Language-side Foreign Function Interfaces with NativeBoost

Accepted to IWST 2013

Camillo Bruni Stéphane Ducasse
Igor Stasenko
RMoD, INRIA Lille - Nord Europe, France
http://rmod.lille.inria.fr

Luc Fabresse
Mines Telecom Institute, Mines Douai, France
http://car.mines-douai.fr

Abstract

Foreign-Function-Interfaces (FFIs) are a prerequisite for close system integration of a high-level language. With FFIs the high-level environment interacts with low-level functions allowing for a unique combination of features. This need to interconnect high-level (Objects) and low-level (C functions) has a strong impact on the implementation of a FFI: it has to be flexible and fast at the same time.

We propose NativeBoost a language-side approach to FFIs that only requires minimal changes to the VM. NativeBoost directly creates specific native code at language-side and thus combines the flexibility of a language-side library with the performance of a native plugin.

Categories and Subject Descriptors D.3.3 [*Programming Language*]: Language Constructs and Features; D.3.2 [*Programming Language*]: Language Classifications—Very high-level languages

Keywords system-programming, reflection, managed runtime extensions, dynamic native code generation

1. Introduction

Currently, more and more code is produced and available through reusable libraries such as OpenGL[1] or Cairo[2]. While working on your own projects using dynamic languages, it is crucial to be able to use such existing libraries with little effort. Multiple solutions exist to achieve access to an external library from dynamic languages that are executed on the top of a virtual machine (VM) such as Pharo[3], Lua[4] or Python[5]. Figure 1 depicts four possibilities of dealing with new or external libraries in a high-level language.

Language-side Library. One solution is to re-implement a library completely at language-side (cf. Figure 1.a). Even though this is the most flexible solution, this is often not an option, neither from the technical point of view (performance penalty), nor from the economic point of view (development time and costs).

VM Extension. The second one (1.b) is to do a *VM extension* providing new primitives that the high-level language uses to access the native external library. This solution is generally efficient since the external library may be statically compiled within the VM. However a tight integration into the VM also means more dependencies and a different development environment than the final product at language-side.

VM Plugin. The third solution (1.c) is similar to the previous one but the extension is factored out of the VM as a *plugin*. This solution implies again a lot of low-level development at VM-level that must be done for each external library we want to use. Additionally we have to adapt the plugin for all platforms on which the VM is supposed to run on.

FFI. A higher-level solution is to define *Foreign Function Interfaces* (FFIs) (cf. Figure 1.d). The main advantage of this approach is that once a VM is FFI-enabled, only a language extension (no VM-level code) is needed to provide access to new native libraries. From the portability point of view, only the generic FFI VM-plugin has to be implemented on all platforms.

[1] http://www.opengl.org/

[2] http://cairographics.org/

[3] http://pharo.org/

[4] http://lua.org/

[5] http://python.org/

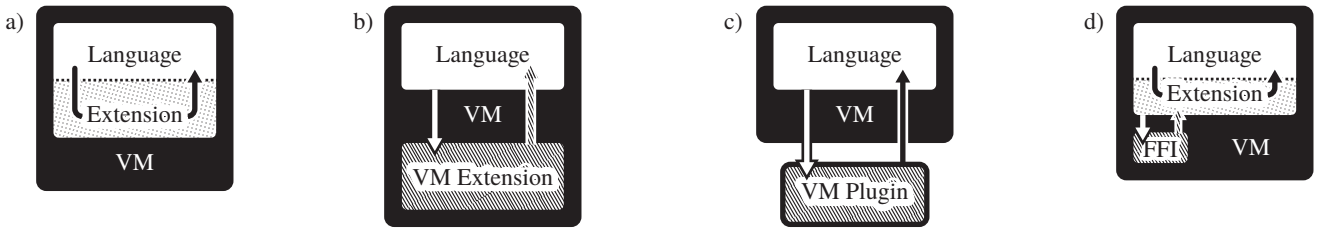

Figure 1: Comparing different extension mechanisms: a) library implemented completely at language-side running on a standard VM, b) language using features from a VM extension, c) language using features from a VM plugin, d) language-side implementation of an extension.

Implementing an FFI library is a challenging task because of its antagonist goals:

- it must be flexible enough to easily bind to external libraries and also express complex foreign calls regarding the memory management or the type conversions (marshalling);
- it must be well integrated with the language (objects, reflection, garbage collector);
- it must be efficient.

Existing FFI libraries of dynamic languages all have different designs and implementations because of the trade-offs they made regarding these goals and challenges. Typical choices are resorting purely to the VM-level and thus sacrificing flexibility. The inverse of this approach exists as well: FFIs can be implemented almost completely at language-side but at a significant performance loss. Both these pitfalls are presented in more detail in Section 3.

