Sista: Saving Optimized Code in Snapshots for Fast Start-Up

Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker, Stéphane Ducasse

Sista: Saving Optimized Code in Snapshots for Fast Start-Up

Clément Béra
RMOD - INRIA Lille Nord Europe, France
clement.bera@inria.fr

Eliot Miranda
Independant consultant, USA
eliot.miranda@gmail.com

Tim Felgentreff
Hasso Plattner Institute University of Potsdam, Germany
tim.felgentreff@hpi.uni-potsdam.de

Marcus Denker
RMOD - INRIA Lille Nord Europe, France
marcus.denker@inria.fr

Stéphane Ducasse
RMOD - INRIA Lille Nord Europe, France
stephane.ducasse@inria.fr

Abstract

Modern virtual machines for object-oriented languages such as Java HotSpot, Javascript V8 or Python PyPy reach high performance through just-in-time compilation techniques, involving on-the-fly optimization and deoptimization of the executed code. These techniques require a warm-up time for the virtual machine to collect information about the code it executes to be able to generate highly optimized code. This warm-up time required before reaching peak performance can be considerable and problematic. In this paper, we propose an approach, Sista (Speculative Inlining SmallTalk Architecture) to persist optimized code in a platform-independent representation as part of a snapshot. After explaining the overall approach, we show on a large set of benchmarks that the Sista virtual machine can reach peak performance almost immediately after start-up when using a snapshot where optimized code was persisted.

CCS Concepts. • Software and its engineering~Just-in-time compilers • Software and its engineering~Runtime environments • Software and its engineering~Object oriented languages • Software and its engineering~Interpreters.

Keywords. Language virtual machine, Just-in-time compilation, Runtime compiler, Object-oriented language.

1. Introduction

Most object-oriented languages, such as Java or Javascript run on top of a virtual machine (VM). 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, 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 unoptimized code. As speculative optimization relies on previous runs to speculate one way or another, the VM needs a warm-up time to reach peak performance. Depending on the virtual machine specifications and the application run, the warm-up time can be significant.

Snapshots. To avoid this warm-up time, this paper introduces an architecture to save a platform-independent version of the optimized code as part of a snapshot. Snapshots are available in multiple object-oriented languages such as Smalltalk (Goldberg and Robson 1983) and later Dart (Annamalai 2013). Snapshots allow the program to save the heap in a given state, and the virtual machine can resume execution from this snapshot later. Usually, compiled code is available in different versions. On the one hand, a bytecoded version, which is on the heap if the bytecoded version of functions is reified as an object (as in Dart and Smalltalk). On the other hand one or several machine code versions are available in the machine code zone. Machine code versions are usually not part of the heap directly but of a separated part of memory which is marked as executable. Snapshots cannot save easily machine code versions of functions as a snapshot needs to be platform-independent and machine code versions of functions are not regular objects.

Overall solution. Our architecture, *Sista* (Speculative Inlining SmallTalk Architecture) works as follows. Our optimizing compiler, after doing language-specific optimizations such as speculative inlining or array bounds check elimination, generates an optimized version of the function using a bytecode representation and does not directly generate machine code. This optimized version has access to an extended bytecode set to encode unchecked operations such as array access without bounds checks similar to the work of Béra et al.(Béra and Miranda 2014). Optimized bytecoded functions are reifed as objects the same as normal bytecoded functions, hence they can be saved without any additional work as part of the snapshot. Then, the VM uses the baseline Just-in-Time compiler (JIT) as a back-end to generate machine code from the optimized bytecoded function. The optimized functions are marked, so the VM can decide to handle differently optimized bytecoded functions.

Dynamic deoptimization is also split in two steps. Firstly, Sista asks the baseline JIT to reify the stack frame from machine state to bytecode interpreter state of the optimized bytecoded function, mapping correctly the register to stack entries and converting object representations from unboxed versions to boxed versions, as it would do for any unoptimized version of the function. Secondly, a separate deoptimizer maps the bytecode interpreter state of the optimized bytecoded function to multiple stack frames corresponding

to the bytecode interpreter state of multiple unoptimized functions, rematerializing objects from constants and stack values.

With this architecture, the Sista VM can reach peak performance almost immediately after start-up if it starts from a snapshot where optimized code was persisted.

Terminology. In the languages supporting snapshots we refer to, non-tracing JITs are available, we call the compilation unit for the JIT compilers a *function*, which corresponds in practice to a method or a closure.

2. Problem: Warm-up Time

The time to reach peak performance in a language virtual machine is a well-known problem and many teams are trying to solve it with different approaches such as snapshots in Dart (Annamalai 2013), tiered compilation in Java hotspot for Java 7 and 8 (Oracle 2011) or by saving runtime information across start-ups (Sun Microsystems 2006; Systems 2002). In some use-cases, this time does not matter. The warm-up time required to reach peak performance is negligible compared to the overall runtime of the application. However, when applications are started frequently and are short-lived, this time can matter.

The problem statement addressed in this article is then: *Can an object-oriented language virtual machine use runtime optimizations without requiring warm-up time at each start-up ?*

We give three examples where the virtual machine start-up time matters.

Distributed application. Modern large distributed application run on hundreds, if not thousands, of slaves such as the slaves one can rent on Amazon Web Services. Slaves are usually rented per hour, though now some contracts allow one to rent a slave for 5 minutes or even 30 seconds. If the application needs more power, it rents new slaves, if it does not need it anymore, it frees the slaves. The slaves are paid only when needed, no application users imply no cost whereas the application can scale very well.

The problem is that to reduce the cost to the minimum, the best would be to rent a slave when needed, and at the second where the slave is not used, to free it not to pay anymore for it. Doing that implies having very short lived slaves, with an order of 30 seconds life-time for example. To be worth it, the time between the slave start-up and the peak performance of the language used has to be as small as possible. A good VM for such kind of scenario should reach peak performance very fast.

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

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

3. Solution: Snapshotting optimized code

3.1 Overview

We solve this warm-up time problem by saving runtime optimizations across start-ups. This way, we are able to start the virtual machine in a pre-heated state. With this approach, we reuse and take advantage of techniques reaching peak performance but with the advantage of being effective without warm-up time.

Platform-independent. We save optimizations across start-ups in a platform-independent way: this implies that we cannot save directly machine code. As our technique depends on snapshots, the platform-dependency depends on the snapshot being dependent on a platform or not.

Bytecode saved. Our approach saves the optimized code as a bytecoded version because the languages with snapshots already support saving bytecoded functions as part of the snapshot. Bytecode is already a compressed and platform-independent representation of executable code. The optimized code is saved using an extended bytecode set to encode unchecked operations.

