Redesigning FFI calls in Pharo: exploiting the baseline JIT for more performance and low maintenance

Bianchi Juan Ignacio[1], Polito Guillermo[1]

[1]*University of Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL, F-59000 Lille, France*

Abstract

The Pharo programming environment heavily relies on a lot of different C functions. Such functionality is implemented through a Foreign Function Interface (FFI). Pharo implements FFI calls through a single primitive that implements all call cases. This generalization of behavior has performance drawbacks. In this paper, we present a new design for FFI calls. The key goal of the new design is to obtain better performance for the most used callout signatures while keeping maintenance low.

Keywords

Pharo, FFI, JIT

1. Introduction

The Pharo programming environment heavily relies on native libraries. For example, Pharo's IDE and graphical environment use libraries such as Cairo and SDL implemented in C. Such native libraries are accessed through a Foreign Function Interface (FFI) that provides access to libraries respecting a common binary interface (ABI). Typically, those functions are written in the C programming language and compiled through a standard compiler such as GCC or Clang.

As of today, all Pharo FFI calls are handled by a single primitive receiving as argument the signature of the foreign function, and the list of values that should be used as function arguments. Such a primitive validates the function signature, transforms the function arguments following the signature types, and finally performs the function call using `libffi` [1]. `libffi` is a library that handles FFI in a portable way, implementing the entire calling convention, *i.e.,* the convention stating how arguments are passed to functions and how to access their result. This design suffers from performance overhead because:

- `libffi` trades-off performance for generality.
- the single primitive that receives the function specification as an argument forbids us from JIT compiling it, since the same primitive is used from different call sites with different function signatures. This means that many checks that could be constant (*i.e.,* function signature does change once an FFI call site is defined) should be checked at run time.

This paper presents a new design for FFI calls in the Pharo Virtual Machine [2, 3]. This new design aims to obtain better performance for the most-used callout signatures while keeping maintenance low. The solution is based on the following key points:

New FFI call bytecode. Using a bytecode instead of a primitive allows us to benefit from the context available at compile time (*e.g.,* method literals). Such compilation context allows us to stage several checks at JIT compile time and reduce runtime overhead.

`libffi` avoidance for the fast path. When JIT-compiling an FFI call with a commonly used function signature, we generate machine code respecting the system's calling convention using pre-existing support and avoid `libffi`.

Fallback on the old generic implementation. Our solution only supports optimizing a fixed set of function signatures defined statically in the source code. We extracted such commonly used function signatures by profiling existing applications—all non-optimized cases fall back into the old mechanism using the pre-existing primitive.

Our benchmarks show a ∼12x improvement over the baseline implementation when the JIT compiler is active, and a ∼3x improvement when JIT compilation is not active. Moreover, signatures that are rarely used or complex to implement and fallback on the old implementation see no degradation in performance.

2. Context: optimizing FFI calls

Pharo implements FFI calls through a single primitive that implements all call cases. The calls that are the most used are handled the same as the ones that are used only once. Pharo VM leverages `libffi` to support such generality. `libffi` is a library that handles FFI in a portable manner, implementing the calling convention of many different architectures.

2.1. Current implementation overview

The current FFI implementation as per Pharo 12 uses the Unified FFI framework [4] (UFFI). In UFFI, FFI function calls are done through normal methods that are *bound* to external functions. FFI bindings are expressed using the ffiCall: message, as shown in Figure 1. The method in the figure shows a method bound to a function named f with argument arg of type int and returning a void*.

```
MyClass >> myMethod: arg
    ^ self ffiCall: #(void* f(int arg))
```

Figure 1: A Pharo method defining an FFI binding

UFFI extends the Pharo bytecode compiler and transform all methods sending the ffiCall: to introduce a runner and an externalFunction, as illustrated by Figure 2. The runner is an object driving the external function execution, specifying typically if the call should be synchronous or asynchronous. The externalFunction is an object gathering all necessary meta-data for the FFI call, including the function signature and the function pointer. Both the runner and the external function are generated by the UFFI plugin and stored as literals in the compiled method and do not change during the life-cycle of the application.

```
MyClass >> myMethod: arg
   ^ runner
      invokeFunction: externalFunction
      withArguments: {arg asInteger}.
```

Figure 2: The FFI binding as it is transformed by the UFFI framework

The actual FFI call is performed by the runner when it is sent the invokeFunction:withArguments: message. Both implementations of this message, synchronous and asynchronous, are defined as *primitive methods*. Moreover, this message receives an array of objects that will be used as function arguments, which is built dynamically on each FFI call. This array of arguments goes through a process of transformation to native types, also known as *marshaling*, and described in the next section.

