Assessing primitives performance on multi-stage execution

Sophie Kaleba, Clément Béra, Stéphane Ducasse

Assessing primitives performance on multi-stage execution

Sophie Kaleba
RMoD - Inria Lille
sophie.kaleba@etudiant.univ-lille1.fr

Clément Béra
Software Languages Lab - Vrije Universiteit Brussel
clement.bera@vub.ac.be

Stéphane Ducasse
RMoD - Inria Lille
stepahne.ducasse@inria.fr

Abstract

Virtual machines, besides the interpreter and just-in-time compiler optimization facilities, also include a set of primitive operations that the client language can use. Some of these are essential and cannot be performed in any other way. Others are optional: they can be expressed in the client language but are often implemented in the virtual machine to improve performance when the just-in-time compiler is unable to do so (start-up performance, speculative optimizations not implemented or not mature enough, etc.).

In a hybrid runtime, where code is executed by an interpreter and a just-in-time compiler, the implementor can choose to implement optional primitives in the client language, in the virtual machine implementation language (typically C or C++), or on top of the just-in-time compiler backend. This raises the question of the maintenance and performance trade-offs of the different alternatives. As a case study, we implemented the String comparison optional primitive in each case. This paper describes the different implementations, discusses the maintenance cost of each of them and evaluates for different string sizes the execution time in Cog, a Smalltalk virtual machine.

Keywords Just-in-Time compiler, Primitive, Virtual machine, Managed runtime

1. Introduction

High-level object-oriented programming languages are often implemented on top of a virtual machine (VM). As VMs have become more popular, different techniques have been set up using Just-In-Time (JIT) compilation to improve the overall runtime performance: method JITs (Hölzle 1994) which usually compile methods to native code and tracing JITs (Bala et al. 2000; Bolz et al. 2009) which usually compile linear traces of execution into native code.

On top of its virtual machine, the client language can use a set of *primitives*[1], which are performed directly by the interpreter rather than by evaluating expressions in a method. We distinguish two kinds of primitives:

- *Essential primitives* cannot be performed in any other way. A high-level object-oriented language without primitives can move values from one variable to another, but cannot add two integers together. Many arithmetic and comparison operations between numbers are primitives. Some primitives allow one to communicate with I/O devices such as the disk, the display and the keyboard.
- *Optional primitives* exist only to make the system run faster (Goldberg and Robson 1983). They can be implemented in the client language directly, making its implementation as a primitive optional, but they are often implemented directly in the VM to improve performance.

Optional primitives as client code. The implementation of an optional primitive in the VM rather than the client language improves performance but also often increases the implementation engineering cost. For example, accessing objects in the VM usually requires understanding of the memory layout or implementation details, which are not needed in the client language.

For this reason, some virtual machine implementors attempted to remove most or all optional primitives. Their goal was to use a JIT performing speculative optimizations to be able to implement the optional primitives in the client language without performance loss. Ideally they wanted to get the performance of the same primitives written in the VM. Self (Hölzle 1994) and Strongtalk (Sun Microsystems 2006) were the first to try this approach. Early versions of the Javascript engine V8 (Google 2008) also implemented most of the primitives, including for example Array operations in

[1] We use in the paper the Smalltalk terminology (primitive) as this is the programming language used for our evaluation. Some other programming languages, such as Javascript, prefer to use the term built-in instead of primitive.

Javascript itself. However, without optional primitives implemented in the VM, several problems rise:

- High performance relies on a JIT with speculative optimizations, which is hard to implement, maintain and evolve.
- There is a big performance gap between the baseline performance and the peak performance.
- Even mature JITs fail to optimize some narrow cases, where the performance is drastically slower (Barrett et al. 2017).

Overall, this approach requires high engineering time to get a mature optimizing JIT with speculative optimizations and even then, it leads in practice to unreliable performance.

World border cost. VMs are traditionally implemented in a low-level language such as C or C++. To balance between memory footprint, start-up performance and peak performance, the execution of code is usually done through multiple execution tiers: the first few executions of a code snippet are done through an interpreter and the JIT compiler optimizes at runtime the frequently used code snippets.

This leads to annoying concerns: for example, let's say we implement a primitive in C. Frequently used portion of code calling that primitive are going to be compiled to native code by the JIT. This is problematic since the native code generated by the JIT is not directly compatible with native code generated by the C compiler: a call from one to another may require to edit the stack pointer and the frame pointer from the client stack to the C stack and to spill or move multiple registers. Switching between both usually wastes up to around a dozen native instructions. If the primitive is going to execute many instructions, the switch overhead might be negligible, but if the primitive executes only a few instructions, the overhead can be noticeable.

Measurements. In this paper, we want to measure the complexities and gain of different implementations. For this purpose, we take the example of a string comparison optional primitive. The primitive compares two strings and answers if one string is greater, equal or lower than the other string. The exact specification is detailed in Section 2.4. The primitive is convenient for measurements since it can be executed with strings of small sizes or large sizes, leading the primitive to execute many processor instructions or only a few of them.

We implemented a first version of the optional primitive in Pharo. We then implemented two versions compiling through C compilers to native code, using different parts of our VM infrastructure. Lastly, we implemented a version in the back-end of the JIT. Each time, we tried to write the code in the most efficient way possible. Then, we compared the execution time of all versions on strings of different sizes, showing that overall the most complex implementation is the fastest to execute.

Section 2 describes our implementation context, *i.e.,* the execution model of the VM we used for our evaluation, how it executes primitives and the specification of our string comparison primitive. Section 3 details the various implementations of the primitive we used for the evaluation. In Section 4, we evaluate the performance of the different implementations with different C compilers and string sizes. We also show the native code of the performance critical part of the primitive generated by each version and discuss it. Further sections compare our work to related works, discuss future work and conclude.