Simplicity. We try to keep the solution simple by reusing the existing snapshot infrastructure, which can persist the bytecode version of each method. We do not want to extend the snapshot logic to be able to persist machine code as it is very fiddly. More precisely, we would need to extend the snapshot logic with specific code for each back-end supported (currently at least ARMv5, x86, x64 and MIPS little endian) and we would need to handle cases such as position-dependent machine code.

Overall architecture. For the first runs of a bytecode function, the architecture introduced in the paper is quite standard: the first few runs are interpreted by a bytecode interpreter and for subsequent runs, a baseline JIT generates a native function corresponding to the bytecode function and the VM uses it to execute the code. Once a threshold is reached, the bytecode function is detected as a hot spot and the optimising JIT kicks in.

In a classical non meta-tracing VM, an optimised native function would be generated and used to improve the performance. In our case, the optimising JIT is split in two. First, the high-level part of the JIT generates an optimised bytecode function based on the unoptimised bytecode functions and the runtime information provided by the baseline JIT. Second, the low-level part of the JIT, which is the baseline JIT with small extensions, generates an optimised native function from the optimised bytecode function. Figure 1 summarizes the overall architecture.

As all bytecode functions, optimised bytecode functions are platform-independent and can be persisted across multiple start-ups of the VM. Native functions are however always discarded when the VM shuts down.

3.2 Optimization process

The runtime optimization process has overall the same behavior as other virtual machines: in the unoptimized machine code, a portion of code frequently used is detected, recompiled on-the-fly using information relative to the previous runs. Then the virtual machine uses the optimized portion of code. The difference lies in the generation of an optimized bytecoded function in the middle. The full runtime optimization process is as follows:

1. *Hot spot detection:* When the baseline JIT compiler generates an unoptimized version of functions in machine code, it inserts counters on specific locations detailled later in the paper. Each time the execution flow reaches a counter, it increments it by one, and when the counter reaches a threshold, the portion of code is detected as frequently used, *i.e.,* as being a hot spot.

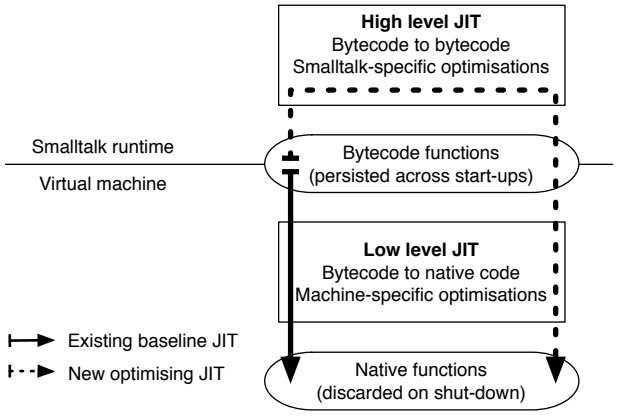

Figure 1: Overview of the architecture

2. *Choosing what to optimize:* Once a hot spot is detected, the VM launches the runtime optimizer. The optimizer tries to find what function is the best to optimize. It walks a few frames in the stack from the active stack frame and based on simple heuristics (mostly, it tries to find a stack frame where as many closure activations as possible available on the current stack can be inlined), it determines a function to optimize.
3. *Decompilation:* The optimizer then decompiles the selected function to an IR (intermediate representation) to start the optimization process. During decompilation, the virtual machine extracts runtime information from the machine code version of the function if available. The decompiler annotates all the virtual calls and branches in the IR with type and branch information.
4. *Overall optimization:* The optimizer then performs several optimization passes. We detail that part in Section 5.2.1.
5. *Generating the optimized function:* Once the function is optimized, the optimizer outputs an optimized bytecoded function, that is encoded thanks to an extended bytecode set. A specific object is kept in the literal frame of the optimized method to remember all the deoptimization metadata needed for dynamic deoptimization. This optimized bytecoded function looks like any unoptimized bytecoded function, so it can be saved as part of snapshots.
6. *Installation:* The optimized function is installed, either in the method dictionary of a class if this is a method, or in a method if it's a closure.
7. *Dependency management:* All the dependencies of the optimized functions are recorded. This is important as if the programmer installs new methods or changes the superclass hierarchy while the program is the running, the dependency manager knows which optimized functions needs to be discarded.

3.3 Deoptimization process

The dynamic deoptimization process, again, is very similar to other virtual machines (Fink and Qian 2003; Hölzle et al. 1992). The main difference is that it is split in two parts: firstly the baseline JIT maps machine state to a state as if the bytecode interpreter would execute the function, second the deoptimizer maps the interpreter state to the deoptimized interpreter frames.

During dynamic deoptimization, we deal only with the recovery of the stack from its optimized state using optimized functions to the unoptimized state using unoptimized functions. The unoptimized code itself is always present, as the bytecode version of the unoptimized function is quite compact. As far as we know, modern VM such as V8 (Google 2008) always keep the machine code representation of unoptimized functions, which is less compact than the bytecode version, so we believe keeping the unoptimized bytecode function is not a problem in terms of memory footprint.

1. *JIT map:* Deoptimization can happen in two main cases. First, a guard inserted during the optimization phases of the compiler has failed. Second, the language requests the stack to be deoptimized, typically for debugging. Once deoptimization is triggered, the first step of the deoptimization process, done by the baseline JIT compiler, is to map the machine code state of the stack frame to the bytecode interpreter state, as it would do for an unoptimized method. This mapping is a one-to-one mapping: a machine code stack frame maps to a single interpreter stack frame. In this step, the baseline JIT maps the machine code program counter to the bytecode program counter, boxes unboxed values present and spills values in registers on stack.
2. *Deoptimizer map:* The JIT then requests the deoptimizer to map the stack frame of the optimized bytecoded function to multiple stack frames of unoptimized functions. In this step, it can also rematerialize objects from values on stack and constants, whose allocations have been removed by the optimizer. The stack with all the unoptimized functions at the correct bytecode interpreter state is recovered. The deoptimizer then edits the bottom of the stack to use the deoptimized stack frames instead of the optimized ones, and resumes execution in the unoptimized stack.

4. Implementation context

All our implementation and validation have been done on top of the Cog virtual machine (Miranda 2008) and its Smalltalk clients Pharo (Black et al. 2009) and Squeak (Ingalls et al. 1997). The main reason that led our choice to Smalltalk is the very good support for snapshots (they're part of the normal developer workflow). This section discusses the specificities of Smalltalk, especially in the case of the two dialects we used, and the implementation of their common virtual machine. We focus on the features that had an impact on the design of the architecture.

4.1 Smalltalk characterization

In this subsection we describe briefly Smalltalk as the language specificities lead to specific design choices. We present a small language overview then three points that directly impact the architecture presented in this paper.

