise, the optimiser cannot stop the application for too long or the application becomes unresponsive, hence it postpone the optimisation to a background compilation queue where functions are optimised when the application is in idle. When performing optimisations in the background, the optimiser can optimise itself entirely.

The deoptimiser cannot solve the problem the same way as it cannot be disabled at any time or Smalltalk code cannot be executed any more. The deoptimiser avoids the problem by being written using a small number of classes that cannot be optimised nor call any other libraries.

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 persistance 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. The first section discusses how snapshots are implemented in Smalltalk, including how the code and the running green threads are persisted across start-ups. The second section focuses on the main issue: many VMs today have performance at start-up far worse than at peak performance. Several cases where the start-up performance is a problem are described. Section 7.3 details our solution: optimised v-functions are persisted across multiple start-ups, allowing to persist optimised code and the running green threads. Section 7.4 compares our approach to existing VMs. Few VMs attempts to persist the runtime state across multiple start-ups, but some VMs include different solution to improve start-up performance, solving most of the problem.

7.1 Snapshots and persistance

- discuss snapshots again. and persist v-function, Explain briefly cogit mapping and v-function persistance. Explain saving processes platform independent Restart from snapshot and move from v-function interpretation to n-func in loop and function call.

7.2 Warm-up time problem

- discuss warm-up time problem and why we talk mainly about this. (other people dont persist processes)

7.3 Persistance of optimised virtual functions

- Solution snapshot optimised v-functions - work the same way. platform independent, moving from v-func interpretation to n-function. Discussion on reg alloc and Cogit compilation time.

7.4 Related work

- related work from state of the art.

Conclusion. This chapter discusses how the runtime state is persisted across multiple start-ups, improving the performance during start-up. The next chapter validates the Sista architecture, 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 the Sista architecture, we evaluated in section 8.1 the execution time of a set of benchmarks on the Pharo VM with and without the Sista architecture. The VM was also evaluated in different configuration 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

We evaluate our architecture on a variety of benchmarks from the Squeak/Smalltalk speed center [Felgentreff 2016] 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.

Figure 8.1 shows all the measurements in the form of graphs. The exact values of the measurements are reported in figure 8.2. The following paragraphs describe each a benchmark and its corresponding performance measurements.

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 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.

Figure 8.1(a) shows that adding counters to the VM comes with a slight overhead. A cold start for the benchmarks manages to improve the performance, but due to the ongoing optimisations in the beginning of the measurments, the error is extremely large and thus the run is not significantly faster. Measuring the warmed up image, we see a significant improvement being about 30% faster.

Binary tree. The binary tree benchmark allocates, walks and deallocates binary trees. The benchmark is parameterized with the maximum tree depth, which