2. Implementation context

Our evaluation is based on the Cog VM: a Smalltalk virtual machine. The Cog VM is based on Dan Ingalls' Smalltalk VM (Ingalls et al. 1997) and has been enhanced with a JIT (Miranda 2011) compiling one method or closure at a time to native code. Historically, the VM implemented a 32-bits adaptation of the Smalltalk-80 specifications (Goldberg and Robson 1983), but the Cog VM is now the default VM for multiple programming languages such as Pharo (Black et al. 2009), Squeak (Black et al. 2007) and Newspeak (Bracha et al. 2010).

Section 2.1 describes briefly the compilation process of the Cog VM. Section 2.2 explains briefly the hybrid execution model with an interpreter and a JIT. In Section 2.3, we detail how primitives are executed. Section 2.4 discusses the string comparison primitive specifications used for our evaluation.

2.1 VM compilation

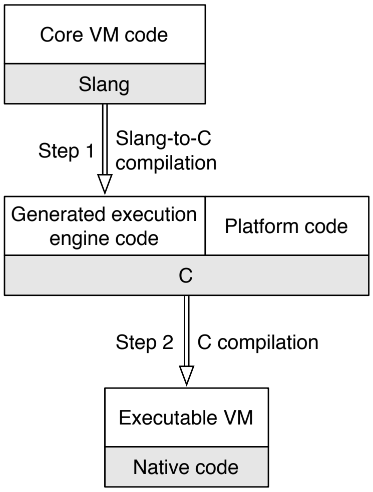


Figure 1: Cog VM compilation

Most of the Cog VM code base 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) is entirely written in Slang. The two main purposes of using Slang code over plain C are:

- to specify with annotations what function needs to be inlined or duplicated with constant parameters, and
- to be able to simulate VM execution, executing the Slang code as Smalltalk code on top of a compiled VM for debugging purposes.

The executable is generated in two steps as shown on Figure 1, similarly to the RPython toolchain (Rigo and Pedroni 2006). The first step is to generate the two C files representing the whole execution engine written in Slang using the Slang-to-C compiler. During the second step, a C compiler is called to compile the execution engine and the platform-specific code written directly in C to the executable VM.

Plugins. The Cog VM can be extended using plugins (Guzdial and Rose 2001). VM plugins enable the addition of new features to the VM. The main benefit of writing a plugin over extending the VM is modularity, plugins are in separate code bases and can be easily added or removed from the VM. Plugins can be compiled as internal or external:

- *External plugins* are compiled as a dynamic libraries distributed with the VM. They can be added or removed from a compiled VM: each dynamic library can be removed, or recompiled separately and modified.
- *Internal plugins* are compiled with the VM executable. They can be added or removed at VM compilation time.

There is a performance overhead for the plugins compared to normal VM code. For example, accesses to object headers are dependent on the memory representation and therefore need to be done indirectly through calls to main VM executable, since the plugin cannot know ahead of time which memory representation is used. Such calls cannot be inlined by the C compiler[2].

2.2 VM execution

The Cog VM executes the client's code using an hybrid runtime with an interpreter and a JIT. The interpreter uses a global look-up cache to speed-up virtual calls. On a cache hit, it requests the JIT to compile that method and the native code version is used instead. In most case, the JIT is used at the second activation of the method. Closures have similar heuristics.

The VM code, including the interpreter code, is compiled by a C compiler and uses the C stack at runtime. During its execution, the interpreter modifies the execution stack of the client language, which is different from its own C execution stack as shown on Figure 2. This means for example that the frame pointer register in the processor refers to a stack frame on the C stack and that a processor push instruction pushes a value on the C stack. When executing the native code generated by JIT, only the client language stack is used. This means for example that the frame pointer register in the processor refers to a stack frame on the client stack and that a processor push instruction pushes a value on the client stack.

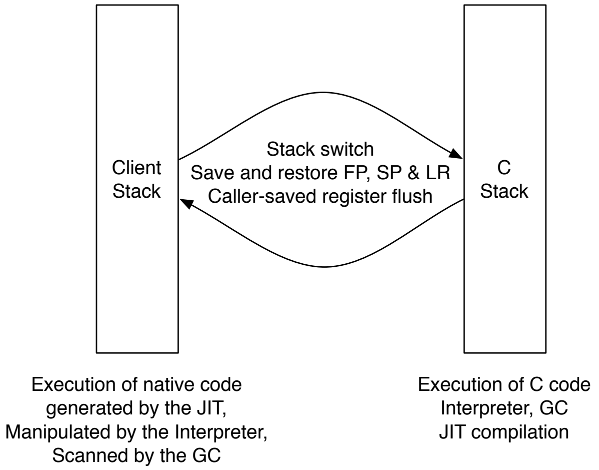

Client Stack
Stack switch
Save and restore FP, SP & LR
Caller-saved register flush
C Stack
Execution of native code generated by the JIT, Manipulated by the Interpreter, Scanned by the GC
Execution of C code Interpreter, GC JIT compilation

Figure 2: Client and C stacks.

Because of this design, switching the execution between the interpreter and the native code generated by the JIT has a cost. Effectively, the VM needs to switch the frame pointer, the stack pointer and the link register (the latter only on some architectures) from one stack to the other one. It also needs to save those pointers to be able to resume execution from one stack to the other one. In addition, caller-saved registers on the C conventions or the JIT convention need to be saved since the VM cannot tell ahead what code is going to be used in the other execution model and what registers are required. We estimate the overhead to up to around a dozen of processor instructions, the exact number of numbers varies a lot (the number of registers to edit depends on the processor used and on the code executed).