Smalltalk overview. Smalltalk is an object-oriented language. Everything is an object, including classes or bytecoded versions of methods. It is dynamically-typed and every call is a virtual call. The virtual machine relies on a bytecode interpreter and a JIT to gain performance, similarly to Java virtual machines (Oracle 2014). Modern Smalltalks directly inherit from Smalltalk-80 specified in (Goldberg and Robson 1983) but have evolved during the past 35 years. For example, real closures and exceptions were added.

About native threads: the implementations we used have a global interpreter lock. Only calls to external libraries through the foreign function interface and specific virtual machine extensions have access to the other native threads. All the different virtual machine tasks, such as bytecode interpretation, machine code execution, just-in-time compilation or garbage collection are not done concurrently. Therefore, we do not discuss about concurrency as it is not relevant for our system.

We present now the specific Smalltalk features that are impacting our approach and not necessarily present in other object-

oriented languages: first-class activation record, snapshots, and reflective APIs.

First-class activation record. The current VM evolved from the VM specified in the blue book (Goldberg and Robson 1983). The original specification relied on a spaghetti stack: the execution stack was represented as a linked list of function activations. Each function activation was represented as an object that was available to the programmer to be read or written as any other object. Over the years, Deutsch *et al.,* (Deutsch and Schiffman 1984) changed the representation of the stack in the VM to use a call stack as used in other programming languages, where multiple functions activations are next to each other on stack. However, the VM still provides the ability to the programmer to read and write function activations as if they were objects in the state the original bytecode interpreter would provide. To do so, each stack frame is reified as an object on demand.

The reified object acts as a proxy to the stack frame for reads and simple write operations. Advanced operations, such as setting the caller of a stack frame, are done by abusing returns across stack pages. In rare cases, the context objects can be a full object and not just proxies to a stack frame, for example when the VM has no more stack pages available, it creates full context objects for all the stack frames used on the least recently used stack pages. Returning to such context objects can be done only with a return across stack pages, and the VM recreates a stack frame for the context object to be able to resume execution.

The debugger is built on top of this stack reification. In addition, exceptions and continuations are implemented directly in the language on top of this stack reification, without any specific VM support.

Snapshots. In the Smalltalk terminology, a snapshot, also called *image*, 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 processes, the virtual machine can, at start-up, load all the objects from a snapshot and resume the execution based on the active process precised by the snapshot. In fact, this is the normal way of launching a Smalltalk runtime.

In Dart, the word snapshot refers to the serialized form of one or more Dart objects (Annamalai 2013). Dart snapshots can save the whole heap, as part of their *full snapshots*, but as far as we know it is not possible in this language to save processes. Hence, the virtual machine always restarts at the main function once the snapshot is loaded.

One interesting problem in snapshots is how to save the call stack, *i.e.,* the processes. It possible in the Smalltalk virtual machine to convert each stack frame to a context object reifying the function activation. To perform a snapshot, each stack frame is reified and only objects are saved in the snapshot. When the snapshot is restarted, the virtual machine recreates a stack frame for each function activation lazily from the context objects.

Reflective APIs. Most Smalltalk runtimes push the reflection to the extreme: The original Smalltalk IDE is implemented directly on top of the reflective API itself. For example, to add a method to a class, the reflective API on classes to install new methods is used. One interesting aspect is that all the bytecoded versions of the functions (compiled method and compiled closures) are available as objects from the language.

4.2 Existing Cog Smalltalk runtime

The existing runtime, provided with the Smalltalk we used, provides a bytecode compiler and a virtual machine with a bytecode interpreter, a baseline JIT and a memory manager. It is almost completely written in a restricted subset of Smalltalk that compiles to C.

Bytecode compiler. The bytecode compiler is written in Smalltalk itself and compiles to a stack-based bytecode set. On the contrary to the Java bytecode set, none of the operations are typed. Most bytecodes push a value on stack, edit a variable's value or encode a virtual call. Conditional and unconditional jumps are also available to compile loops and branches.

Interpretation. The VM uses the interpreter to run the first execution of a function. The VM trusts the language to provide correct bytecodes, there is no bytecode verifier as in JVMs. At any interrupt point, the VM can recreate a context object from any stack frame if needed. Virtual calls are implemented in the interpreter with a global lookup cache, else the look-up is actually performed.

JIT compilation. As each function activation, at the exception of closure activations, are done through virtual call, the VM uses a simple heuristic: on global lookup cache hit, the function is eligible for JIT compilation. The global lookup cache is voided on full garbage collection and under special circumstances such as the loading of a new class in the system. It is also partially voided when a new method is installed or when there is no room to write a new entry. Hence, in most case a function is compiled to machine code at second activation, except if it is really not frequently used.

Machine code version of functions. The machine code generated uses a simple linear scan register allocator, calling convention for Smalltalk virtual calls and inline caching techniques (with monomorphic, polymorphic and megamorphic send sites). Each generated function is followed by metadata so the JIT can convert at any interrupt point the machine code stack frame to a context object. The metadata includes mainly a mapping from machine code program counter to bytecode program counters, but it can also contain information related to the representation of objects (boxing and unboxing) or the register state. This is used mostly for debugging. In this JIT compiler, no runtime recompilation based on previous runs of the function is done.

The executable machine code zone, where the machine code version of functions are present has a fixed sized. It is a start-up setting set by default at 2 Mb. When the machine code zone overflows, machine code is evicted on a least recently used basis. Each frame pointing to a machine code function which is freed is converted to a bytecode interpreter frame. To avoid converting too many frames, the execution stack has also a limited size. The stack has a fixed number of stack pages allocated at start-up. Stack pages are handled manually by the VM as they are abused for several features such as setting the caller of a stack frame to another frame. If there are no more stack pages available, the least recently used page is freed by converting all the stack frames to context objects on the heap.

From bytecode to machine code. The runtime switches between bytecode interpretation to machine code execution in two places:

- *function entry:* when activating a function through a virtual call or a closure activation, if the function to activate has been compiled to machine code, the machine code version of the function is activated.
- *loop entry:* After a certain number of iteration (currently 20) in the bytecode interpreter, the frame is converted at loop entry from bytecode interpreter state to machine code state.

A machine code stack frame can be converted at any interrupt point to an interpreter frame for debugging. In practice, the runtime switches mostly from the machine code runtime to the interpreter runtime when a call to a function not present in the machine code

zone happens or for specific routines, such as the garbage collector's write barrier.

Platform supported. The runtime is production ready on x86 and ARMv5, while experimental back-ends for x64 and MIPS are available but have not been used in production yet. The Cog virtual machine is deployed mainly on Windows, Mac OS X, Linux and RISC OS (Raspberry pie). It can also run on iOS and Android, but most people prefer to use native applications on those OS. The VM supports running in 32 and 64 bits mode, through the snapshot provided to start the VM is dependent on 32 or 64 bits.

5. Implementation

5.1 Extending the Interface Language - Virtual Machine

A critical part in the architecture is the design of the interface between the language and the virtual machine.