2.2. Marshaling by example

Marshaling is the process of transforming objects between two different representations to allow interoperability between different technologies. When using FFI, we consider marshaling the process of converting Pharo objects to native values as expected by C functions and doing the inverse with return values. The process of marshaling is split in two: a *high-level* marshaling and a *low-level* one. The *high-level* marshaling takes as input arbitrary Pharo objects and outputs primitive Pharo objects such as small integers, floats, strings, and external addresses. The *low-level* marshaling takes primitive Pharo objects and outputs native equivalent values.

The high-level marshaling is introduced by the UFFI code transformation. In the example shown in Figure 2, the declared function signature of f expects an int. The function argument is then transformed to an integer using the asInteger message. In the case of more complex function signatures, generated bytecode includes other kinds of messages to consider *e.g.,* floats, structs, and strings.

The low-level marshaling is implemented in the Virtual machine, in the primitive. It typically requires the untagging of tagged objects *e.g.,* converting a Pharo's `SmallInteger` to a `C int` or the unboxing of external references (extracting the actual external address from the Pharo object `ExternalAddress`).

2.2.1. `libffi` integration

Performing the FFI call requires using the `ffi_call` function defined by `libffi` and statically linked with the VM source code. This function requires four different arguments as shown in Listing 3.

A `cif` object: a description of the external function signature. Currently, the cif pointer is built by the UFFI plugin and wrapped inside the external function object.

A function pointer to call, `fn`: The pointer to the function to call is looked up by the UFFI plugin and wrapped inside the external function object.

Arguments and return value holder. A collection of memory addresses pointing to the passed arguments, and a holder address for the return value.

```
void ffi_call(ffi_cif *cif,
         void (*fn)(void),
         void *rvalue,
         void **avalue);
```

Figure 3: An example of a Pharo method

2.3. Identified problems in the current implementation

We have encountered three main problems with the current implementation:

Function signature is known at run time. The external function, its signature and the call arguments are all accessed at run time. While function arguments change from one call to another, the external function and its signature remain stable across calls from the same call site. However, the current implementation does not take advantage of such knowledge.

Cogit JIT compiler does not allow primitive specialization. The Cogit JIT compiler is a non-optimizing method compiler that does a one-to-one mapping between Pharo bytecode methods and their natively compiled code. This compiler does not automatically generate multiple versions of a single method specialized *e.g.,* for its arguments. Thus, even if JIT compiled, a primitive will have only one native version and not be specialized per function call signature.

Generality vs performance. The entire architecture trades off performance for generality. Having a single primitive supporting all cases forces the dynamic construction of the argument array, producing unnecessary stress on the garbage collector. Moreover, both `libffi` and the primitive using it need to support all the existing calling conventions, the rarely used ones as well as the most used ones.

Our goal: Our goal is to propose a new design that splits (a) a fast path allowing specialized JIT compilation of commonly-used function signatures from (b) a slow path that implements a general form and supports all other cases and presents a performance similar to the current implementation.

3. Towards a more efficient FFI design

We propose to extend the Pharo VM and the Opal bytecode compiler with a new FFI call bytecode supporting *synchronous FFI calls*. This new bytecode instruction is not supposed to replace completely the current implementation. Our goal is for them to coexist: the current implementation will be used as fallback mechanism for the slow path and asynchronous calls.