This paper presents NativeBoost-FFI[6] an FFI library at language-side for Pharo that supports callouts and callbacks, which we present in Section 2. There are at least two other existing FFI libraries in Pharo worth mentioning: C-FFI and Alien. Nevertheless, they both present shortcomings. C-FFI is fast because it is mostly implemented at VM-level, however it is limited when it comes to do complex calls that involve non-primitive types or when we want to define new data types. On the opposite, Alien FFI is flexible enough to define any kind of data conversion or new types directly at language-side but it is slower than C-FFI because it is mostly implemented at language-side. In essence, NativeBoost-FFI combines the flexibility and extensibility of Alien that uses language-side definition for marshalling and the speed of C-FFI which is implemented at VM-level. The main originalities of NativeBoost-FFI are:

Extensibility. NativeBoost-FFI relies on as few VM primitives as possible (5 primitives), essentially to call native code. Therefore, most of the implementation resides at language-side, even low-level mechanisms. That makes NativeBoost-FFI easily extensible because its implementation can be changed at any time, without needing to update the runtime (VM). It also presents a noticeable philosophical shift, how we want to extend our language in future. A traditional approach is to implement most low-level features at VM-side and provide interfaces to the language-side. But that comes at cost of less flexibility and longer development and release cycles. On the opposite, we argue that extending language features, even low-level ones, should be done at language-side instead. This results in higher flexibility and without incurring high runtime costs which usually happen when using high-level languages such as Smalltalk.

Language-side extension. Accessing a new external library using NativeBoost-FFI involves a reduced amount of work since it is only a matter of writing a language-side extension.

Performance. Despite the fact it is implemented mostly at language-side, NativeBoost-FFI achieves superior performance compared to other FFI implementations running Pharo. This is essentially because it uses automatic and transparent native code generation at language-side for marshalling.

2. NativeBoost-FFI: an Introduction

This section gives an overview of the code that should be written at language-side to enable interactions with external libraries.

2.1 Simple Callout

Listing 1 shows the code of a regular Smalltalk method named `ticksSinceStart` that defines a callout to the `clock` function of the `libc`. NativeBoost imposes no constraint on the class in which such a binding should be defined. However, this method must be annotated with a specific pragma (such as `<primitive:module:>`) which specifies that a native call should be performed using the NativeBoost plugin.

[6] `http://code.google.com/p/nativeboost`

```
ticksSinceStart
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin>
    ^ self
        nbCall: #(uint clock ())
        module: NativeBoost CLibrary
```

Code 1: NativeBoost-FFI example of callout declaration to the `clock` function of the `libc`

The external function call is then described using the `nbCall:module:` message. The first parameter (`#nbCall:`) is an array that describes the signature of C function to callout. Basically, this array contains the description of a C function prototype, which is very close to normal C syntax. The return type is first described (`uint` in this example[7]), then the name of the function (`clock`) and finally the list of parameters (an empty array in this example since `clock` does not have any). The second argument, `#module:` is the module name, its full path or its handle if already loaded, where to look up the given function. This example uses a convenience method of NativeBoost named `CLibrary` to obtain a handle to the standard C library.

2.2 Callout with Parameters

Figure 2 presents the general syntax of NativeBoost-FFI through an example of a callout to the `abs` function of the `libc`. The `abs:` method has one argument named `anInteger` (cf. ❶). This method uses the pragma `<primitive:module:error:>` which indicates that the `#primitiveNativeCall` of the `#NativeBoostPlugin` should be called when this method is executed (cf. ❷). An `errorCode` is returned by this primitive if it fails and the regular Smalltalk code below is executed (cf. ❸). The main difference with the previous example is that the `abs` function takes one integer parameter. In this example, the array `#(uint abs(int anInteger))` passed as argument to `#nbCall:` contains two important information (cf. ❹). First, the types annotations such as the return type (`uint` in both examples) and arguments type (`int` in this example). These types annotations are then used by NativeBoost-FFI to automatically do the marshalling between C and Pharo values as illustrated by the next example. Second, the values to be passed when calling out. In this example, `anInteger` refers to the argument of the `abs` method, meaning that the value of this variable should be passed to the `abs` C function. Finally, this `abs` function is looked up in the `libc` whose an handle is passed in the `module:` parameter (cf. ❺).

[7] The return type of the `clock` function is `clock_t`, but we deliberately used `uint` in this first example for the sake of simplicity even if it is possible to define a constant type in NativeBoost.

```
abs: anInteger ❶
    <primitive: #primitiveNativeCall ❷
     module: #NativeBoostPlugin
     error: errorCode> ❸
    ^ self
        nbCall: #(uint abs(int anInteger)) ❹
        module: NativeBoost CLibrary ❺
```

Figure 2: Example of the general NativeBoost-FFI callout syntax

2.3 Automatic Marshalling of Known Types

Listing 2 shows a callout declaration to the `getenv` function that takes one parameter.

```
getenv: name
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin>

    ^ self
        nbCall: #(String getenv(String name)
        module: NativeBoost CLibrary
```