Extending the bytecode set. To support unsafe operations, the bytecode set needed to be extended. Béra *et al.,* describes the extended bytecode set used (Béra and Miranda 2014). The extended bytecode set design relies on the assumption that only a small number of new bytecode instructions are needed for the baseline JIT to produce efficient machine code. Three main kind of instructions were added into the bytecode set:

- **Guards**: guards are used to ensure a specific object has a given type, else they trigger dynamic deoptimization.
- **Object unchecked accesses**: normally variable-sized objects such as arrays or byte arrays require type and bounds checks to allow a program to access their fields. Unchecked access directly reads the field of an object without any checks.
- **Unchecked arithmetics**: Arithmetic operations needs to check for the operand types to know what arithmetic operation to call (on integers, double, etc.). Unchecked operations are typed and do not need these check. In addition, unchecked operations do not do an overflow check and are converted efficiently to machine code conditional branches if followed by a conditional jump.

We are considering adding other unchecked operations in the future. For example, we believe instructions related to object creation or stores without the garbage collector write barrier could make sense.

As the optimized methods are represented as bytecodes, one could consider executing them using the bytecode interpreter. This is indeed possible and we explain later in section 5.4 that in very uncommon cases it can happen in our runtime. However, improving the performance to speed-up the bytecode interpreter or to speed-up the machine code generated using the baseline JIT as a back-end are two different tasks that may conflict with one another. We designed the bytecode set so the machine code generated by the baseline JIT as a back-end is as efficient as possible, not really considering the speed of the interpretation of those methods as they are almost never interpreted in our runtime.

New primitive operation. To extract information from the machine code version of a method, we added a new primitive operation *sendAndBranchData*. This operation can be performed only on compiled methods. If the method has currently a machine code version, the primitive answers the types met at each inline cache and the values of the counters at each branch. This information can be then used by the runtime optimizer to type variables and to detect the usage of each basic block. The primitive answers the runtime information relative to the compiled method and all the closures defined in the compiled method.

New callbacks. As in our implementation the runtime optimizer and deoptimizer are implemented in Smalltalk run and not in the virtual machine itself, we needed to introduce callbacks activated by the virtual machine to activate the optimizer and the deoptimizer.

These callbacks use the reification of stack frame available for the debugger to inform the language which frame had its method detected as a hot spot and which frame has to be deoptimized.

5.2 Overall high-level architecture

In this section we describe the infrastructure available in the Smalltalk runtime: the runtime optimizer, the deoptimizer and the dependency manager.

5.2.1 Optimizer.

Inputs. The optimizer is triggered by a VM callback. As input it receives the reification of the active stack frame where the hot spot was detected. It then figures out which function to optimize and start optimizing. To do speculations, for example guessing the type of an object for inlining, the optimizer asks the baseline JIT compiler for runtime information about the functions it is optimizing. Hence we can consider that there are two main inputs:

1. the execution stack. The bottom stack frame is the one with a function where the hot spot was detected.
2. the runtime information of the optimized function (including all the to be inlined functions).

Outputs. The optimizer outputs an optimized bytecoded function. This optimized function is associated with metadata and dependency information for dynamic deoptimization. The dependency information is needed to know when optimized code needs to be discarded in case of, for example, loading new code at runtime. The optimizer has therefore three outputs:

1. an optimized bytecoded function.
2. metadata to be able to deoptimize the function's stack frame.
3. dependency information to know which code to discard.

Optimizer process. We give in the following a high-level overview of the optimizer. The goal is not to provide the details, but instead give the reader an idea of what kind of optimizations can be done at this level.

1. From the execution stack, the optimizer finds the best function to optimize. If one of the two bottom stack frames is a closure activation, then the optimizer attempts to pick a the method of the stack frame enclosing the closure activation. It is typically a few frames away. If the activation is too far up the stack or if no closure activations are present, the optimizer picks the bottom but one stack frame's function. In addition, if one function has been discarded because it triggered too many dynamic deoptimizations, the optimizer will not consider it. In this case it walks up the stack for a few frame to find a better function to optimize or even cancel the optimization entirely.
2. Once selected, the optimizer decompiles the function to a SSA (Single Static Assignment) IR represented as a control flow graph. It annotates the graph with runtime information if available. It then inlines all the virtual calls so that the current stack, from the stack frame holding the function to optimize to the bottom, would need a single stack frame. Each function inlined is decompiled to the IR and annotated with runtime information if available. The inlining is always speculative based on the existing stack and the types previously met in previous runs. The optimizer inserts guards that trigger dynamic deoptimization if the assumptions met at compilation time are not valid any more as described in (Hölzle and Ungar 1994).

3. The optimizer aggressively inlines all virtual calls present if the runtime type information allows it or the type can be inferred from previous inlining. It starts by inlining the calls in the inner most loops and finishes with the outer most loops. Different heuristics constraint inlining. For example: a recursive function can be inlined up to three times only in itself; and the maximum inlining depth is set to 10. The optimizer stops either when nothing is left to do or if the optimized code is estimated to have reached a certain size limit
4. The optimizer proceeds to variable sized object bounds check elimination (normally accessing a field in an array or a byte array requires a bound check to ensure the field index is within the object). The implementation is inspired by the work of Bodik *et al.,* (Bodík et al. 2000). As the optimization pass focuses on loops, it also optimizes the iterator of the loop to use typed operations for SmallInteger without overflows.
5. A few other passes are lastly done, to remove dead branches if a conditional jump branches on a boolean, to statically resolved primitive operations between constants and to decrease the number of guards if some types can be inferred.
6. The optimizer then generates an optimized bytecoded. To do so, it estimates if each SSA value can be converted to a value spilled on stack or a temporary variable, analyse the liveness of each SSA value that will be converted to a temporary variable and use a graph coloring algorithm to generate the function with the least number of temporary variables. In the literal frame of the optimized function, a specific literal is added to encode the information related to dynamic deoptimization. The information for the dependency manager is installed while inlining.

For this paper, we focused on having simple benchmarks running to prove the optimizations were saved across start-ups. We needed to have a difference big enough to be seen on benchmark results, but additional work is needed for the optimizer to generate very efficient optimized code.

There is no such thing as switching stack frame state from the machine code version generated by the baseline JIT to the optimized function currently. The execution resumes on the unoptimized machine code. At the next activation of the function, the optimized version will be used. As a specific bit is marked in the optimized function, the VM will attempt to use the machine code version of the optimized function at first call.

5.2.2 Deoptimizer

Inputs. The deoptimizer is triggered by a VM callback, with as input the reification of the active stack frame to deoptimize. It then reads the deoptimization metadata from the optimized function activated. Hence we can consider that there are two main inputs:

1. the stack frame to deoptimize
2. the deoptimization metadata

Outputs. The optimizer outputs a deoptimized stack with potentially multiple stack frames corresponding to the optimized stack frame state. The execution then can resume using the deoptimized stack.