Most of the primitives are implemented either in Slang or directly in C. They benefit from the C compiler optimizations to be efficient. When called from the interpreter, activating a primitive is just a C function call (the primitive function pointer is cached in the look-up cache next to the method to activate). When called from the native code generated by the JIT, activating a primitive requires to switch the execution from the client stack to the C stack, perform the primitive, and switch back to resume execution. If the primitive takes a significant amount of instructions to execute, for example, the primitive is copying 100kb from one array to another one, the stack switch dance overhead is negligible. If the primitive is very quick to perform, for example, the primitive is adding two integers which do not overflow, the overhead is massive: the execution cost of the addition goes from a few processor instructions to a few dozen instructions.

[2] In the case of specific internal plugins, C compiler linking time optimizations can be enabled to limit this overhead.

To solve this performance bottleneck, the Cog VM allows one to redefine primitives in Cog's Register Transfer Language (RTL), an abstract assembly which is compiled to native code through Cog's JIT back-end. Such primitives are then generated to native code at runtime when a method annotated with the primitive is compiled to native code. This allows the primitive to be performed in the client stack with the client calling convention, saving register moves and the stack switch dance. Primitives redefined in such way can be entirely or partially redefined. In the latter case, only the common cases are re-implemented on top of the JIT back-end. If an unimplemented case happens at runtime, the generated native code calls the C primitive instead, adding overhead only in uncommon cases.

2.3 Primitive execution

In the Cog VM, primitives are always associated with compiled methods. A compiled method has information in its header to inform the virtual machine if it has a primitive operation or not. When activating a method with a primitive, the primitive function is executed before the method's bytecode. If the primitive succeeds, the primitive returns a result, as if it were the result of the virtual call. If the primitive fails, it does so without side-effects and execution continues executing the method's bytecode, as if the primitive had not been present.

To fail without side-effects a primitive must validate any parameters and any state fetched from them, before changing execution state by performing its operations. Validation involves any of testing for a specific class, testing for bit vs pointer objects, bounds checks, and recursively applying these tests to substructure of the parameters. For example, the primitive that installs a cursor examines the first parameter to check that it represents a valid cursor object, comprised of two bitmaps, one for the image and one for the shape, plus a point to specify the cursor's hotspot.

2.4 String comparison primitive

In this section we describe first the string representations then the string comparison primitive specification.

String representations. Strings are represented on top of the Cog VM in two possible forms: ByteStrings and WideStrings. ByteStrings encode each character of the string in a single byte. The encoding usually follows the Latin-1 (ISO 8859-1) standard. WideStrings encode each character in 32 bits. The encoding usually follows Unicode specifications. In this paper, we will discuss only the implementation of the string comparison primitive for ByteStrings. WideStrings use an implementation entirely implemented in the client language, and so far, it has not been reported as a performance bottleneck for any program of any users. To sort arrays of strings, strings can be compared. They can be compared in the default order (Latin-1) or using different orders, for example, the case insensitive order.

Primitive specification. The String comparison optional primitive takes two or three parameters. The two first operands are the two ByteStrings to compare and the third operand, optional, is the order table (a ByteArray of size 256 encoding the order of characters). If no order is specified, the VM assumes the byte ordering (Latin-1 order) is the order to compare against, and compares directly the bytes of the ByteStrings. If an order is specified, for example a case insensitive order (the entries for the 41 and 61 in the order byte array, respectively characters A and a in the ASCII table, both answer the same value), the primitive compares the bytes through the indirection order. Having the order optional allows to have quicker string comparison in the common case where the Latin-1 order is used. The primitive answers a negative integer, zero, or a positive integer as the first parameter is less than, equal to, or greater than the second parameter.

In the paper we will focus on the comparison of two strings with the default Latin-1 ordering and measure the performance only of this case. The goal of this work was to improve the performance of common string operations, typically string equality which internally uses the primitive, and the cases where the order is non-Latin-1 are considered uncommon.

Conceptually, the primitive with two parameters first checks if the operands are ByteStrings and fails if they are not. It then extract the ByteString sizes from their object header and computes the minimum size. Lastly, it iterates over the two ByteStrings until the minimum size is reached. If a difference is found, if answers the difference between the two different characters. If no difference is found, it answers the difference between the two string sizes.

3. Different primitive implementations

This section describes the different implementations of the string comparison optional primitive and then compare them.

We then explain how each of these versions is executed by the VM and the pros and cons of each and every.

3.1 Different implementations

We distinguish four implementations: the Baseline in pure Smalltalk, the Plugin version, the Slang version and the Slang+RTL version.

Baseline. To evaluate the primitive performance, we use as baseline an implementation in pure Smalltalk. There are multiple ways of writing the comparison of two strings in Smalltalk, following Smalltalk coding conventions, a standard developer would likely use high-level constructs which apply a closure on each element of the strings. We did not use high-level construct and chose to implement the Smalltalk version the most optimized way Smalltalk allows, *i.e.,* we used the constructs to:do: and ifFalse:, which are both recognized by the Smalltalk to bytecode compiler and compiled respectively to loops and branches at the bytecode level[3]. The implementation follows the new specification, hence two methods are available, baselineCompareWith: and baselineCompareWith:order:. Since the focus is only on Latin-1 ordered comparison, we show the code only of baselineCompareWith:.

```
baselineCompareWith: aString
  | c1 c2 length1 length2 |
  length1 := self size.
  length2 := aString size.
  1 to: (length1 min: length2) do: [ :i |
    (c1 := self basicAt: i) = (c2 := aString basicAt: i) ifFalse: [ ^
      c1 − c2 ] ].
  ^ length1 − length2
```

Plugin. Written in Slang, the VM plugin implementation is compiled to C independently to the VM and is then compiled to native code through the C compiler as an internal plugin, without enabling linking time optimizations. Accesses to the ByteString object header (to check if they are actually ByteString and not other objects as well as extracting their size), and accesses to the stack (to read the operands) are done indirectly through the main VM API since they're dependent on the memory representation of objects and stack layout used. These calls generate some overhead since they cannot be inlined. To avoid such calls in the main comparison loop, the primitive requests the main VM to provide a pointer to the first byte of each string and then assumes all bytes are then contiguous.