We will refer to the newly introduced bytecode for dealing with FFI calls as `bytecodeFFICall`. `bytecodeFFICall` is implemented in both the interpreter and in the JIT, this last one with a particularity: The JIT'ted implementation of `bytecodeFFICall` is specialized at compile time. The set of function signatures is fixed and defined statically in the source code. We say that those function signatures that we chose are *supported*. For those *unsupported* signatures we fall back to the same primitive that the current implementation uses.

The interpreter implementation of `bytecodeFFICall` is still general: it supports all kinds of function prototypes. Even being general we found some other ways to optimize the FFI calls, taking advantage of the context available to us when compiling the new bytecode, we will describe this in more detail in the following sections.

3.1. The new Bytecode

The `bytecodeFFICall` is a 2-byte bytecode. The first byte is the opcode and the second byte encodes two 4-bit numbers. These values will be indices in the table of literals corresponding to the `CompiledMethod` containing the bytecodes. These indices will corresponds to:

- A description of the external function to call. This includes not only the name and the arguments of the function but also the prototype of it. **??** describes what this object looks like.
- The Runner. For the fallback cases, as previously described.

Also, differently from the existing implementation, this bytecode avoids the creation of intermediate arrays: all function arguments are pushed to the stack. The bytecode knows the number of arguments to pop from the function meta-data found in the literals. To illustrate the new bytecode, consider the bytecode sequence of the method `myMethod` shown before, shown in Figure 4

```
MyClass >> myMethod: arg
    pushArgument: 0
    send: asInteger
    ffiCall: f
    returnTop
```

Figure 4: A method calling function `f` using our new bytecode

3.2. Fallback to the current implementation

When dealing with primitives it is common to encounter some cases where the primitive just does not work so in that case it will give control to the interpreter. Then, the interpreter will interpret the method's fallback bytecode. In the bytecode, in contrast to primitives, there is no such notion as a *Bytecode Failure* Instead, a customary solution for this is to introduce *callback messages e.g.,* send a `doesNotUnderstand:` or `mustBeBoolean` message.

In our case, we decided to treat *failure* cases in the bytecode by sending the message `invokeFunction:withArguments` to the Runner object (See Figure 2), which will in turn call the general-case primitive. This allows the new implementation to not handle the errors directly.

4. Evaluation

At the time of writing this article, our `bytecodeFFICall` has support for optimizing the two function prototypes listed below. We chose two because we wanted to first try the idea before implementing all the prototypes we would like to support in the future.

- `uint64_t fn(uint64_t)`
- `void fn(pointer)`

We chose those two signatures because:

- They are simple to implement, so the necessary machinery to implement support for them is not complex.
- Through some micro-benchmarks using BlocBenchs [5], we found that they are part of the most used signatures. BlocBenchs is a project to profile and benchmark Bloc [6] which is a framework for graphics in Pharo using FFI calls. BlocBenchs has already all the profiling and benchmarking infrastructure for us to rely on, which enabled us to extract the most used external function signatures quite easily.

In the following section, we will describe the benchmarks we have made. When we evaluate the performance for our *supported* prototypes we refer to the two previously mentioned. We say a function prototype is *supported* if the JIT implementation optimizes it in the fast path.

4.1. Benchmarks

This section presents the benchmarks we run to compare our new design against the current implementation and to see how the two new implementations (interpreted vs. JIT'ted) differ.

To do the benchmarks we decided to compare the following combination of cases:

- `bytecodeFFICall` vs. current implementation
- Where the external function prototype is supported and where it is not.
- With the Pharo VM built just with the interpreter and no JIT (StackVM) vs. the Pharo VM built as default (interpreter + JIT).

The first case is the most important comparison we want to make: our new proposed design against the current one. For the second case, we wanted to make sure that our fallback mechanism was not introducing a negative performance impact. For the third case, we wanted to evaluate how much speed-up the JIT'ted version would bring us. Comparing the interpreter-only versions would tell us the overhead of doing the checks at compile time vs. at run time.

Naming. Figure 5 compares the performance of four different implementations for FFI calls over three different benchmark cases.

- newStack and oldStack: refers to the new (`bytecodeFFICall`) and the old (current) implementation respectively, with JIT compilation inactive. Only the interpreter version of each.
- newStock and oldStock: refers to the new (`bytecodeFFICall`) and the old (current) implementation respectively, with the Stock Pharo VM, which has the JIT compilation active.
- Supported or not supported signature refers to whether that function signature is optimized.