Deoptimization process. This high-level deoptimization process is the second step of the overall deoptimization process, once the baseline JIT has already mapped the machine state to the bytecode interpreter state for the stack frame:

1. The deoptimizer extracts the deoptimization metadata from the stack frame to deoptimize function. The metadata describes a set of stack frames in the form of a linked list of frame descriptors. A descriptor includes constants, information on how to access the value from the optimized stack frame (the receiver, a value on stack) or how to recreate objects.
2. The deoptimizer walks over the deoptimization metadata and reconstruct the non optimized frames using the optimized stack frames.
3. The deoptimizer edits the *bottom* of the stack to use the non optimized frames.

We note that only the bottom of the stack can be deoptimized, for the rest of the stack, the frames are lazily deoptimized when the execution flow returns to it.

5.2.3 Dependency management

The dependency manager is currently a simple dictionary, mapping selector to optimized functions to discard.

5.3 Extending the baseline JIT

The baseline JIT compiler was extended because it now needs to:

- detect portions of code frequently used (hot spots) on non optimized machine code.
- read profiling information.
- be able to compile the extended bytecode set.
- generate more optimized code.

Hot spot detection. Each bytecoded method is marked as being optimized or not. If the method has not yet been optimized, the baseline JIT will generate counters in the machine code that are incremented each time they are reached by the flow of execution. This way, once a counter reaches a threshold, the JIT compiler can ask the runtime optimizer to generate optimized code.

Based on (Arnold et al. 2002), we decided to extend the way the baseline JIT generates conditional jumps to add counters 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 (Arnold et al. 2002) allowed us to reduce the counter overhead as branches are 6 times less frequent that virtual calls in the Smalltalk code we observe on production application. In addition, the counters provides 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 for which we wanted to detect hot spots for are covered.

Reading the profiling information. A primitive operation was added in the language to extract the *send and branch data* of the associated machine code of each function. It works as follow:

- If the function is present in the machine code zone, then it answers the *send data*, which means the types met at each virtual call site based on the state of the inline caches, and the *branch data*, which means the number of times each branch was taken at each branch.
- If the function is not present in the machine code zone, then this primitive fails.

The data is answered as an array of array, with each entry being composed of:

- the bytecode program counter of the instruction.
- either the types met and the function founds for virtual calls or the number of time each branch was taken for branches.

Adding the new instructions. As the optimized bytecoded methods have access to an extended bytecode set, the baseline JIT has to be able to compile to machine code all the new instructions present in this extended bytecode set.

Most unchecked instructions are fairly easy to support as they compile to one or two machine code instructions. For example, in Smalltalk accessing to the element n of an array is written array at: n and is the equivalent in Java of array[n]. The normal way the at: primitive is compiled is as follows:

1. Is the argument n a SmallInteger [1] ? If not, fail the primitive.
2. Is the object a variable-sized object such as an array or a byte array ? If not, for example if this is an object with only instance variables, fail the primitive.
3. Is the argument within the bounds of the objects ? If not, fail the primitive.
4. Is the array an hybrid object holding both instance variables and variable fields ? If so, shift the index by the number of instance variables.
5. Unbox the argument.
6. Read the value and answer it.

If the optimizer has proven that the at: operation can be unchecked, *i.e.,* the argument is of the correct type, within the bounds of the array, and has generated explicitly a shift if the array has also instance variables, then the code of uncheckedAt: is simplified to the two last steps.

1. Unbox the argument.
2. Read the value and answer it.

The only instruction that is quite complex to support is the guard. A guard ensures that an object has a specific type or, if not, trigger dynamic deoptimization. It is very frequent in optimized code and hence has to be generated efficiently. The problem is that we needed to check the object type versus in most case a single class, in uncommon cases a list of classes. As objects can be immediate, as for example SmallInteger instances, the guard needs to check the object type differently if it checks it against classes having immediate or non immediate instances.

Optimizing the backend. The original baseline JIT did not do many optimizations. This is because in Smalltalk most operations are done through virtual calls, and each virtual call can potentially be a deoptimization point as any called stack frame can be interrupted and debugged. So the JIT could optimize the machine code in between virtual calls, which in practice means there are almost no opportunities for optimization. There are a few exceptions though:

- virtual calls are optimized using monomorphic, polymorphic and megamorphic inline caches.
- simple but efficient calling conventions allowed to pass the receiver and a few arguments (the exact number depends on the processor used) by registers instead of via the stack.

Existing register allocation. In the case of register allocation, the JIT has accessed to the register through abstract names that are mapped to concrete registers depending on the processor. For example, a register is named *ReceiverResultReg*, as it typically holds the receiver of the current activation and the return value of functions. This register is mapped to %EDX in x86, and other registers on other processors. The JIT generates machine code using an abstract assembly representation that includes a set of operations and has access to these abstract registers, plus a list of extra registers of a variable sized depending on the processor.

[1] A SmallInteger is a 31-bit signed integer in 32 bits and a 61-bits signed integer in 64 bits.

With this design, all the instructions generated would use a specific register. For example, for an instance variable store, the JIT always put the object being edited in %EDX and the value to store in %ECX. In practice, in non optimized functions, very few registers can be dynamically allocated. Most operations are virtual calls and follow specific calling conventions, while operations such as instance variable stores require each value to be in a specific register to easily switch with a trampoline to the C runtime if the garbage collector write barrier requires to do extra computation.

Extended register allocation. In the optimized methods, on the other hand, many registers can be dynamically allocated as values are used in many unchecked operations in a row that do not require registers to be fixed. We extended the register allocation algorithm and allowed the JIT to generate instructions with different registers based on which register was available or not. Even though it is an improvement, the back-end remains fairly naive.

5.4 Extending the interpreter

As in the design discussed in the paper the optimized functions are represented as bytecoded functions, they can potentially be interpreted by the bytecode interpreter. In practice, we marked a bit in all the optimized methods so that the virtual machine tries to compile them to machine code at the first execution and use the machine code version directly. However, in some rare cases, the interpreter happens to interpret an optimized method.

Uncommon JIT behavior. The current VM was designed for an hybrid interpreter and Just-In-Time compiler virtual machine. In rare cases where JIT compilation is too complex to perform, a frequently used method can be interpreted once. For example, as the machine code zone for machine code methods has a fixed size of a few megabytes, it can happen that the just-in-time compiler tries to compile a method to machine code while relinking a polymorphic inline cache. If there is not enough room in the machine code zone, a garbage collection of the machine code zone has to happen while compiling code. It is not easy to perform a garbage collection of the machine code zone at this point as it can happen that the polymorphic inline cache or the method referring it is garbage collected. To keep things simple, in this situation, we postpone the garbage collection of the machine code zone to the next interrupt point and interpret the bytecoded method once.