Slang. The Slang implementation is written inside the interpreter and has access to the full interpreter APIs. It is compiled to C and then to native code as part of the main VM. There is no overhead in accessing the object memory representation or the stack as in the Plugin version.

Slang+RTL. In addition to the previous implementation, we've re-implemented here the primitive on top of Cog's JIT back-end. The primitive is written in Cog's RTL, an abstract assembly whose instructions compile almost one-to-one to native instructions. This primitive requires a more important amount of work to be written: we had to be careful to generate efficient machine code (there is no C compiler to optimize the code) and debugging such code requires the use of a processor simulator such as Bochs. Since only performance critical implementations matter in this context, we've only re-implemented the case where there are two parameters. For the three parameter primitive, the VM falls back to the Slang version.

3.2 Comparison

In this section we compare the four different implementations according to the following criteria:

Recompilation required: Changing the implementation requires only Smalltalk bytecode recompilation (No), requires to recompile a plugin to a dynamic library if compiled as external or the recompilation of the full VM if compiled as internal (Plugin) or requires to recompile the full VM (Yes).

Language familiar to developers: Smalltalk developers are considered to be familiar with Smalltalk only, while Cog VM developers are considered being familiar with Smalltalk, Slang, C and Assembly code. Only Cog's JIT implementors are familiar with its RTL, which is a subset of the VM developers. We use All if all communities are familiar with the language, VM if all VM developers are and VM-JIT if only a subset of VM developers are.

Number of lines of code: We measured the number of lines of code of each implementation.

Number of implementations to maintain: In each case, the client language has to implement a fall-back code in Smalltalk to be executed if the primitive fails. This code is maintained by the client language implementors, and not by the VM implementors. Hence we do not count the fall-back code in the number of implementation to maintain, nor the pure Smalltalk implementation. However, as a VM implementor, we count all the Slang implementations as well as the Cog's RTL one.

Stack switch overhead: Yes if activating the primitive from the native code generated by the JIT requires to switch between the C and the client stack, inducing execution time overhead.

Optimization potential: C cross-file inlining is not performed by default for internal plugins. External plugins don't have access to the whole VM API and have to go through an interface to do so, leading to an overhead, limiting the performance unless the string size is really high.

Control over the generated code: The implementations written in Slang are compiled by the C compilers into native code. The generated code can therefore differ according the compilers and the optimizations they performed. On the contrary, the JIT generates the same native code whichever C compiler is used.

Table 1 summarizes the main pros and cons related to each implementation: the engineering time tends to increase when more low-level languages are used. VM recompilation slows slightly the development process, but the resulting code is usually faster to run. Likewise, the use of plugins brings modularity at the cost of some performance loss.

4. Evaluation

The Plugin, Slang and Slang+RTL implementations of this string comparison primitive have been implemented in the Cog VM. In this section, we compare execution speed of

[3] The bytecode compiler moves the upper limit of the loop ahead of the loop, so (length1 min: length2) is not compute at each iteration.

	Software maintenance & evolution				Primitive execution time		
	Recompilation required	Familiar to developers	Lines of code	Number of implementations	Stack switch overhead	Optimization potential	Code generation control
Baseline	No	All	10	0	No	-	-
Plugin	Plugin	VM	35	1	Yes	Limited	Low
Slang	Yes	VM	35	1	Yes	High	Low
Slang+RTL	Yes	VM-JIT	85	2	No	High	High

Table 1: Comparison between implementations.

these primitives against the other existing implementations. We measure their respective execution times for different string sizes.

4.1 Set-up

To evaluate the performance, we had to compile a special VM with all the different primitive implementations included. The production VM normally only features one string comparison primitive implementation, not three of them. We compiled one VM for Linux with GCC, and one for Mac with LLVM (exact version numbers below). We ran all benchmarks on the same run of the same VM on both platforms.

GCC-Linux. The GCC-Linux evaluation was performed on an Asus Zenbook with Ubuntu 16.04.4 LTS, a 2.2 Ghz processor Intel Core 5, 8 Gb 1600MHz DDR3 of RAM. The Linux VM was compiled using GCC version 5.4.0.

LLVM-Mac. The LLVM-Mac evaluation was performed on a MacBook pro with Mac OS 10.11.6, a 2.9 Ghz processor Intel Core 5, 8 Gb 1867MHz DDR3 of RAM. The Mac VM was compiled using Apple LLVM version 8.0.0 (clang-800.0.38).

The following evaluation was performed on 32 bits Intel VMs (x86).

4.2 Methodology

We assessed the execution time of the different implementations by running the following benchmark:

```
#(0 1 5 100 1000) collect: [ :size |
  | iterations time overhead |
  collection := ByteString new: size.
  collection2 := ByteString new: size.
  size < 100
    ifTrue: [iterations := (100000000 // (size*10 max: 1) sqrt)
     floor.]
    ifFalse: [iterations := (10000000 // size sqrt) floor.].
  overhead := [ 1 to: iterations do: [ :i | ] ] timeToRun.
  time := [ 1 to: iterations do: [ :i |
    collection primitiveStringCompare: collection2 ] ] timeToRun.
  stream nextPutAll: ((time − overhead) asString)].
```

Each benchmark was measured 30 times. Each time, we run a string comparison in a loop and measure the total execution time.