Code 2: Example of callout to `getenv`

In this example, the NativeBoost type specified for the parameter is `String` instead of `char*` as specified by the standard `libc` documentation. This is on purpose because strings in C are sequences of characters (`char*`) but they must be terminated with the special character: \0. Specifying `String` in the `#nbCall:` array will make NativeBoost to automatically do the arguments conversion from Smalltalk strings to C strings (\0 terminated `char*`). It means that the string passed will be put in an external C `char` array and a \0 character will be added to it at the end. This array will be automatically released after the call returned. This is an example of automatic memory management of NativeBoost that can also be controlled if needed. Obviously, the opposite conversion happens for the returned value and the method returns a Smalltalk String. This example shows that NativeBoost-FFI accepts literals, local and instance variable names in callout declarations and it uses their type annotation to achieve the appropriate data conversion. Table 1 shows the default and automatic data conversions achieved by NativeBoost-FFI.

Listing 3 shows another example to callout the `setenv` function. The return value will be converted to a Smalltalk `Boolean`. The two first parameters are specified as `String` and will be automatically transformed in `char*` with an ending \0 character. The last parameter is `1`, a Smalltalk literal value without any type specification and NativeBoost translates it as an `int` by default.

```
setenv: name value: value
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin>

    ^ self
```

Primitive Type	Smalltalk Type
uint	Integer
int	Integer
String	ByteString
bool	Boolean
float	Float
char	Character
oop	Object

Table 1: Default NativeBoost-FFI mappings between C/primitive types and high-level types. Note that `oop` is not a real primitive type as no marshalling is applied and the raw pointer is directly exposed to Pharo.

```
nbCall: #(Boolean setenv(String name,
                         String value,
                         1)
module: NativeBoost CLibrary
```

Code 3: Example of callout to `setenv`

Another interesting example of automatic marshalling is to define the `abs` method (cf. Figure 2) in the `SmallInteger` class and passing `self` as argument in the callout. In such case, NativeBoost automatically converts `self` (which is a SmallInteger) into an `int`. This list of mapping is not exhaustive and NativeBoost also supports the definition of new data types and new conversions into more complex C types such as structures (cf. Section 4).

2.4 Supporting new types

The strength of language-side FFIs appears when it comes to do callouts with new data types involved. NativeBoost-FFI supports different possibilities to interact with new types.

Declaring structures. For example, the Cairo library[8] provides complex structures such as `cairo_surface_t` and functions to manipulate this data type. Listing 4 shows how to write a regular Smalltalk class to wrap a C structure. NativeBoost only requires a class-side method named `asNBExternalType:` that describes how to marshall this type back and forth from native code. In this example, we use existing marshalling mechanism defined in `NBExternalObjectType` that just copies the structure's pointer and stores it in an instance variable named `handle`.

```
AthensSurface subclass: #AthensCairoSurface
    instanceVariableNames: 'handle'.

AthensCairoSurface class>>asNBExternalType: gen
    "handle iv holds my address (cairo_surface_t)"
    ^ NBExternalObjectType objectClass: self
```

Code 4: Example of C structure wrapping in NativeBoost

Callout with structures. Listing 5 shows a callout definition to the `cairo_image_surface_create` function that returns a `cairo_surface_t*` data type. In this code example, the return type is `AthensCairoSurface` directly (not a pointer). When returning from this callout, NativeBoost creates an instance of `AthensCairoSurface` and the marshalling mechanism stores the returned address in the `handle` instance variable of this object.

```
primImage: aFormat width: aWidth height: aHeight
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin
     error: errorCode>

    ^self nbCall: #(AthensCairoSurface
        cairo_image_surface_create (int aFormat,
                                    int aWidth,
                                    int aHeight) )
```

Code 5: Example of returning a structure by reference

Conversely, passing an `AthensCairoSurface` object as a parameter in a callout makes its pointer stored in its `handle` iv (cf. Listing 6) to be passed. Since the parameter type is `AthensCairoSurface` in the callout definition, NativeBoost also ensures that the passed object is really an instance of this class. If it is not, the callout fails before executing the external function because passing it an address on a non-expected data could lead to unpredicted behavior.

```
primCreate: cairoSurface
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin>

    ^self nbCall: #(
        AthensCairoCanvas cairo_create (
                AthensCairoSurface cairoSurface))
```

Code 6: Example of passing a structure by reference

Accessing structure fields. In NativeBoost, one can directly access the fields of a structure if needed, even if it is not a good practice from the data encapsulation point of view. Nevertheless, it may be mandatory to interact with some native libraries that do not provide all the necessary functions to manipulate the structure. Listing 7 shows an example of a C struct type definition for `cairo_matrix_t`.

```
typedef struct {
    double xx; double yx;
    double xy; double yy;
    double x0; double y0;
```