Naive interpreter extension. As we need to be able to interpret optimized methods, we extended the interpreter to support the extended bytecode set. However, the encoding of the extended bytecode set was designed to generate efficient machine code and is not designed for fast interpretation. For example, the encoding of unchecked instructions (which are critical performance wise) uses three bytes. In the case of interpretation, the time spent to fetch the bytecodes matters, hence performance critical instructions have to be designed in the way that they are encoded in the fewest byte possible. In our case, interpretation is uncommon so interpretation performance is uncritical. We believe it could be possible to enhance our optimizer architecture to generate bytecoded methods that are interpreted efficiently.

6. Evaluation

We evaluate our architecture on a variety of benchmarks from the Squeak/Smalltalk speed center[2] 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 and Brent 2004) and contributed by the Smalltalk community. We have selected

[2] http://speed.squeak.org

these benchmarks to give an indication of how certain combinations of operations are optimized 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 of the Cog VM (interpreter and baseline JIT) our baseline performance. Since we have added counters to the Cog VM to support our architecture, we also measure the performance overhead of maintaining these counters without any additional optimization (Cog+Counters). To show our approach reduces the required warm-up time, we also compare the VM with our runtime optimizer on a snapshot without any optimized code (Sista Cold), and the VM with the runtime optimizer started on a snapshot that already contains optimized code (Sista Warm).

We measured each benchmark 10 times, with the iteration count chosen so the each measurement takes at least 60 seconds. We report the average milliseconds per single iteration for each benchmark and the 90 % confidence interval. For Sista , we start with an already optimized snapshot, so any initial warmup is only due to the Cog baseline JIT creating machine code from the already optimized bytecode. 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. For these measurements, we configured the VM to detect frequently used portion of code when a profiling counter reaches 65535 iterations (they are encoded as *int16*, so this is currently the maximum) and we allow the optimizer up to 0.4 seconds to produce an optimized method. We use a high counter value and allow for a long optimization time, because as the optimizations are saved across start-ups we believe it does not matter if the VM takes a long time to reach peak performance, and we have found these values to produce good performance across a variety of benchmarks. Because Sista is written in Smalltalk itself, it is possible to configure various other optimization 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 optimization steps. In this set of benchmarks, we use a default configuration for the optimizer across all benchmarks. Besides the graphs given below, we report the measurements in Table 1.

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.

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

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

Richards. Richards is an OS kernel simulation benchmark that focuses on message 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.

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

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

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

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

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

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

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

Results.

We distinguish three categories of benchmarks.

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

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

Very slow start-ups. In the case of Mandelbrot and Spectral Norm, ten runs is far from enough to reach peak performance. An important part of the execution time in the ten first runs is spent in compilation, leading the benchmark to be slower than the base VM. If the benchmark is run a couple hundred times instead of only ten times, the performance of Cold Sista would get close to Warm Sista, so this overhead is not a problem in practice for long-running applications. Once peak performance has been reached, Spectral Norm is 10% faster than Cog. The peak performance of Mandelbrot is similar to Cog performance, only removing the overhead of profiling counters, because Mandelbrot is a floating-pointer intensive benchmark and we have not yet implemented floating-pointer optimizations in Sista.

Discussion. For all benchmarks our approach shows significant performance improvements on the scales that we would expect

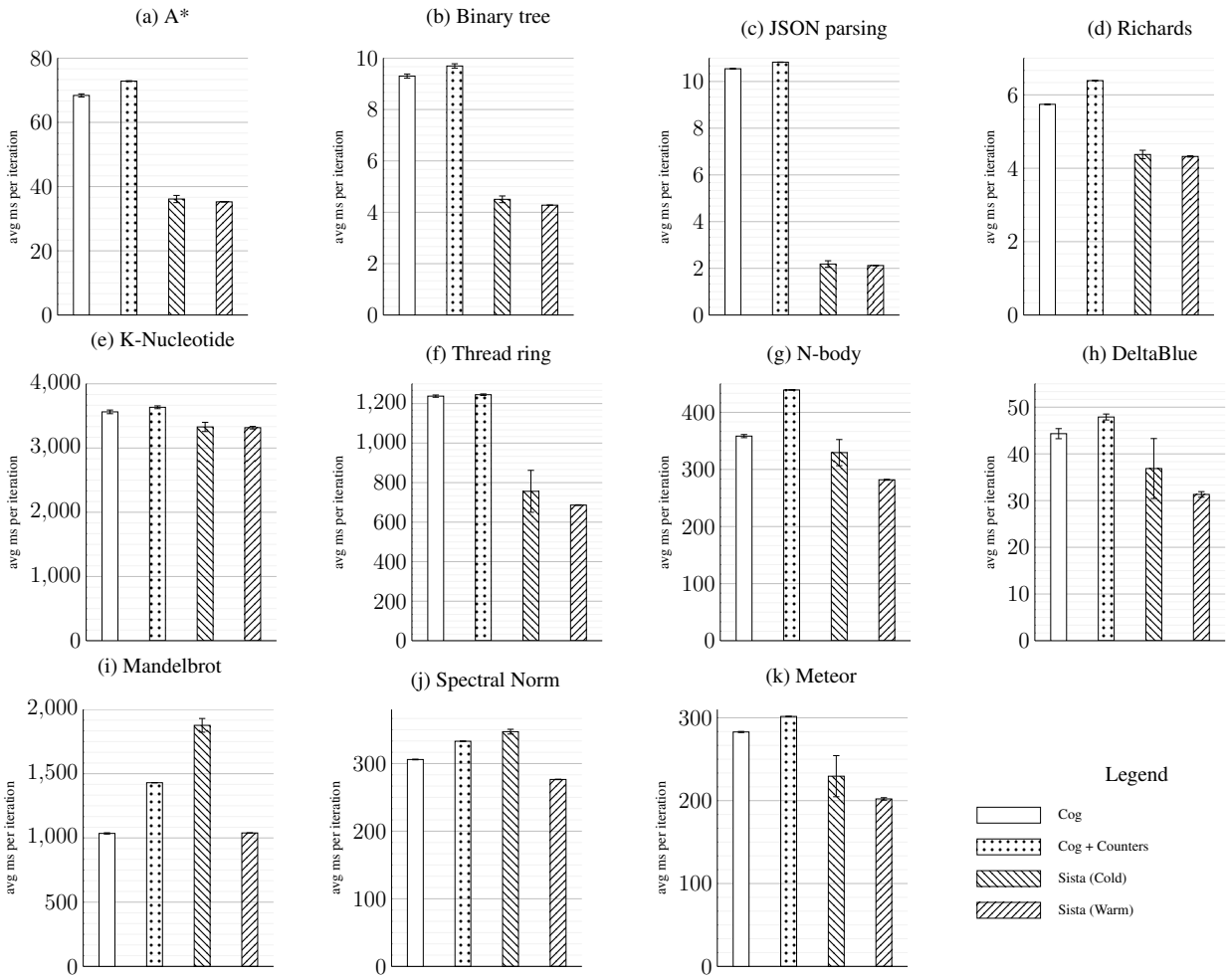

Table 1: Benchmark results with standard errors in avg ms per iteration with 90 % confidence interval

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

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

We ran our VM profiler to profile the VM C code, but as for real world application, the time spent in the baseline JIT compiler generating machine code from bytecode is less than 1% of the total execution time. As the runtime switches from interpreted code to machine code at second invocation for most functions and at first invocation for optimized 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 optimizations 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 optimized bytecoded functions.