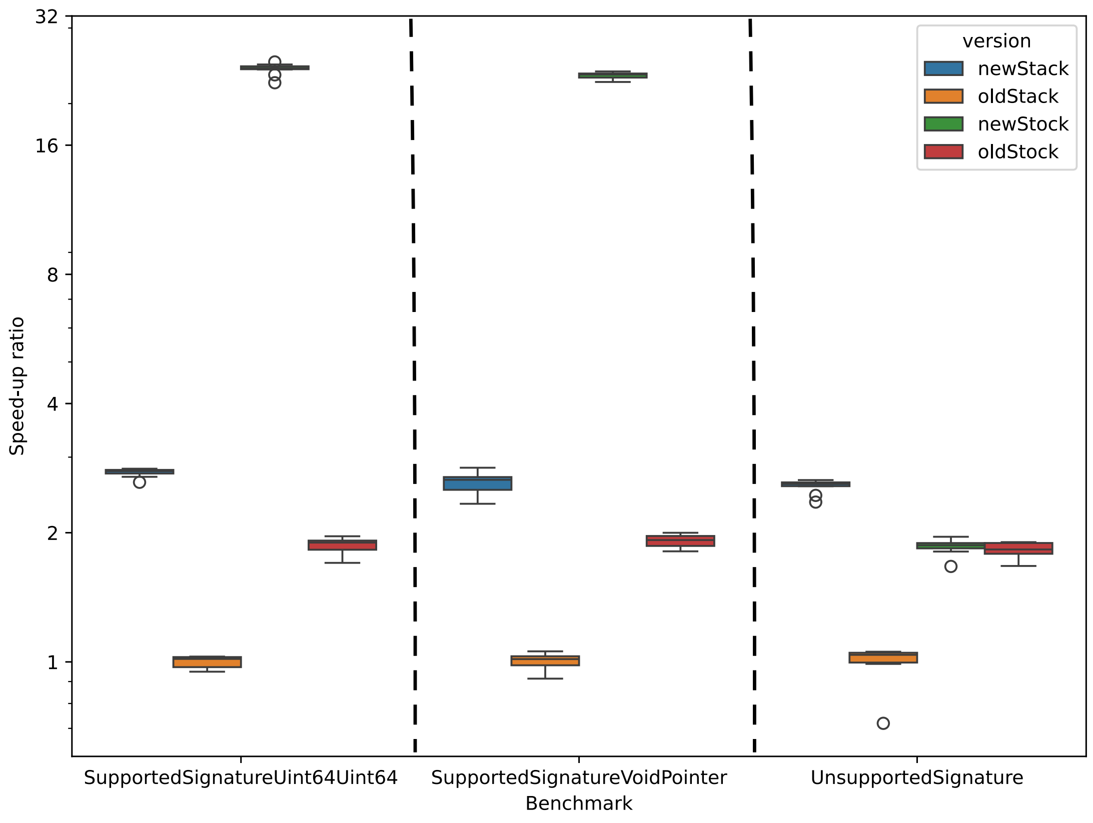

Figure 5: Benchmarks results. Higher is better

Methodology. For each of the microbenchmarks we measured throughput: the number of calls per second. We run each benchmark 100 times doing our best to avoid environment noise. Benchmarks were run on a MacBook Pro with a 2,6GHz 6-Core Intel Core i7, 16 GB 2400MHz DDR4 RAM.

Results. Figure 5 shows the results of our benchmarks. Our results show that with JIT compilation, `bytecodeFFICall` achieves an improvement of 12x over the current implementation when dealing with a function signature supported (optimized). When JIT compilation is not active, the improvement achieved by `bytecodeFFICall` is 3x over the baseline.

For the cases when the function signature is not supported, the figure shows that both `bytecodeFFICall` and the current implementation perform similarly. In the case where JIT compilation is not active, there is a big gap because of the overhead of the work being done at compile time rather than run time. In this case, even though we are not specializing for the function signature, all the checks are done at compile time, so they will done only the first time, in contrast to the current implementation where they will be done each time the primitive gets executed.