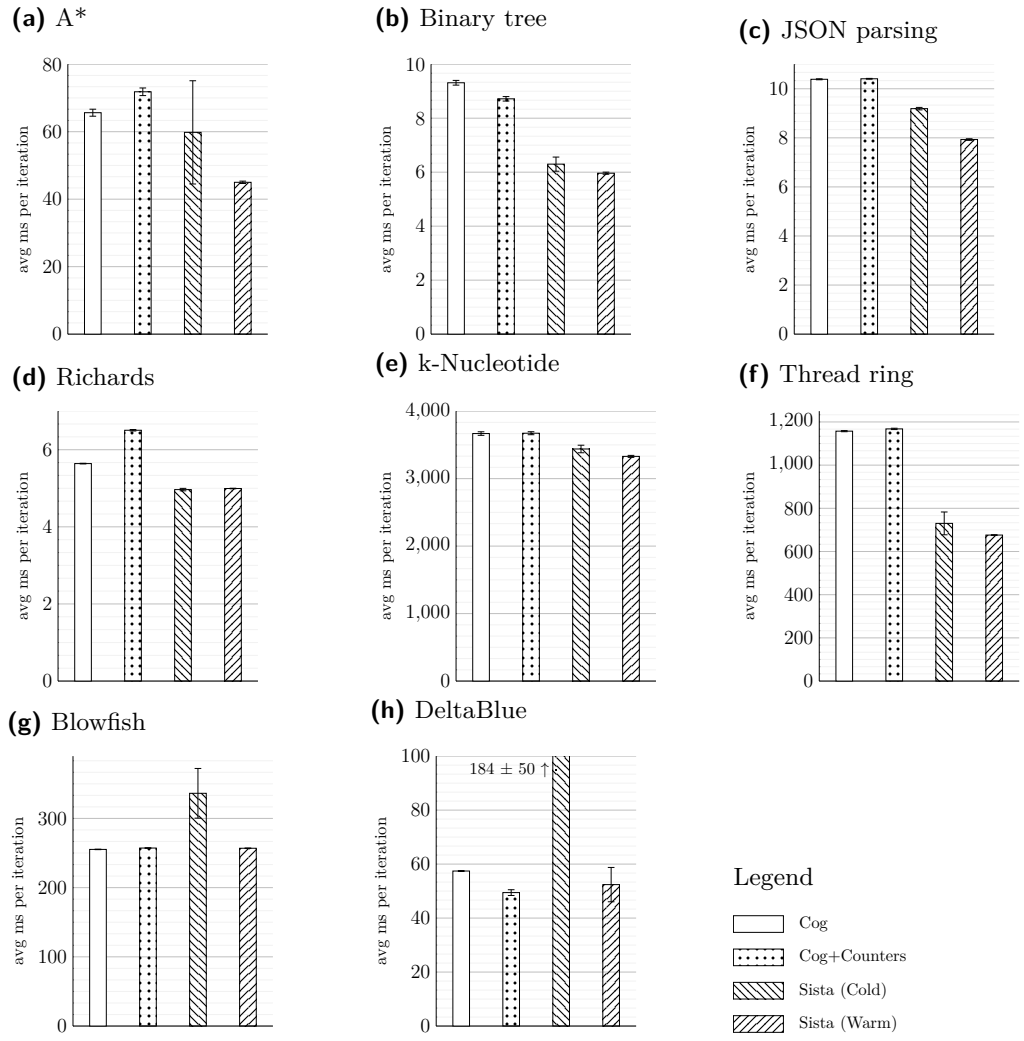


Figure 8.1: Benchmark measurements

we have set to 10.

In this benchmark, peak performance of Sista is about 25% faster than the standard VM, with very little error (Figure 8.1(b)). The overhead of optimisation during a cold start leaves it around 5% behind warm start.

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.

We can see a peak-performance is about 20% faster (Figure 8.1(c)). Even from a cold start Sista quickly improves performance to about 10% faster than Cog.

Richards. Richards is an OS kernel simulation benchmark that focuses on message sending between objects and block invocation. 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.

Figure 8.1(d) again shows a clear decrease in performance after adding counters to the VM. For this benchmark, warm-up is very quick even with our additional optimisations. The top-level benchmarking method is optimised very quickly, and only six optimised methods are installed in total. This quick warm-up means there is no significant difference between the cold start Sista and the warmed up image. Both cases are a little over 10% faster than the Cog VM without counters.

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. Sista manages to optimise it to be about 10% faster, with the cold start showing that the optimisation takes a while to amortize (Figure 8.1(e)).

Thread ring benchmark. 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.

From an unoptimised snapshot, the first iteration of this benchmark is bound to be slow with our architecture, because multiple threads exist with the same code on the stack, each asking the profiler to optimise the same methods, while the optimiser refuses to do so because an optimised function already exists for the frequently used code. This might be a reason for the large error (Figure 8.1(f)). Started again after warm-up, the optimised code runs over 40% faster than the standard VM.

Blowfish. Blowfish tests the performance of bit operations by encrypting and decrypting messages. Each iteration encrypts and decrypts 20 messages with 20 different keys.

Figure 8.1(g) shows no significant differences between Cog, Cog with counters and Sista after warm-up. We explain this with the fact that most time here is

spent in bit operations and very little message sending takes place in comparison. The inlining does not significantly improve the performance in this case, and bit shift, logical AND and OR operations are already treated specially in the default bytecode set so there is not much performance to gain.

DeltaBlue. DeltaBlue is a constraint solver, it tests polymorphic message sending 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.

For our default parameters, the benchmark shows no significant differences between Cog, Cog with counters and Sista – all errors overlap (8.1(h)). We have found that increasing the maximum allowed inlining depth to 14 levels (rather than 12, as used for the other benchmarks), allows the optimiser to produce code that is 20% faster, but it has caused performance problems other benchmarks, so we chosen not to change the variable for the time being.

Comments. 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.

Benchmark	Cog	Cog+Counters	Sista (Cold)	Sista (Warm)
A*	65.63 ± 1.04	71.83 ± 1.13	59.80 ± 15.30	45.00 ± 0.35
Binary tree	9.31 ± 0.09	8.72 ± 0.08	6.30 ± 0.26	5.96 ± 0.04
Blowfish	255.23 ± 0.34	257.20 ± 0.46	336.30 ± 35.90	256.89 ± 0.38
DeltaBlue	57.43 ± 0.17	49.44 ± 1.07	184.60 ± 50.20	52.40 ± 6.36
JSON	10.39 ± 0.02	10.41 ± 0.02	9.19 ± 0.05	7.93 ± 0.03
Richards	5.64 ± 0.01	6.50 ± 0.02	4.96 ± 0.03	4.99 ± 0.01
k-Nucleotide	3667.00 ± 26.40	3672.00 ± 22.20	3439.00 ± 56.00	3329.00 ± 14.80
Threadring	1157.00 ± 2.56	1167.00 ± 3.09	730.20 ± 52.80	676.20 ± 1.66

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

Our optimiser is controlled by a number of variables that have been heuristically 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 (such as DeltaBlue), 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 a new 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 first ten inferred types. Our ordering shows a twofold improvement when compared with the unordered base approach, *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 this 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.