The number of iterations of the loop depends on the string size: the comparison is repeated between 1414 and 10000 times for a short string (5 characters or less), and between 100 and 316 times for a long string (100 characters or more). The warm-up time of the JIT is in this context not noticeable on the performance results: the primitive is compiled to native code usually at the second call as explained in Section 2.2 and the VM sampling profiler (Kaleba et al. 2017) confirms no significant time is spent in the interpreter loop (Between 0 and 1 sample over dozens of thousands are shown in the interpreter loop). The number of iterations allows the benchmarks to run for at least 1 second in the slower implementations to avoid processor and OS noises.

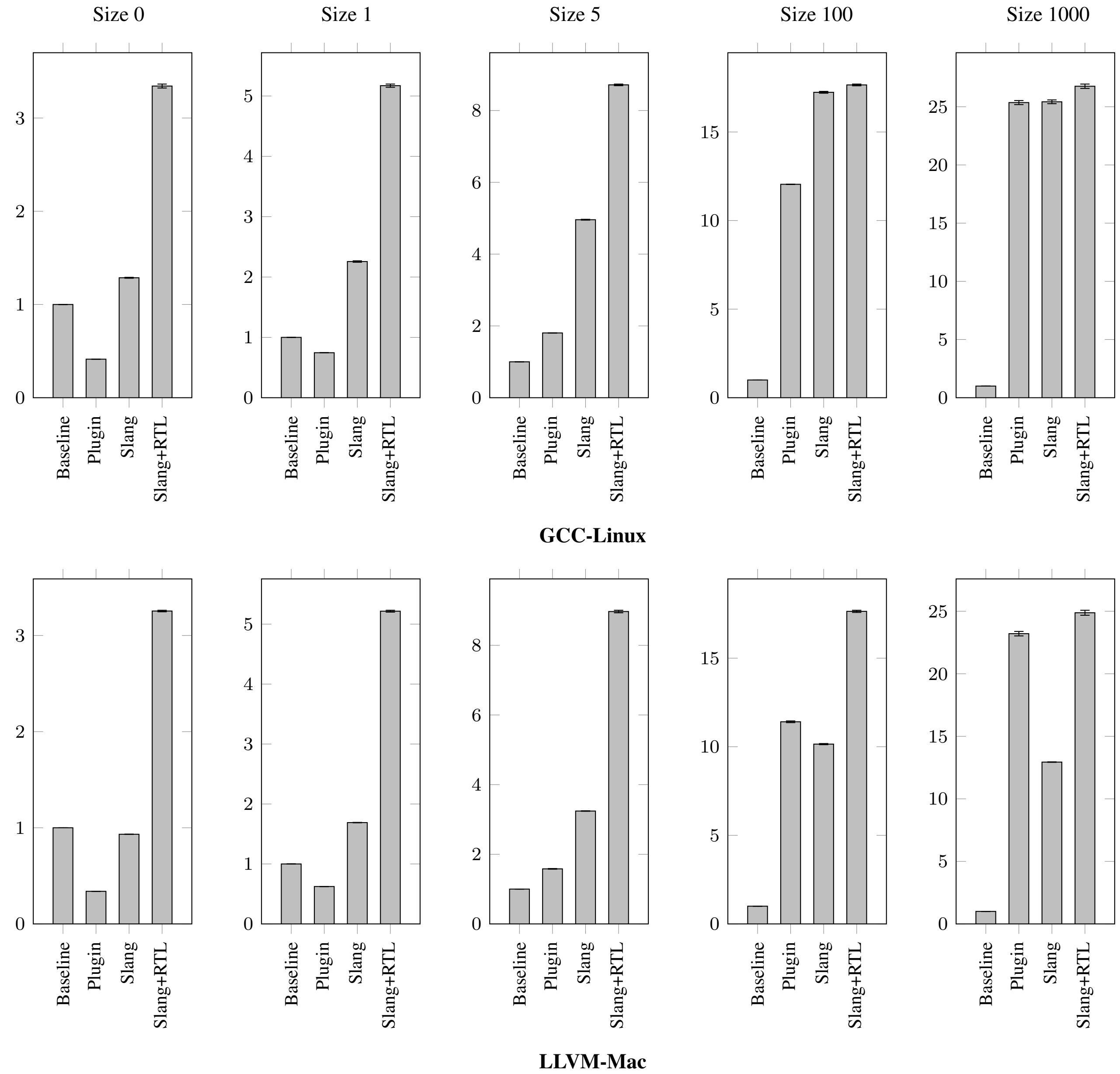
The loop overhead is measured and removed from the measurements (We made sure the JIT did not remove the dead loop to measure it). The two compared strings are equal as this situation corresponds to the worst scenario for most implementations.

4.3 Results

We performed the benchmarks upon five different string sizes: 0, 1, 5, 100 and 1000 characters. In the case of small strings, most of the time is spent activating the primitive and in the code before the main comparison loop, the primitive is performed with a little overall number of native instructions. In the case of large strings, the time spent executing the primitive is dominated by the main comparison loop. Figure 3 and Table 2 show the results in terms of relative speed-up towards the baseline implementation, respectively for GCC-Linux and LLVM-Mac.

These results can be explained based on the last 3 criteria detailed in Section 3.2.

Stack switch overhead. As mentioned in Section 2.2, all implementations written in Slang come with an overhead due to the switch between the C and Smalltalk stacks. This overhead is clearly visible in the bench results in Figure 3 for small strings (size 1 or less): the Plugin implementation is between 58,7 (GCC-Linux) to 66,1% (LLVM-Mac) slower than the baseline's for empty strings.The problem is even more important for the Plugin version, since the beginning of its code, before the main comparison loop, is slower due to the calls to the main VM code. As the string size goes larger, the overhead is less significant because it is absorbed by other performance gains. Regarding this criteria, Slang+RTL version is always the fastest as it is not impacted by the switch overhead.