[8] `http://cairographics.org`

	Memory	Address	Marshalling	Constraint
C-managed struct	C heap	fixed	passed by reference	must be freed
Pharo-managed struct	Object memory	variable	passed by reference or passed by copy	may move costly

Table 2: Wrapping structures possibilities in NativeBoost

```
} cairo_matrix_t;
```

Code 7: Example external type to convert back and forth with the Cairo library

Listing 8 shows that the `NBExternalStructure` of NativeBoost-FFI can be subclassed to define new types such as `AthensCairoMatrix`. The description of the fields (types and names) of this structure is provided by the `fieldsDesc` method on the class side. Given this description, NativeBoost lazily generates field accessors on the instance side using the field names.

```
NBExternalStructure
    variableByteSubclass: #AthensCairoMatrix.

AthensCairoMatrix class>>fieldsDesc
    ^ #(  double sx; double shx;
          double shy; double sy;
          double x; double y;  )
```

Code 8: Example of NativeBoost-FFI definition of an `ExternalStructure`

Listing 9 shows a callout definition to the `cairo_matrix_multiply` function passing `self` as argument with the type `AthensCairoMatrix*`. NativeBoost handles the marshalling of this object to a struct as defined in the `fieldsDesc`.

```
AthensCairoMatrix>>primMultiplyBy: m
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin
     error: errorCode>

"C signature"
"void cairo_matrix_multiply (
                    cairo_matrix_t *result,
                    const cairo_matrix_t *a,
                    const cairo_matrix_t *b );"
    ^self nbCall: #(void   cairo_matrix_multiply
        (AthensCairoMatrix * self,
        AthensCairoMatrix * m ,
        AthensCairoMatrix * self ) )
```

Code 9: Example of callouts using `cairo_matrix_t`

Memory management of structures. Table 2 shows a comparison between C-managed and Pharo-managed structures. The first ones are allocated in the C heap. Their addresses are fixed and they are passed by reference during a callout. But the programmer must free them by hand when they are not needed. The second ones are allocated in the Pharo object-memory. Their addresses are variable since their enclosing object may be moved by the garbage collector. They can either passed by copy which is costly or by reference. Passing a reference may lead to problems is the C function stores the address and try to access it later on since the address may changed.

2.5 Callbacks

NativeBoost supports callbacks from native code. This means it is possible for a C-function to call back into the Pharo runtime and activate code. We will use the simple `qsort` C-function to illustrate this use-case. `qsort` sorts a given array according to the results of a compare function. Instead of using a C-function to compare the elements we will use a callback to invoke a Pharo block which will compare the two arguments.

```
bytes := #[ 120 12 1 15 ].
callback := QSortCallback on: [ :a :b |
                (a byteAt: 0) - (b byteAt: 0) ].

self ffiQSort: bytes
     length: bytes size
     compareWith: callback
```

Code 10: Example of callout passing a callback for `qsort`

Code 10 shows the primary Pharo method for invoking `qsort` with a `QSortCallback` instance for the compare function. In this example `qsort` will invoke run the Pharo code inside the callback block to compare the elements in the `bytes` array.

To define a callback in NativeBoost we have to create a specific subclasses for each callback with different argument types.

```
NBFFICallback
    subclass: #QSortCallback.

NBFFICallback class>>signature
    ^#(int (NBExternalAddress a, NBExternalAddress b))
```

Code 11: Example of callback definition

Code 11 shows `QSortCallback` which takes two generic external addresses as arguments. These are the argument types that are being passed to the sort block in Example 10.

```
ffiQSort: base len: size compare: qsortCallback
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin>

    "C qsort signature"
    "void qsort(
        void *base,
        size_t nel,
        size_t width,
        int (*compar)(const void *, const void *));"

    ^ self
        options: #( optMayGC )
        nbCall: #(void qsort (
                    NBExternalAddress array,
                    ulong size,
                    1, "sizeof an element"
                    QSortCallback qsortCallback))
        module: NativeBoost CLibrary
```

Code 12: Example of callout passing a callback

The last missing piece in this example is the callout definition shown in Code 12. The NativeBoost callout specifies the callback arguments by using `QSortCallback`.

Callback lifetime. Each time a new callback is instantiated it reserves a small amount of external memory which is freed once the callback is no longer used. This is done automatically using object finalization hooks..

2.6 Overview of NativeBoost-FFI Internals

This section provides an overview of the internal machinery of NativeBoost-FFI though it is not mandatory to know it in order to use it as demonstrated by previous examples.

General Architecture. Figure 3 describes the general architecture of NativeBoost. Most code resides at language-side, nevertheless some generic extensions (primitives) to the VM are necessary to activate native code. At language-side, callouts are declared with NativeBoost-FFI which processes them and dynamically generates x86 native code using the `AsmJit` library. This native code is responsible of the marshalling and calling the external function. NativeBoost then uses a primitive to activate this native code.