CHAPTER 9

Future work and conclusion

Maybe future work, discussion

——————————— For the end in future work.—- Discussion energy cant be evaluated for other reasons (poor event management...) pre-opt as in android, distib app with amazon lambda and co

APPENDIX A

Complete list of publications

Here is a complete list of my publications in chronological order:

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

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

3. 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.

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

5. 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.

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

Here is the list of my publications waiting for approval:

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

2. Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker and Stéphane Ducasse. Sista: Saving Optimized Code in Snapshots for Fast Start-Up. Submitted to European Conference on Object-Oriented Programming, ECOOP'17, 2017.

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 Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '99, 1999. 2, 3, 21, 76, 77

[Annamalai 2013] Siva Annamalai. *Snapshots in Dart*, 2013. https://www.dartlang.org/articles/snapshots/. 24

[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. 87

[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. 18, 21, 77, 78

[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. 41, 56

[Bala 2000] Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia. *Dynamo: A Transparent Dynamic Optimization System*. In Programming Language Design and Implementation, PLDI '00, 2000. 19

[Béra 2013] Clément Béra and Markus Denker. *Towards a flexible Pharo Compiler*. In International Workshop on Smalltalk Technologies 2013, IWST '13, 2013. 41

[Béra 2014] Clément Béra and Eliot Miranda. *A bytecode set for adaptive optimizations*. In International Workshop on Smalltalk Technologies 2014, IWST '14, 2014. 57, 62

[Béra 2016a] Clément Béra. *A low Overhead Per Object Write Barrier for the Cog VM*. In International Workshop on Smalltalk Technologies IWST'16, 2016. 63

[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. 86

[Black 2007] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Squeak by example. Square Bracket Associates, 2007. 2

[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. 2, 25

[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. 45, 60

[Bunge 2009] Philipp Bunge. Scripting Browsers with Glamour. Master's thesis, University of Bern, 2009. 87

[Deutsch 1984] L. Peter Deutsch and Allan M. Schiffman. *Efficient Implementation of the Smalltalk-80 system*. In Principles of Programming Languages, POPL '84, 1984. 12, 15, 17, 27, 28, 35, 59

[Duboscq 2013] Gilles Duboscq, Lukas Stadler, Thomas WÃ$\frac{1}{4}$rthinger, Doug Simon, Christian Wimmer and Hanspeter Mössenböck. *Graal IR: An Extensible Declarative Intermediate Representation*. In Asia-Pacific Programming Languages and Compilers Workshop, 2013. 5, 14, 21, 51, 76

[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. 87

[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. 87

[Felgentreff 2016] Tim Felgentreff. *Squeak VM speed center*, 2016. http://speed.squeak.org. 81

[Fernandes 2007] Hilaire Fernandes and Serge Stinckwich. *Morphic, les interfaces utilisateurs selon Squeak*, 2007. 87

[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. 46

[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. 19

[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. 20

[Gîrba 2010] Tudor Gîrba. *The Moose Book*, 2010. 87

[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, 25, 28

[Google 2008] Google. *V8 source code repository*, 2008. https://github.com/v8/v8. 13, 14, 16, 18, 46, 53, 60, 76, 86

[Gouy 2004] Isaac Gouy and Fulgham Brent. *The Computer Language Benchmarks Game*, 2004. http://benchmarksgame.alioth.debian.org/. 81

[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. 52

[Group 2015] WebAssembly Community Group. *WebAseembly official website*, 2015. http://webassembly.org/. 18, 52, 53

[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. 12, 15, 27, 35, 46, 59

[Hölzle 1992] Urs Hölzle, Craig Chambers and David Ungar. *Debugging Optimized Code with Dynamic Deoptimization*. In Programming Language Design and Implementation, PLDI '92, 1992. 46

[Hölzle 1994] Urs Hölzle. *Adaptive optimization for Self: reconciling high performance with exploratory programming*. Ph.D. thesis, Stanford, 1994. 10, 11, 13, 18, 20

[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, 25

[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. 87

[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/. 2, 25

[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. 32, 62

[Oracle 2013] Oracle. *OpenJDK: Graal project*, 2013. http://openjdk.java.net/projects/graal/. 14, 21, 51, 76

[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. 21

[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, 20, 76

[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. 52

[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. 18

[Sun Microsystems 2006] Inc. Sun Microsystems. *Strongtalk official website*, 2006. http://www.strongtalk.org/. 11, 13, 20, 23

[Systems 2002] Azul Systems. *Azul official website*, 2002. https://www.azul.com/. 23

[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, 3, 21, 76

[Webkit 2015] Webkit. *Introducing the Webkit FTL JIT*, 2015. https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/. 13, 16, 17, 20, 23, 53, 76

[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, 3, 21, 76

[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. 21, 77