GCC-Linux

LLVM-Mac

Figure 3: String comparison benchmarks for strings of different sizes on different architectures. Average relative speed-up regarding baseline. The higher the better.

Optimization potential. When it comes to comparing large strings (100 characters and more), the execution time is mainly spent in the comparison loop (Figure 4). The stack switch overhead is still noticeable in the Slang implementation when strings of 100 characters are compared: in GCC-Linux, Slang is 17.2 times faster than Baseline while Slang+RTL is 18,6 times faster than Baseline. Due to the beginning of its code calling the VM code, the Plugin version is still around 0.68 times slower than the Slang and Slang+RTL version for a string of size 100 (Still in GCC-Linux). For strings of size 1000, the performance of all primitives is very similar, since they have almost the same comparison loop, except for the Slang version on LLVM-Mac, as we are about to discuss in the next paragraph.

Control over generated code. One can notice a rather striking performance difference between the 2 set-ups we have. In GCC-Linux, the performance between the Slang and Slang+RTL implementations is really close for long strings (1000 characters wide), with respectively 25.24 and 26.76 relative speed-up factors for a 1000-characters string. Yet performance greatly differs when the LLVM-Mac set-up is used, with respectively 12,94 and 24,88 relative speed-up factors.

		0	1	5	100	1000
Plugin	GCC-Linux	0.413 ± 0.0005	0.746 ± 0.0008	1.804 ± 0.0005	12.05 ± 0.0088	25.36 ± 0.1742
	LLVM-Mac	0.339 ± 0.0007	0.624 ± 0.0012	1.581 ± 0.0092	11.41 ± 0.0474	23.21 ± 0.1789
Slang	GCC-Linux	1.287 ± 0.0048	2.257 ± 0.0116	4.958 ± 0.0111	17.24 ± 0.0515	25.43 ± 0.1621
	LLVM-Mac	0.932 ± 0.0015	1.690 ± 0.0030	3.241 ± 0.0067	10.15 ± 0.0321	12.94 ± 0.0177
Slang+RTL	GCC-Linux	3.343 ± 0.0211	5.171 ± 0.0265	8.715 ± 0.0221	17.66 ± 0.0481	26.76 ± 0.1861
	LLVM-Mac	3.255 ± 0.0079	5.216 ± 0.0153	8.971 ± 0.0359	17.64 ± 0.06	24.88 ± 0.1983

Table 2: Benchmark results with standard errors. Average relative speed-up regarding baseline.

This difference is mainly due to the optimizations performed or not by the GCC and Clang compilers for the Slang version. As depicted in Figure 4, GCC-Linux generates instructions that are very close to the ones generated by the JIT, hence the execution time is similar. On the contrary, LLVM-Mac generates more instructions: it fails at keeping all the operands live across the comparison loops, leading to two extra memory reads per iteration (bolded in Figure 4).

This is an example of one native function in the whole VM and we do not conclude anything about the C compiler capabilites from it. VMs compiled with LLVM and GCC are around equally fast except in narrow cases like this one and LLVM is able to optimize correctly the Plugin version. LLVM likely fails at optimizing the Slang version due to the old GNU C extensions we use in the interpreter code present in the same file to force variables into specific registers, which are better supported by GCC, the only compiler we supported back in the days.

However, we can see that the Slang+RTL version has the most reliable performance since it does not depend on what optimization the C compiler is able to perform or not. This is even more relevant in less common back-ends not shown in this evaluation, such as the MIPSEL back-end we support. On such uncommon processors, C compilers are usually less clever at performing back-end specific optimizations since less engineering time has been invested into it. Cog's RTL is closer to the native instructions and can therefore often generate better code in this case.

Conclusion

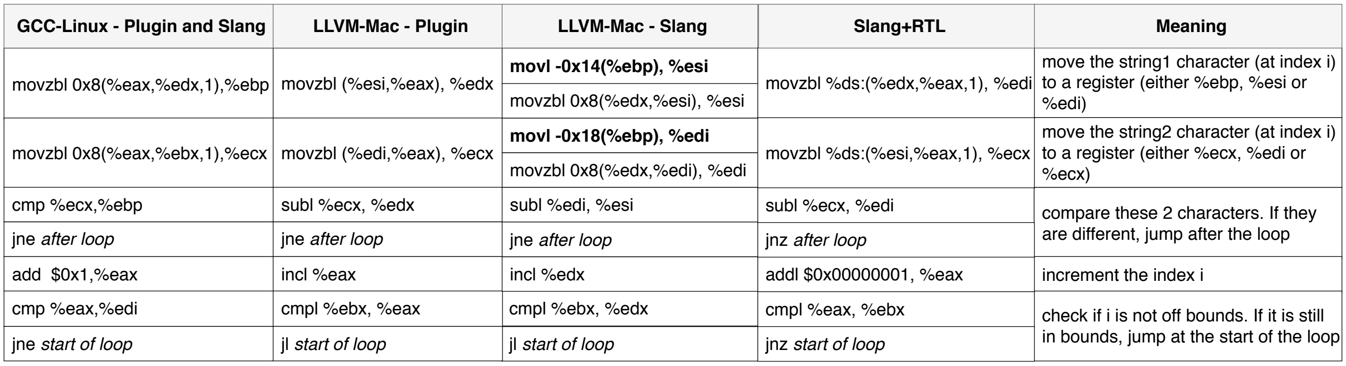
In this section, we compared the performance of the different implementations against a baseline (*i.e.,* pure Smalltalk code) implementation. The results showed the Slang+RTL implementation was always the fastest. Otherwise, most of the performance gaps between the primitive versions could be explained by 3 criteria: stack switching, optimization for large strings, and the control over generated code.