Callout propagation. Figure 4 shows a comparison of the resolution of a FFI call both in NativeBoost-FFI and a plugin-based FFI. At step 1, a FFI call is emitted. The NativeBoost-FFI call is mostly processed at language-side and it is only during step 4 that a primitive is called and the VM effectively does the external call by executing the native code. On the opposite, a plugin-based FFI call already crossed the low-level frontier in step 2 resulting that part of the

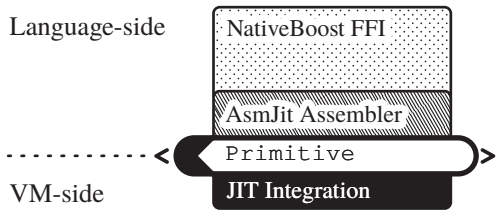

Figure 3: NativeBoost main components that major part of the code resides at language-side.

type conversion process (marshalling) is already done in the VM code. In NativeBoost-FFI, doing most of the FFI call processing at language-side makes easier to keep control, redefine or adapt it if needed.

3. NativeBoost-FFI Evaluation

In this section we compare NativeBoost with other FFI implementations.

Alien FFI: An FFI implementation for Squeak/Pharo that focuses on the language-side. All marshalling happens transparently at language-side.

C-FFI: A C based FFI implementation for Squeak/Pharo that performs all marshalling operations at VM-side.

LuaJIT: A fast Lua implementation that has a close FFI integration with JIT interaction.

Choice of FFI Implementations. To evaluate NativeBoost we explicitly target FFI implementations running on the same platform, hence we can rule out additional performance differences. Alien and C-FFI run in the same Pharo image as NativeBoost allowing a much closer comparison.

Alien FFI is implemented almost completely at language-side, much like NativeBoost. However, as the following benchmarks will stress, it also suffers from performance loss.

On the other end there is C-FFI which is faster than Alien but by far not as flexible. For instance only primitive types are handled directly.

As the third implementation we chose Lua. Lua is widely used as scripting language in game development. Hence much care has been taken to closely integrate Lua into C and C++ environments. LuaJIT integrates an FFI library that generates the native code for marshalling and directly inlines C functions callout in the JIT-compiled code.

Evaluation Procedure. To compare the different FFI approaches we measure 100 times the accumulative time spent to perform $1'000'000$ callouts of the given function. From the 100 probes we show the average and the standard deviation for a 68% confidence

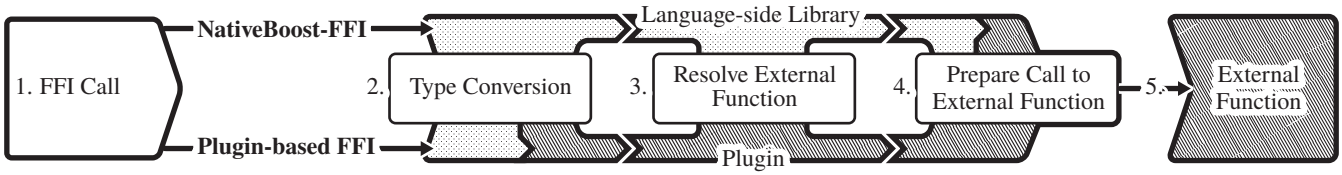

Figure 4: Comparison of FFI calls propagation in NativeBoost-FFI and a typical VM plugin-based implementation. NativeBoost resorts to VM-level only for the native-code activation, whereas typical implementations cross this barrier much earlier.

interval in a gaussian distribution. To exclude the calling and loop overhead we subtract from each evaluation the time spent in the same setup, but without the FFI call. The final deviation displayed is the arithmetic average of the measured deviation of the base and the callout measurement.

The three Smalltalk FFI solutions (NativeBoost, Alien, C-FFI) are evaluated on the very same Pharo 1.4 (version 14458) image on a Pharo VM (version of May 5. 2013). For the Lua benchmarks we use LuaJIT 2.0.1. The benchmarks are performed under the constant conditions on a MacBook Pro. Even though a standalone machine could improve the performance we are only interested in the relative performance of each implementation.

Choice of Callouts. We chose a set of representative C functions to stress different aspects of an FFI implementation. We start with simple functions that require little marshalling efforts and thus mainly focus on the activation performance and callout overhead. Later we measure more complex C functions that return complex types and thus stress the marshalling infrastructure.

3.1 Callout Overhead

The first set of FFI callouts show mainly the overhead of the FFI infrastructure to perform the callout.

For the first FFI evaluation we measure the execution time for a `clock()` callout. The C function takes no argument and returns an integer thus guaranteeing a minimal overhead for marshalling and performing the callout.

	Call Time	Relative Time
NativeBoost	`492.13` ± `0.73` ms	1.0×
Alien	`606.6` ± `1.9` ms	≈ 1.2×
C-FFI	`541.77` ± `0.88` ms	≈ 1.1×
LuaJIT	`343.0` ± `1.2` ms	≈ 0.7×

Table 3: Speed comparison of an `uint clock(void)` FFI call (see Code 1).