Our optimizer 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 optimized methods as well as methods to be inlined, as well as the time allowed for the optimizer to create an optimized method before it is aborted. We have found that for certain benchmarks, these variables can have a great impact. We are working on fine-tuning these default values, as well as enabling heuristics to dynamically adapt these values depending on the application.

7. Related Work

7.1 Preheating through snapshots

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

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

In this case, the code is loaded pre-optimized with inline caches prefilled values. However, optimized functions are not loaded as our architecture allows to do. Only unoptimized code with precomputed runtime information is loaded.

iOS. For iOS, the Dart snapshot is slightly different as iOS does not allow JIT compilers. All reachable functions from the iOS application are compiled ahead of time, using only the features of the Dart 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.

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

Cloneable VMs. In Java, snapshots are not available and used by default. However, Kawachiya and all describe in their work (Kawachiya et al. 2007) extensions to a Java VM to be able to clone the state of a running Java VM in a similar way to snapshots. In this work, the cloned VM duplicates the heap but also the machine code generated by the different JIT tiers. Cloning the machine code improves start-up performance over our approach, as Sista requires to generate machine code from the optimized bytecode functions. However, the clone is processor-dependent: there is no way of cloning with their approach a Java runtime from an x86 machine to an ARMv6 machine. Our approach requires slightly more warm-up time to quickly compile optimized functions to machine code, but is platform-independent.

7.2 Fast warm-up

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

Tiered architecture One solution, used by the most recent JVMs and several Javascript VMs such as V8 (Google 2008) or Webkit, is to have a tiered architecture. The idea is that code would be executed slowly the few first iterations, a bit faster the next iterations, and very quickly after an certain number of optimizations.

If we take the example of Webkit (version in production from March 2015 to February 2016) (Webkit 2015), the code is:t

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

At each step, the compilation time is greater but the execution time decreases. This tiered approach (4 tiers in the case of Webkit), allows to have good performance from start-up, while reaching high performance for long running code.

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

In Strongtalk (Sun Microsystems 2006), a high-performance Smalltalk, it is possible to save the inlining decision of the optimizing compiler in a separate file. The optimizing compiler can then reuse this file to take the right inlining decision in subsequent start-ups. In (Arnold et al. 2005), the profiling information of unoptimized runs is persisted in a repository shared by multiple VMs, so new runs of the VM can re-use the information to direct compiler optimizations.

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

Saving machine code. In the Azul VM Zing (Systems 2002), available for Java, the official web site claims that "operation 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.

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

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

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

Ahead-of-time compilation. The last alternative is to pre-optimize the code ahead of time. This can be done by doing static analysis over the code to try to infer types. Applications for the iPhone are a good example where static analysis is used to pre-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 are 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.

8. Discussion and future work

8.1 Handling exotic Smalltalk operations

Smalltalk provides some operations that are not typically available in other object-oriented languages. We call them *exotic operations*. These operations are problematic for the optimizer. We provide examples for these exotic operations and discuss how the system can handle them.

Become. One operation is called *become*. It allows an object to swap identity with another one, *i.e.,* if an object *a* becomes an object *b*, all the references to *a* now refer to *b* and all the references to *b* refer to *a*. This operation was made efficient using different strategies described in (Miranda and Béra 2015). This feature has some implications in the context of the runtime optimizer. At any interrupt point, there could be a process switch and from the other process, any of the temporary variable of the optimized stack frame could be changed to any object in the heap. This would invalidate all assumptions taken by the optimizer.

Heavy stack manipulation. The other exotic operations are related to stack manipulation. Smalltalk reifies the call stack and allows the program not only to reflect on the call stack, but also to manipulate it. We discussed this in in Section 4 when we explained that for example the developer can set the caller of any stack frame to another frame.

Current Solution: Deoptimization for exotic operations. All these operations are uncommon in a normal Smalltalk program at runtime. They are usually used for implementing the debugging functionality of the language. Currently, profiling production applications does not show that we would earn noticeable performance if we would optimize such cases. The solution therefore is to to always deoptimize the stack frames involved when such an operation happens. In the case of become, if a temporary variable in a stack frame executing an optimized method is edited, we deoptimize the frame. In the case of the stack manipulation, if the reification of the stack is mutated from the language, we deoptimize the corresponding mutated stack frames.

Future Work: Optimizing exotic operations. It could be possible to have the runtime optimizer aware of these features and to handle them specifically. In fact, optimizing the stack manipulation would be similar to the optimization of exceptions. (Ogasawara et al. 2001).

8.2 Platform-dependency and Snapshots

In the case of Smalltalk, snapshots are independent of the processor and the OS used. It is proven as the same snapshot can be deployed for example on x86, ARMv5 and Windows or Linux. However, Smalltalk snapshots are dependent on the machine word size: 32 bit or 64 bit snapshots are not compatible. They are not compatible because the size of managed pointer is different, but also because the representation of specific objects, such as numbers, is different. It is however possible to convert offline a 32 bit snapshot to 64 bit and vice-versa.

As some optimizations related to number arithmetics, such as overflow checks elimination, depends on the number representations, the current optimizing compiler also adds some dependencies to the machine word size. A fully portable solution would either need not to do optimizations on machine word specific number representations or de-optimize the affected code on startup.

8.3 Limitation of the stack-based IR

The bytecoded function (optimized or not) are encoded in a stack-based representation. This can be seen as a problem as it is very difficult to do the optimizations passes on a stack-based IR. To avoid this problem, the optimizer decompiles the bytecode to a non stack-based SSA IR. This implies that the optimizer looses time to translate the bytecode to its IR, and then its IR back to the extended bytecode. The latter is questionable as the optimizer IR has more information than the generated bytecode (for example, it knows the liveness of each SSA value). Information lost here could be profitable for low level optimization such as register allocation and instruction selection.

A possible future work is to design a better representation for bytecoded functions, especially the optimized ones.

We have not invested yet in that direction as we believe that low level machine specific optimizations do not earn a lot of performance for high level languages such as Smalltalk compared to language-specific optimizations. Our belief is based on the optimizing compiler Crankshaft, Javascript V8 (Google 2008) previous optimizing compiler, which is doing very little low level optimizations and is performing very well. Our back-end uses only a few simple heuristic for instruction selection and a simple linear scan algorithm for register allocation.