The figure also shows how both oldStack and oldStock perform very uniformly across the three benchmarks.

4.2. Maintainability

The number of lines of code added to implement all of this design is 380 so far, including the changes made in the VM as well as in the bytecode compiler. The effort to implement the new design was divided into two main parts: The VM's side and the bytecode compiler's side.

For `bytecodeFFICall` we implemented a new bytecode. This implies having extended the bytecode set of Pharo. For maintenance purposes, this should not be an issue. It would be uncommon to have to modify some of these changes to the bytecode set. Where most of the possible changes/fixes would take place is in the VM.

In the VM, we added support for the interpreter for the new bytecode. Again, the interpreted version of `bytecodeFFICall` should not be something that we expect to get modified a lot. The JIT'ted versions are where we expect future work to happen. As we discussed, the JIT'ted version of the new bytecode is specialized. This means that we have to support only a couple of function signatures. If for some reason in the future, we decide to add support for a specific signature, the only method to modify would be `genBytecodeFFICall` which is where all the JIT'ted implementation of `bytecodeFFICall` resides.

5. Discussion

When planning the new bytecode-based implementation we considered some different options. Figure 6 shows all the possible combinations for implementing FFI calls in the Pharo VM, including the current implementation and `bytecodeFFICall`.

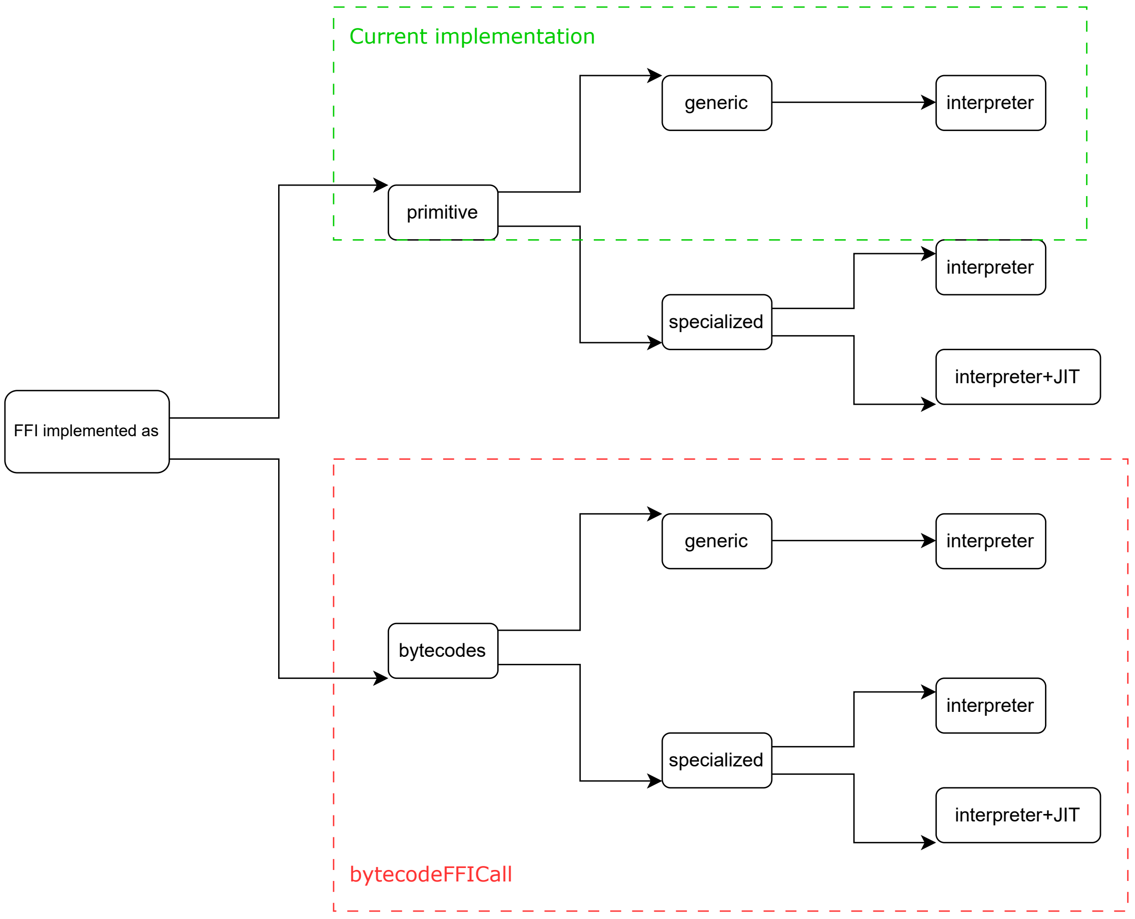

Figure 6: Implementation options for FFI calls

The new bytecode could be implemented in the interpreter in a generic (all kinds of function signatures) or in specialized way (only handle some function signatures). If the bytecode was to be specialized, this would imply:

- Faster interpreter: We know beforehand that the bytecode being executed is for a specific function signature (The bytecode compiler and the UFFI plugin would make sure of that) so there would

be not many checks to do. The disadvantage of this approach is its maintainability. We would end up with a bytecode set much bigger, one bytecode for each function prototype that we want to support.
- Generic fallback: To deal with all the unsupported cases.

If the bytecode was to be generic that would imply having a slower interpreter. This is so because of all the checks we would have to perform at compile time to decide what kind of function signature we are dealing with. In that case, we could specialize it in the JIT. This second option is the one we decided to go with.

6. Related work

This article describes a new design for FFI calls that achieves good performance aiming at a low implementation complexity. The key to our solution is to distinguish between fast and slow execution paths and apply that distinctions to a mixture of bytecode, interpretation, and JIT compilation.