`abs` is a about the same complexity as the `clock` function, however accepting a single integer as argument.

	Call Time	Relative Time
NativeBoost	`65.34` ± `0.23` ms	1.00×
Alien	`175.77` ± `0.31` ms	≈ 2.69×
C-FFI	`148.77` ± `0.21` ms	≈ 2.27×
LuaJIT[9]	`2.035` ± `0.015` ms	≈ 0.03×

Table 4: Speed comparison of an `int abs(int i)` FFI call (see Figure 2).

Evaluation. For measuring the calling overhead we chose the `abs` FFI callout. This C function is completed in a couple of instructions which in comparison to the conversion and activation effort of the FFI callout is negligible. In Table 4 we see that NativeBoost is at least a factor two faster than the other Smalltalk implementation. Yet LuaJIT outperform NativeBoost by an impressive factor 30. LuaJIT has a really close integration with the JIT and this is what makes the impressive FFI callout results possible.

3.2 Marshalling Overhead for Primitive Types

The third example calls `getenv('PWD')` expecting a string as result: the path of the current working directory. Both argument and result have to be converted from high-level strings to C-level zero-terminated strings.

	Call Time	Relative Time
NativeBoost	`105.29` ± `0.24` ms	1.0×
Alien	`1058.7` ± `2.0` ms	≈ 10.1×
C-FFI	`282.94` ± `0.24` ms	≈ 2.7×
LuaJIT[10]	`97.3` ± `5.1` ms	≈ 0.9×

Table 5: Speed comparison of an `char * getenv(char *name)` FFI call (see Code 2).

As a last evaluation of simple C functions with NativeBoost, we call `printf` with a string and two integers as argument. The marshalling overhead is less than for

[9] Downsampled from increased loop size by a factor 100 to guarantee accuracy.

[10] Downsampled from increased loop size by a factor 10 to guarantee accuracy.

the previous `getenv` example. However, `printf` is a more complex C function which requires more time to complete: it has to parse the format string, format the given arguments and pipe the results to standard out. Hence the relative overhead of an FFI call is reduced.

	Call Time	Relative Time
NativeBoost	371.03 ± 0.51 ms	1×
Alien	1412.37 ± 0.79 ms	≈ 3.8×
C-FFI	605.02 ± 0.23 ms	≈ 1.6×
LuaJIT	202.4 ± 2.1 ms	≈ 0.6×

Table 6: Speed comparison of an `int printf(char *name, int num1, int num2)` FFI call

Evaluation. Table 3 and Table 4 call C functions that return integers for which the conversion overhead is comparably low. However we see that Alien compares worse in the case of more complex Strings. Table 5 and Table 6 show this behavior. For the `getenv` a comparably long string is returned which causes a factor 10 conversion overhead for Alien.

3.3 Using Complex Structures

To evaluate the impact of marshalling complex types, we measure the execution time for a callout to `cairo_matrix_multiply`. In all cases, the allocation time of the structs is not included in the measurement nor their field assignments. Table 7 shows the results.

	Call Time	Relative Time
NativeBoost	79.00 ± 0.27 ms	1.0×
Alien	753.82 ± 0.51 ms	≈ 9.5×
C-FFI	380.8 ± 2.7 ms	≈ 3.6×
LuaJIT	5.66 ± 0.15 ms	≈ 0.07×

Table 7: Speed comparison of an `cairo_matrix_multiply` FFI call (cf. Listing 9)

Evaluation. In Table 7 shows that NativeBoost outperforms the two other Smalltalk implementations.

3.4 Callbacks

Table 8 shows a comparison of `qsort` callouts passing callbacks. Callbacks are usually much more slower than callouts.

	Call Time	Rel. Time
NativeBoost	2300.0 ± 1.1 ms	1.0×
Alien	600.83 ± 0.35 ms	≈ 0.26×
C-FFI	NA	NA
LuaJIT	46.13 ± 0.62 ms	≈ 0.02×
NativeBoost with Native Callbacks	4.98 ± 0.21 ms	≈ 0.002×

Table 8: Speed comparison of a `qsort` FFI call (cf. Listing 10)

Evaluation. The results show that NativeBoost callbacks are currently slower than Alien's ones. This is because Alien relies on specific VM support for callbacks making their activation faster (context creation and stack pages integration). On the opposite, NativeBoost currently uses small support from the VM side and even do part of the work at image side. This `qsort` demonstrates the worst case because it implies a lot of activations of the callback. For each of these calls, NativeBoost creates a context and make the VM switch to it. To really demonstrate that these context switches are the bottleneck, Table 8 also shows the result of doing the same benchmark in NativeBoost but using a native callback i.e. containing native code. We do not argue here that callbacks should be implemented in native code but that NativeBoost support for callback can be optimized to reach Alien's performance at least.