8.4 Optimizer

We chose to implement the runtime compiler from bytecoded functions to optimized bytecoded functions in Smalltalk instead of C as the rest of the VM. We made this decision because our engineering is more productive in high-level language such as Smalltalk com-

pared to low-level languages such as C. The optimizer is running in the same runtime as the application.

Pros. There were good points in our experience, as for example we could use all the Smalltalk IDE tools and debug the optimizing compiler while it was optimizing a method in the active runtime. Using Smalltalk allows to ignore all the memory management constraints that we have in C.

Cons. However, there are some drawbacks.

Firstly, the runtime now depends on each library the optimizer uses. For example, if you decide to use a specific collection in the runtime optimizer, then editing the collection implementation may break the optimizer compiler and crash the runtime. Hence, we chose to limit as much as possible the dependencies of the runtime compiler, to a minimal part of the Smalltalk kernel. Programming the optimizer is therefore quite different from normal Smalltalk development as we have to keep as few dependencies as possible.

Secondly, the language has now access to both the optimized and non optimized state of each function activation. When the programmer now accesses the reification of a stack frame, depending on the state of optimization, an optimized function activation might be shown. We are adapting the debugging tools to request function activation to be deoptimized when needed. In fact, we are adding an IDE settings: the developer may or may not want to see the stack internals, depending on what he wants to implement. When programming normal applications, the developer usually does not want to see the optimized code, but when programming the optimizing compiler itself, the developer usually wants to see it.

8.5 Process and snapshots.

In the case of Smalltalk, processes are persisted in the snapshot. For example, if a snapshot is taken while some code displays an animation, restarting the VM using the snapshot will resume the animation at the exact same point where it was when the snapshot was taken. To persist a process, the Smalltalk runtime has to persist all the execution stacks.

In a classical JIT compilation approach, only machine code versions of optimized functions are available and stack frames refer to them. As it is very difficult to save directly the machine code version of the method in the snapshot (because of platform-dependency and position-dependent code for example), persisting stack frames referring to optimized functions is problematic. Optimized function are generated in a non deterministic way as the optimizing compiler depends on runtime type information, so it is not trivial to recreate them at start-up.

Persisting processes is difficult in classical JIT compiler. Our architecture solves that problem by allowing to persist bytecoded versions of optimized function. In our case, the VM persists processes by mapping all machine code state of stack frames to bytecode interpreter state, and then persist all the stack frames in their reified form.

8.6 Memory footprint

Usually when dealing with speculative optimizations in JIT compilers, one evaluates the memory footprint taken by the deoptimization metadata. That evaluation would be really interesting in our context as the metadata is split in two parts:

- A part next to the machine code version of the method to map machine state to bytecode interpreter state.
- A part in the literal frame of the bytecoded optimized function to map the optimized stack frame to non optimized stack frames.

Does the split implies a larger memory footprint, and, if so, how much bigger is the memory footprint ? In our implementation, we have kept the metadata almost uncompressed (We used a very naive compression algorithm). Working on an efficient serializer to compress this metadata and an analysis of memory usage is future work.t

9. Conclusion

In this paper we described an architecture that saves optimization across start-ups by saving optimized functions as part of a snapshot. The architecture allows to decrease the warm-up time needed by an object-oriented language virtual machine required to reach peak performance.

A first version has been implemented and it can run simple benchmarks. We need to spend more time stabilizing the optimizer and integrating it with the debugging tools to allow it to be used in production applications. Especially, we are targeting a distributed application deployed at a customer.

Snapshot is not really a well-know technique. To our knowledge, the most popular languages providing this feature are Smalltalk and Dart. Decreasing the warm-up time for virtual machine is an interesting problem as it applies directly on today's application use-cases such as web pages, mobile applications and distributed applications. It seems that snapshot is a useful technique to speed up start-up time in this context, so maybe this technique will become more popular in the future.

References

Siva Annamalai. 2013. Snapshots in Dart. (2013). https://www.dartlang.org/articles/snapshots/.

Matthew Arnold, Michael Hind, and Barbara G. Ryder. 2002. Online Feedback-directed Optimization of Java. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 111–129.

Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving Virtual Machine Performance Using a Cross-run Profile Repository. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 297–311.

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

Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland. 333 pages.

Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: Eliminating Array Bounds Checks on Demand. In *Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 321–333. https://doi.org/10.1145/349299.349342

Dan Bornstein. 2008. Dalvik Virtual Machine internal talk, Google I/O. (2008).

L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 system. In *Principles of Programming Languages (POPL '84)*. ACM, New York, NY, USA, 297–302. https://doi.org/10.1145/800017.800542

Stephen J. Fink and Feng Qian. 2003. 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)*. IEEE Computer Society, Washington, DC, USA, 241–252.

Nicolas Geoffray. 2015. From Dalvik to ART: JIT! -> AOT! -> JIT! internal talk, Google compiler phd summit. (2015).

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

Google. 2008. V8 source code repository. (2008). https://github.com/v8/v8.

Isaac Gouy and Fulgham Brent. 2004. The Computer Language Benchmarks Game. (2004). http://benchmarksgame.alioth.debian.org/.

Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Programming Language Design and Implementation (PLDI '92)*. ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 326–336. https://doi.org/10.1145/178243.178478

Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*. ACM, New York, NY, USA, 318–326. https://doi.org/10.1145/263698.263754

Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. 2007. Cloneable JVM: A New Approach to Start Isolated Java Applications Faster. In *International Conference on Virtual Execution Environments (VEE '07)*. ACM, New York, NY, USA, 1–11.

Chandra Krintz and Brad Calder. 2001. Using Annotations to Reduce Dynamic Optimization Time. In *Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 156–167.

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

Eliot Miranda and Clément Béra. 2015. A Partial Read Barrier for Efficient Support of Live Object-oriented Programming. In *International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 93–104.

Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. 2001. A Study of Exception Handling and Its Dynamic Optimization in Java. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 83–95. https://doi.org/10.1145/504282.504289

Oracle. 2007. JRockit. (2007). https://docs.oracle.com/cd/E13188_01/jrockit/docs142/.

Oracle. 2011. Java HotSpot™ Virtual Machine Performance Enhancements. (2011). http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html.

Oracle. 2014. The Java Virtual Machine Specification, Java SE 8 Edition. (2014).

Vijay Janapa Reddi, Dan Connors, Robert Cohn, and Michael D. Smith. 2007. Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, Washington, DC, USA, 74–88.

Inc. Sun Microsystems. 2006. Strongtalk official website. (2006). http://www.strongtalk.org/.

Azul Systems. 2002. Azul official website. (2002). https://www.azul.com/.

Webkit. 2015. Introducing the Webkit FTL JIT. (2015). https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/.