Bytecode design. Bytecode and instruction design has been a matter of discussion for a long time. Smalltalk and descendants have for a long time used execution engines based on bytecode and primitive methods, as described in the blue book [7]. Although implementations diverged over the years, the current design is architected similarly. Our solution introduces a new FFI-call bytecode that can be embedded within a method and benefit from the compilation context and literals in the embedding method. For the slow path, we decided to use a primitive method: our new bytecode can simply fall back to it by compiling a message send and letting the runtime lookup do the rest of the work.

More recently, Béra et al. [8] introduced a new bytecode set into Pharo, namely the Sista bytecode set. The sista bytecode set introduced a redesign of the bytecode set originally inherited from Squeak [9], intended to do bytecode-to-bytecode compiler optimizations [10]. For this purpose, this new bytecode set introduces prefix bytecodes and unsafe bytecodes. Prefix bytecodes (namely extensions in the implementation) annotate existing bytecodes to extend their behavior. Unsafe bytecodes (re-)implement the behavior of existing bytecodes and primitives without safety checks (*e.g.,* type, overflow, and bound checks). Our work extends this existing bytecode set with a new 2-byte bytecode instruction in an unused opcode, not requiring prefixes or unsafe bytecodes. Our new bytecode instruction is so far limited to encoding literals in 4-bit nibbles. However, we envisage using prefix bytecodes to extend the indexable literals.

FFI Implementations. Many projects and programming language implementations acknowledge the importance of integration and interaction with external libraries. We find in the literature FFI implementations for Scheme [11, 12], ML [13], Java [14, 15], Lua [16, 17, 18], R [19] and Smalltalk [20, 4, 21]. Our work investigates the trade offs that can be applied within these implementations.

Instead of realizing a custom implementation, `libffi` [1] persents itself as the de-facto standard to implement foreign function calls in open-source implementations, even accommodating to research projects [22]. For example, `libffi`'s website describes its usage in *e.g.,* the CPython, OpenJDK, Ruby-FFI, Dalvik, and the Racket engines. Our current implementation uses `libffi`for the slow fall-backs, implementing rare function signatures.

Several research projects also considered the modularity and flexibility of the solution, proposing solutions for data-level interoperability [23], modular foreign function interface [24, 25], and even frameworks to configure and specify interoperability patterns [26]. Our solution aims at exploring the performance landscape of FFI from a traditional closed architecture.