5. Discussion and related work

In some programming languages (Strongtalk, Self, RSqueak), the implementors tried to decrease the overall number of optional primitives by improving the client language performance, reducing the need for such primitives (Sun Microsystems 2006; Hölzle 1994; Felgentreff et al. 2015). In other work (Ballard et al. 1986; Chari et al. 2013), optional primitives can be written directly in a subset of the client language and compiled at runtime to native code, hence they can be modified in the client language without recompiling the VM. Lastly, several VMs implemented their interpreter on top of an abstract assembly (Google 2008; Wingo 2012), allowing the interpreter to use the same register conventions as the code generated by the JIT and avoiding the stack switch cost.

5.1 Decreasing the number of primitives

The philosophy of Self and Strongtalk, both high performance VMs, were to improve the performance of the client



GCC-Linux - Plugin and Slang	LLVM-Mac - Plugin	LLVM-Mac - Slang	Slang+RTL	Meaning
movzbl 0x8(%eax,%edx,1),%ebp	movzbl (%esi,%eax), %edx	**movl -0x14(%ebp), %esi** movzbl 0x8(%edx,%esi), %esi	movzbl %ds:(%edx,%eax,1), %edi	move the string1 character (at index i) to a register (either %ebp, %esi or %edi)
movzbl 0x8(%eax,%ebx,1),%ecx	movzbl (%edi,%eax), %ecx	**movl -0x18(%ebp), %edi** movzbl 0x8(%edx,%edi), %edi	movzbl %ds:(%esi,%eax,1), %ecx	move the string2 character (at index i) to a register (either %ecx, %edi or %ecx)
cmp %ecx,%ebp	subl %ecx, %edx	subl %edi, %esi	subl %ecx, %edi	compare these 2 characters. If they are different, jump after the loop
jne *after loop*	jne *after loop*	jne *after loop*	jnz *after loop*	
add $0x1,%eax	incl %eax	incl %edx	addl $0x00000001, %eax	increment the index i
cmp %eax,%edi	cmpl %ebx, %eax	cmpl %ebx, %edx	cmpl %eax, %ebx	check if i is not off bounds. If it is still in bounds, jump at the start of the loop
jne *start of loop*	jl *start of loop*	jl *start of loop*	jnz *start of loop*	

Figure 4: GCC-Linux and LLVM-Mac: different optimizations on the comparison loop.

language with a mature optimizing JIT featuring advanced optimizations such as speculative optimizations (Sun Microsystems 2006; Hölzle 1994). In this context, the need of primitives is reduced, since the performance gain from using a primitive over the client language is low or inexistant. More recently, Felgentreff *et al.* (Felgentreff et al. 2015) analysed the speed difference between primitives implemented in C, RPython, and Smalltalk. They showed that with the RPython toolchain framework, the speed difference between all three implementations is not that big, hence they could choose to implement some optional primitives in the client language over C or RPython without performance drop to reduce the number of primitives to maintain.

These approaches have the advantage of decreasing the primitive maintenance cost since fewer primitives have to be maintained. The performance problem is however partially solved: only the peak performance is similar with and without the primitives. Since those systems rely on JITs, start-up performance is worse than peak performance, increasing the performance difference at start-up between the client code and the primitive. In addition, the performance can be unreliable with optimizing JITs. The implementors of Morphic (Maloney and Smith 1995) on top of the Self VM were complaining that changing a single line of code could lead to important performance drop and they would not understand why. Those performance drops were due to the optimizing JIT taking different optimization decisions. Aside from the start-up performance and the performance unreliability problems described, these approaches also require a mature optimizing JIT, hard to implement and maintain.

5.2 Primitives written in the client language

In QuickTalk (Ballard et al. 1986) and Waterfall (Chari et al. 2013), the implementors are able to write the primitives in a subset of the client language. The primitives are not part of the VM anymore, but are instead generated as part of the compiled code bytecode representation with a special encoding. They are executed by generating at runtime native code for the special primitive encoding. They can be modified without recompiling the VM and can be performed by the client language implementors instead of the VM implementors, lowering the maintenance cost for the VM implementor.

However, this design has some issues that can lead to performance drops. Many VMs support multiple processor back-ends and multiple memory representation of objects. Primitives written in such DSL are required to be independent from the processor used and the memory representation of objects. If they were written in the VM, a different version could exist for each different processor and memory representation and a different version would be picked at compile-time. Having different versions allows to refine carefully the instructions generated for performance critical primitives (Array accesses for example) and can lead to noticeable speed-ups.

5.3 No stack switch overhead

To avoid paying the stack switch cost between the client Stack and the C stack, some VMs include an interpreter written on top of an abstract assembly. It allows the interpreter to have the same register convention as the native code generated by the JIT, removing the stack switch overhead. For example, the V8 team recently wrote an interpreter named *Ignition*. This interpreter is compiled ahead of time through their JIT back-end to native code. With their primitives and their interpreter written in such way, the V8 runtime rarely needs to switch between the client and the C stack (no need to switch to interpret code, no need to switch for primitives). In Javascript Core, a similar approach is taken with the interpreter named *LLInt* (Wingo 2012). LLInt is written in an abstract assembly code compiled ahead of time to native code compatible with the JIT's calling conventions.