4. NativeBoost-FFI Implementation Details

The following subsections will first focus on the high-level, language-side aspects of NativeBoost, such as native code generation and marshalling. As a second part we describe implementation details of the low-level extensions, such as the NativeBoost primitives and the JIT interaction.

4.1 Generating Native Code

In NativeBoost all code generation happens transparently at language-side. The various examples shown in Section 2 show how an FFI callout is defined in a standard method. Upon first activation the NativeBoost primitive will fail and by default continues to evaluate the following method body. This is the point where NativeBoost generates native code and attaches it to the compiled method. NativeBoost then reflectively resends the original message with the original arguments (for instance `abs:` in the example Figure 2). On the second activation, the native code is present and thus the primitive will no fail but run the native code. Section 4.2.1 will give more internal details about the code activation and triggering of code generation.

4.1.1 Generating Assembler Instructions

Figure 3 shows that NativeBoost relies on AsmJit[11], a language-side assembler. AsmJit emerged from an existing C++ implementation[12] and currently supports the x86 instruction set.

In fact it is even possible to inline custom assembler instructions in Pharo when using NativeBoost. This way it is possible to meet critical performance requirements. Typically Smalltalk does not excel at algorithmic code since such code does not benefit from dynamic message sends.

4.1.2 Reflective Symbiosis

NativeBoost lives in symbiosis with the Pharo programming environment. As shown in the examples in Section 2 and in more detail in Figure 2 NativeBoost detects which method arguments correspond to which argument in the FFI callout. To achieve this, NativeBoost inspects the activation context when generating native code. Through reflective access to the execution context we can retrieve the method's source code and thus the argument names and positions.

4.1.3 Memory Management

NativeBoost supports external heap management with explicit allocation and freeing of memory regions. There are interfaces for `allocate` and `free` as well as for `memcopy`:

```
memory := NativeBoost allocate: 4.
bytes  := #[1 2 3 4].
"Fill the external memory"
NativeBoost memCopy: bytes to: memory size: 4.

"FFI call to fill the external object"
self fillExternalMemory: memory.

"Copy back bytes from the external object"
NativeBoost memCopy: memory to: bytes size: 4.
NativeBoost free: memory.
```

Code 13: Example of external heap management in NativeBoost

Using the external heap management it is possible to prepare binary blobs and structures for FFI calls.

In the previous example Code 13 the `memory` variable holds a wrapper for the static address of the allocated memory. Hence accessing it from low-level code is straight forward. However in certain situations it is required to access a high-level object from assembler. Pharo has a moving garbage collector which means that you can not refer directly to a high-level object by a fixed address.

[11] `http://smalltalkhub.com/#!/~Pharo/AsmJit`

[12] `https://code.google.com/p/asmjit/`

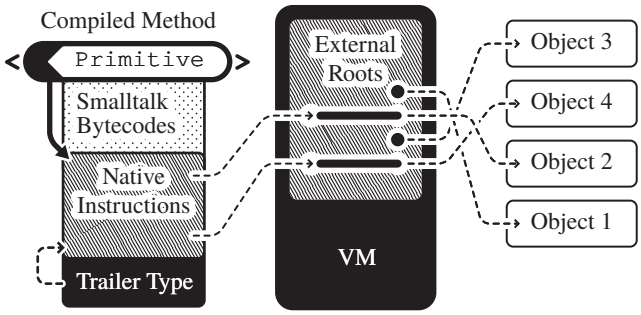

Figure 5: Pointers in a CompiledMethod to objects registered as external roots are pinpointed at fixed offset in global VM-level object.

To deal with this problem the VM has a special array at a known address that contains pointers to high-level objects. The garbage collector keeps this external roots array up to date. Hence it is possible to statically refer to a Pharo object using a double indirection over the external roots. Figure 5 visualizes how native code directly accesses Pharo objects through this indirection.

4.2 Activating Native Code

In this section we present the VM-level interaction of NativeBoost. Even though NativeBoost handles most tasks directly at language-side it requires certain changes on VM level:

- executable memory,
- activation primitives for native code.

Since NativeBoost manages native code at language-side there is no special structure or memory region where native code is stored. Native instructions are appended to compiled methods which reside on the heap. Hence the heap has to be executable in order to jump to the native instructions.

4.2.1 The NativeBoost activation Primitive

In Section 4.1 we explained how NativeBoost creates FFI callouts at language-side. However, so far we left out the part on how the generated native code is activated.

The examples in Section 2, especially Figure 2 show that each NativeBoost FFI callout requires a special primitive. Figure 6 shows how a NativeBoost method is activated.