The GildaVM extension for the OpenSmalltalkVM redesigned FFI support to implement asynchronous calls through a global interpreter lock and software-simulated interrupts migrating the VM thread [27]. Software-simulated interrupts dynamically migrate the VM thread to allow continuing Pharo execution when an FFI callout takes longer than a threshold. The current implementation in the Pharo VM does not use such an implementation. It implements instead asynchronous calls through queues and worker threads. In this alternative implementation, developers must annotate potentially expensive function calls asynchronous. The work described in this paper extends the current Pharo VM implementation with a new bytecode meant for synchronous FFI calls.

Dealing with low-level concerns and FFI implementation details. In the past, many works have proposed to expose low-level behavior to the Pharo programming language, a feature that has been exploited to implement FFI bindings [21] Salgado et al. proposed lowcode [28], a Pharo extension to support native types and operations. Similarly, Benzo proposes a so-called *Reflective Glue for Low-level Programming*, exposing native machine code to the high-level programming [29]. Chari et al. propose Waterfall, a framework to dynamically generate primitives [30]. While we worked on the standard Pharo VM, we believe that these approaches, orthogonal to our work, could allow further experimentation with the trade-offs between performance and flexibility.

7. Conclusion

The usage of FFI calls in Pharo is crucial. In a typical run of a Pharo image, a large amount of FFI calls get performed. It makes sense for something so important and widely used to try to give it the best performance we can get. In this work, we examined some redesigns we can do to try to gain performance while keeping the effort and maintenance low.

This paper introduced a new design for FFI calls for Pharo. We described some of our key points in designing a new implementation for calling functions that reside outside of Pharo. We explored how a primitive can be redesigned into a bytecode, obtaining more context to work at compile time in the call site opening the opportunity of getting better performance. We found that our new design achieves a boost in performance in microbenchmarks. With little maintenance cost, we were able to achieve a 12x improvement in the JIT'ted specialized cases.

As future work, we want to have a way to easily automatically modify the `genBytecodeFFICall` method to support the most used function signatures but dynamically, this means that we would not need to add support manually statically for some prototypes, the system would do it depending on how much the prototype is used. In some way, it would work like the JIT compiler, in that it would only do its work only if the prototype is *hot* (to notice dynamically if the prototype is being used a lot instead of statically setting them).

References

[1] A. Green, Libffi, 2024. URL: https://sourceware.org/libffi/.

[2] E. Miranda, C. Béra, E. G. Boix, D. Ingalls, Two decades of Smalltalk VM development: live VM development through simulation tools, in: Proceedings of International Workshop on Virtual Machines and Intermediate Languages (VMIL'18), ACM, 2018, pp. 57–66. doi:10.1145/3281287.3281295.

[3] G. Polito, P. Tesone, S. Ducasse, L. Fabresse, T. Rogliano, P. Misse-Chanabier, C. H. Phillips, Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8, in: Proceedings of the 18th international conference on Managed Programming Languages and Runtimes (MPLR '21), Münster, Germany, 2021. URL: https://hal.inria.fr/hal-03332033. doi:10.1145/3475738.3480715.

[4] G. Polito, S. Ducasse, P. Tesone, T. Brunzie, Unified ffi - calling foreign functions from pharo, 2020. URL: http://books.pharo.org/booklet-uffi/.

[5] M. Dias, Blocbenchs, 2024. URL: https://github.com/pharo-graphics/BlocBenchs.

[6] Pharo, Bloc, 2024. URL: https://github.com/pharo-graphics/Bloc.

[7] A. Goldberg, D. Robson, Smalltalk 80: the Language and its Implementation, Addison Wesley, Reading, Mass., 1983. URL: http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

[8] C. Béra, E. Miranda, A bytecode set for adaptive optimizations, in: International Workshop on Smalltalk Technologies (IWST), 2014.

[9] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, a practical Smalltalk written in itself, in: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'97), ACM Press, 1997, pp. 318–326. doi:10.1145/263700.263754.