In our context, implementing a similar approach would require us to compile Slang to native code through our JIT compiler back-end or to rewrite the interpreter on top of Cog JIT's RTL. Slang has some abstractions over native code, similarly to C. Compiling Slang ahead of time to native code efficiently requires us to re-implement optimizations implemented in C compilers and we don't want that complexity. Alternatively, we could try to lower the abstractions level of Slang to express native code optimizations, but that would increase the complexity of Slang, which we don't want either. Rewriting the interpreter on top of Cog JIT's RTL is possible but requires a significant amount of work. It would be nice to do so as a future work.

6. Future work and conclusion

In this section we discuss two future work directions and conclude.

Future work: back-end specific. All the primitive implementations we compared are written in a processor independent way, due to the C compiler abstraction for the Slang versions, the abstract assembly abstraction for Cog's RTL version or the VM abstraction for the baseline implementation in Smalltalk. We could rewrite the primitive differently for each back-end we currently support in production (x86, x64, MIPSEL, ARMv6) and for the back-end yet to support (ARMv8, ...). It could be relevant in some cases performance-wise, for example, on Intel processors, the JVM implements string comparison using the SSE4.2 string comparison instructions when available (Davis 2016). We did not go in this direction because the maintenance cost is too high since we would have to maintain a different implementation per back-end.

Future work: adaptive optimizer integration. For the past few years, an experimental adaptive optimizer (a JIT featuring advanced optimizations such as speculative inlining) has been developed on top of the Cog VM (Béra et al. 2017; Bera 2017). The optimizer has specific rules to be able to inline primitive operations without performance loss. We need to carefully measure, in different benchmarks, the performance of the inlined string comparison primitive. We also need to discuss the maintenance cost, for example, do we need a new representation of the primitive for the optimizing JIT to inline it? The micro-benchmarks used in the paper were relevant to show the client stack to C stack switch overhead as well as baseline performance. In the optimizing JIT context, such micro-benchmarks are not really relevant anymore since the optimizing JIT usually entirely optimizes away such benchmarks and real-application performance depends on how well the primitive is inlined and specialized using information from the optimized method (typically, if one operand of the string comparison is a constant string, can the generated code benefit from this knowledge?).

Conclusion. In this paper we compared four different implementations of the string comparison primitive on top of the Cog VM. We showed the performance difference between each implementation for different string sizes. Choosing the right implementation for one's VM is a trade-off between maintenance cost and execution time, the quickest to execute the primitive is, the harder it is to maintain.

Acknowledgments

We thank Levente Uzonyi for providing the benchmark template used in this article and Eliot Miranda for his advice on the implementation.

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

References

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Programming Language Design and Implementation*, PLDI '00, 2000. doi: 10.1145/1988042.1988044.

M. B. Ballard, D. Maier, and A. Wirfs-Brock. Quicktalk: A Smalltalk-80 dialect for defining primitive methods. In *Proceedings OOPSLA '86*, volume 21, pages 140–150, Nov. 1986.

E. Barrett, C. F. Bolz, R. Killick, V. Knight, S. Mount, and L. Tratt. Virtual machine warmup blows hot and cold. 08 2017.

C. Bera. *Sista: a Metacircular Architecture for Runtime Optimisation Persistence*. PhD thesis, Université de Lille, 2017. URL `http://rmod.inria.fr/archives/phd/PhD-2017-Bera.pdf`.

C. Béra, E. Miranda, T. Felgentreff, M. Denker, and S. Ducasse. Sista: Saving optimized code in snapshots for fast start-up. In *Managed Languages and Runtimes*, ManLang 2017, 2017. ISBN 978-1-4503-5340-3. doi: 10.1145/3132190.3132201.

A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Squeak by Example*. Square Bracket Associates, 2007. ISBN 978-3-9523341-0-2.

A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS'09, 2009. doi: 10.1145/1565824.1565827.

G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules As Objects in Newspeak. In *European Conference on Object-oriented Programming*, ECOOP'10, 2010.

G. Chari, D. Garbervetsky, C. Bruni, M. Denker, and S. Ducasse. Waterfall: Primitives generation on the fly. Technical report, Inria, sep 2013. URL `http://rmod.inria.fr/archives/reports/Char13a-Waterfall.pdf`.

J. Davis. How the jvm compares your strings using the craziest x86 instruction you've never heard of, 2016. Blog post: http://jcdav.is/2016/09/01/How-the-JVM-compares-your-strings/.

T. Felgentreff, T. Pape, L. Wassermann, R. Hirschfeld, and C. F. Bolz. Towards reducing the need for algorithmic primitives in dynamic language vms through a tracing jit. In *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '15, 2015. doi: 10.1145/2843915.2843924.

A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

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

M. Guzdial and K. Rose. *Squeak — Open Personal Computing and Multimedia*. Prentice-Hall, 2001.

U. Hölzle. *Adaptive optimization for Self: reconciling high performance with exploratory programming*. Ph.D. thesis, Stanford, 1994.

D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. 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. doi: 10.1145/263698.263754.

S. Kaleba, C. Béra, A. Bergel, and S. Ducasse. A detailed vm profiler for the cog vm. In *International Workshop on Smalltalk Technologies*, IWST'17, 2017. doi: 10.1145/3139903.3139911.

J. H. Maloney and R. B. Smith. Directness and liveness in the morphic user interface construction environment. In *User Interface and Software Technology*, UIST '95, 1995. doi: 10.1145/215585.215636.

E. Miranda. The cog smalltalk virtual machine writing a jit in a high-level dynamic language. In *VMIL '11*, VMIL 2011, 2011.

A. Rigo and S. 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. doi: 10.1145/1176617.1176753.

I. Sun Microsystems. Strongtalk official website, 2006. http://www.strongtalk.org/.

A. Wingo. Inside Javascriptcore's Low-Level Interpreter, 2012. http://wingolog.org/archives/2012/06/27/inside-javascriptcores-low-level-interpreter.