- In the first step (cf. ❶) the NativeBoost callout primitive is activated. The primitive checks if the compiled method actually contains native code.
- On the first activation there is no native code available yet. Hence the primitive will fail and the Smalltalk body (cf. ❷) of the NativeBoost method

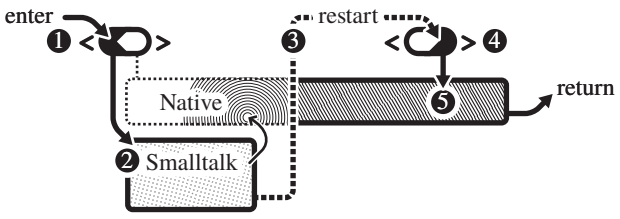

Figure 6: Native code activation. The first call triggers the code generation. Then the method is restarted and the native code executed.

gets evaluated. This is where NativeBoost prepares the native code for the FFI callout.

- After installing the native code in the method trailer, the NativeBoost method is reactivated with the original arguments (cf. ❸).
- Again we end up in the NativeBoost activation primitive (cf. ❹). However, this time there is native code (cf. ❺) available and thus the primitive jumps to the native code instead.

5. Related Work

Typical Smalltalk system are isolated from the low-level world and provide only limited interoperability with C libraries. However there are notable exceptions: Étoilé and Smalltalk/X.

Chisnall presents the Pragmatic Smalltalk Compiler [**?**], part of the Étoilé project, which focuses on close interaction with the C world. The main goal of this work is to reuse existing libraries and thus reduce duplicated effort. The author highlights the expressiveness of Smalltalk to support this goal. In this Smalltalk implementation multiple languages can be mixed efficiently. It is possible to mix Objective-C, Smalltalk code. All these operations can be performed dynamically at runtime. Unlike our approach, Étoilé aims at a complete new style of runtime environment without a VM. Compared to that, NativeBoost is a very lightweight solution.

Other dynamic high-level languages such as Lua leverage FFI performance by using a close interaction with the JIT. LuaJIT [**?**] for instance is an efficient Lua implementation that inlines FFI calls directly into the JIT compiled code. Similar to NativeBoost this allows one to minimize the constant overhead by generating custom-made native code. The LuaJIT runtime is mainly written in C which has clearly different semantics than Lua itself.

On a more abstract level, high-level low-level programming [**?**] encourage to use high-level languages for system programming. Frampton et al. present a low-level framework which is used as system interface for Jikes, an experimental Java VM. However their approach focuses on a static solution. Methods have to be annotated to use low-level functionality. Additionally the strong separation between low-level code and runtime does not allow for reflective extensions of the runtime. Finally, they do not support the execution and not even generation of custom assembly code on the fly.

QUICKTALK [**?**] follows a similar approach as NativeBoost. However Ballard et al. focus mostly on the development of a complex compiler for a new Smalltalk dialect. Using type annotations QUICKTALK allows for statically typing methods. By inlining methods and eliminating the bytecode dispatch overhead by generating native code QUICKTALK outperforms interpreted bytecode methods. Compared to Waterfall, QUICKTALK does not allow to leave the language-side environment and interact closely with the VM.

Kell and Irwin [**?**] take a different look at interacting with external libraries. They advocate a Python VM that allows for dynamically shared objects with external libraries. It uses the low-level DWARF debugging information present in the external libraries to gather enough metadata to automatically generate FFIs.

6. Future Work

Even though NativeBoost shows good overall performance when it comes to callbacks it does not keep up with other Smalltalk-based solutions. In the current development phase not much attention was payed to callback performance as it is not a common use case for FFI callouts. Fast callbacks require close interaction and specific modifications at VM-level. However, initially NativeBoost kept the modifications to the VM at a minimum. We assume that we can reach the same performance as Alien relying on the same low-level implementation.

As a second issue we would like to address the callout overhead by using an already existing JIT integration of NativeBoost. Currently the VM has to leave from JIT-mode to standard interpretation mode when it activates an NativeBoost method. This context switch introduces an unnecessary overhead for an FFI callout. A current prototype directly inlines the native code of a NativeBoost method in the JIT. Hence the cost for the context switch plus the cost of activating the NativeBoost callout primitive can be avoided.

7. Conclusion

In this paper we presented NativeBoost a novel approach to foreign function interfaces. Our approach relies only on a very generic extension of the VM to allow for language-side code to directly call native instructions.

Using a in depth evaluation of NativeBoost comparing against two other Smalltalk FFI implementations

and LuaJIT we showed in Section 3 that our language-side approach is competitive. NativeBoost reduces the callout overhead by more than a factor two compared to the two closest Smalltalk solutions.

Compared to LuaJIT there is still space for improvements. We measured a factor 30 lower calling overhead due to a close JIT integration. However for typical FFI calls the absolute time difference between NativeBoost and Lua is roughly 30%. With a partial solution ready to integrate NativeBoost closer with the JIT we expect to come close to Lua's performance.

Furthermore we showed that NativeBoost essentially combines VM-level performance with language-side flexibility when it comes to marshal complex types. New structures are defined practically at language-side and conversion optimizations are added transparently.

Acknowledgments

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219 and the MEALS Marie Curie Actions program FP7-PEOPLE-2011-IRSES.