[10] C. Béra, Sista: a Metacircular Architecture for Runtime Optimisation Persistence, Ph.D. thesis, Université de Lille, 2017.

[11] E. Barzilay, D. Orlovsky, Foreign interface for plt scheme, in: Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming, sn, 2004, pp. 63–74.

[12] F. Klock, The layers of larceny's foreign function interface (2010).

[13] M. Blume, No-longer-foreign: Teaching an ml compiler to speak c "natively", Electronic Notes in Theoretical Computer Science 59 (2001) 36–52.

[14] T. Fast, T. Wall, L. Chen, Java native access (jna), URL: https://github.com/twall/jna (visited on 2013-12-08) (2007).

[15] Y.-H. Tsai, I.-W. Wu, I.-C. Liu, J. J.-J. Shann, Improving performance of jna by using llvm jit compiler, in: 2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS), 2013, pp. 483–488. doi:10.1109/ICIS.2013.6607886.

[16] R. Ierusalimschy, L. H. De Figueiredo, W. Celes, Passing a language through the eye of a needle, Communications of the ACM 54 (2011) 38–43.

[17] M. Pall, Luajit, a just-in-time compiler for lua.(2005), URL http://luajit. org/luajit. html (2005).

[18] G. C. De Paula, R. Ierusalimschy, A foreign function interface for pallene, in: Proceedings of the XXVI Brazilian Symposium on Programming Languages, SBLP '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 32–40. URL: https://doi.org/10.1145/3561320.3561321. doi:10.1145/3561320.3561321.

[19] D. Adler, Foreign Library Interface, The R Journal 4 (2012) 30–40. URL: https://doi.org/10.32614/RJ-2012-004. doi:10.32614/RJ-2012-004.

[20] D. Chisnall, Smalltalk in a c world, in: Proceedings of the International Workshop on Smalltalk Technologies, IWST '12, Association for Computing Machinery, New York, NY, USA, 2012. URL: https://doi.org/10.1145/2448963.2448967. doi:10.1145/2448963.2448967.

[21] C. Bruni, L. Fabresse, S. Ducasse, I. Stasenko, Language-side foreign function interfaces with nativeboost, in: International Workshop on Smalltalk Technologies 2013, 2013.

[22] D. A. Torrance, Foreignfunctions package for macaulay2, 2024. URL: https://arxiv.org/abs/2405.12365. arXiv:2405.12365.

[23] K. Fisher, R. Pucella, J. Reppy, Data-level interoperability, Technical Report, Citeseer, 2000.

[24] J. Yallop, D. Sheets, A. Madhavapeddy, A modular foreign function interface, Science of Computer Programming 164 (2018) 82–97. URL: https://www.sciencedirect.com/science/article/pii/S0167642317300709. doi:https://doi.org/10.1016/j.scico.2017.04.002, special issue of selected papers from FLOPS 2016.

[25] J. Yallop, D. Sheets, A. Madhavapeddy, Declarative foreign function binding through generic programming, in: Functional and Logic Programming: 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings 13, Springer, 2016, pp. 198–214.

[26] K. Fisher, R. Pucella, J. Reppy, A framework for interoperability, Electronic Notes in Theoretical Computer Science 59 (2001) 3–19.

[27] G. Polito, P. Tesone, E. Miranda, D. Simmons, Gildavm: a non-blocking i/o architecture for the cog vm, in: International Workshop on Smalltalk Technologies, Cologne, Germany, 2019. URL: https://hal.archives-ouvertes.fr/hal-02379275.

[28] R. Salgado, S. Ducasse, Lowcode: Extending Pharo with C Types to Improve Performance, in: International Workshop on Smalltalk Technologies IWST'16, Prague, Czech Republic, 2016. doi:10.1145/2991041.2991064.

[29] C. Bruni, L. Fabresse, S. Ducasse, I. Stasenko, Benzo: Reflective glue for low-level programming, in: International Workshop on Smalltalk Technologies 2014, 2014.

[30] G. Chari, D. Garbervetsky, C. Bruni, M. Denker, S. Ducasse, Waterfall: Primitives Generation on the Fly, Technical Report, Inria, 2013.