Accepted to IWST2016

Lowcode:

Extending Pharo with C Types to Improve Performance

Ronie Salgado
Pleiad Lab, DCC, University of Chile
roniesalg@gmail.com

Stéphane Ducasse
RMoD, INRIA Lille
stephane.ducasse@inria.fr

Abstract

The highly dynamic nature of Smalltalk provides a high degree of flexibility, but at the expense of performance. On the other hand, static system programming languages such as C are really fast, but less flexible and harder to use than Smalltalk. Our hypothesis is that by mixing the concepts of these two worlds in a single programming environment, we are able to have improved performance and a high level of flexibility at the same time.

In this work we extend Pharo by adding a type system that provides the native data types present in C along with the dynamic object type. We extend Pharo compiler and virtual machine to use our type system by add custom bytecode instructions for dealing with native data. With our approach we obtain a performance improvement on average between two and five times faster for numerical computations using single precision floating point arithmetic in Smalltalk.

1. Two worlds of programming languages

Nowadays most of the programming languages can be placed in two very different worlds: the convenient to use managed high-level world, and the hard to use but very fast world of system programming languages, with the king known as C. Smalltalk is a very dynamic programming language that belongs to the managed world.

In this language everything is an object, and objects only communicate via message passing. These objects live in a world managed by a virtual machine that provides the services of managing object lifetime, dispatching messages between objects by executing compiled methods that implement handlers for messages. Compiled methods contain portable bytecode instructions that tell the virtual machine what to do. The virtual machine executes a compiled method directly by interpreting each bytecode, or indirectly by recompiling bytecode instructions into platform specific machine code, via a just-in-time compiler (Béra et al. 2016), and then letting the CPU execute these instructions.

The C programming language in the other extreme belongs to the system programming world. In this language everything is compiled directly into machine instructions and the data types most of the time have direct correlation with data types that are natively supported by the CPU. Unlike Smalltalk, since C has a direct correspondence with the underlying machine, it is a really fast language. However, its downside is that programming in C is very error prone due to the burden of manual memory management placed all of the time on the programmer.

Smalltalk can be an order of magnitude slower than C, and 90% of the time of a program is spent in only a 10% of the code. Therefore it makes perfect sense to use C only for the performance critical sections of a software, and for the rest use only Smalltalk. For this very same reason, Smalltalk provides a mechanism known as *primitives* (Goldberg and Robson 1983, 1989). Primitives in fact serve two different purposes: first, accessing elementary functionalities provided by the virtual machines and that cannot be expressed in Smalltalk such as allocating new objects, swapping pointers), second accelerating extremely used behavior such as point creation.

Primitive methods are implemented by the virtual machine by escaping from the Smalltalk world and calling a C function. The main problem of primitives is that to implement them, it is necessary to get exposed to low-level language structures and conventions. Pharo virtual machine (inheriting from the Squeak virtual machine architecture) supports named primitives which can be compiled separately from the core Virtual Machine (Guzdial and Rose 2001). Traditional primitives add a maintenance problem of having to maintain two separate code bases for a single project: the project itself, and the custom virtual machine extended with the primitives used by the project (Bruni et al. 2013b).

In the particular case of the Pharo virtual machine, there is also the problem that primitive methods can not perform message sends. This means that primitive are usually not completely flexible as normal methods.

The same problem of hard to write primitives occurs when having to support FFI. NativeBoost was a solution to support the expression of FFI at the Pharo level by providing a full assembler at the Smalltalk level (Bruni et al. 2013b): 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. In their works, Bruni (Bruni et al. 2014b; Dias 2014) and Chari (Chari et al. 2015) acknowledge the definition of more open and reflective virtual machines exposing parts of the virtual machine that can be changed from within itself at runtime.

In this paper we propose a method to reduce the gap between these two worlds by extending Pharo with a type system and its virtual machine with a custom bytecode set. The type system adds the native data types present in C. We extended the compiler to use this type information to generate custom low-level bytecode instructions that are interpreted or compiled into fast machine code instructions by the Just-in-Time compiler available in the virtual machine.

We structure our paper by posing some hypotheses about what we want to achieve by using our type system and custom bytecode instructions. We first begin by extending the bytecode instructions to be able to do the same operations that can be done in C. We proceed by designing a type system to make a symbiosis between the high-level and the low-level programming worlds. With the new bytecode instructions and a type system in hand, we describe how we extended the compiler to perform type checking, type inference and to generate the low-level bytecode instructions. Once we have built all of our infrastructure, we proceed to evaluate this infrastructure by designing, running and discussing some benchmarks. We conclude our work by discussing future work to do and some related works in the literature.

2. Hypothesis and methodology

Our hypothesis is that by extending the bytecode set of the Pharo virtual machine and Pharo itself, we can get the same kind of performance for the cases where primitives are used, and the same time we can avoid most of their problems. Since the virtual machine usually performs just in time compilation of the bytecodes into native machine code, the performance improvements can be substantial.

To test our hypotheses:

1. We extend the bytecode set of the Pharo virtual machine, in its two flavors: the interpreter and the Just-in-Time compiler version. The extended bytecode set provides instructions that allows the VM to operate directly with primitive data types, pointers and the memory.
2. Then we add a way to define a class that represents a C like structure inside the Pharo image.
3. Our next step is to extend the semantic analyzer of the compiler to perform type checking and annotate the AST with type information.
4. By extending the bytecode translator of the compiler, we emit our extended bytecode when types annotated in the AST allow us to use them.

Our final stage is to run some simple benchmarks with basic matrix and vector operations that are commonly used on 3D graphics. Our benchmarks show us a speed improvement between two times and five times faster than normal Smalltalk methods when using our extended compiler with the proper type annotations and the Just-in-Time compiler version of the virtual machine.

3. Extending the VM with new Bytecodes

The extended bytecode set that we call Lowcode is a stack-based instruction set. Our first step to design and implement Lowcode is the creation of a formal specification of the Lowcode instruction set which is used to document and implement the bytecode instruction set. The Lowcode specification describes the instruction arguments that are encoded in the compiled method instruction stream: it describes the arguments that are popped from the stack by the instruction, along with the results that are pushed into stack. All of these parameters are described with an associated type that can be one of: *oop* (object pointer), *int32*, *int64*, *pointer*, *float32*, *float64*. These types are enough to describe most of the data types that are supported directly by the virtual machine or by the CPU. As a example, here we have the specification for the *float32Add* instruction:

```
<instruction opcode="1037" mnemonic="float32Add">
    <name>Float32 addition</name>
    <description>
        This instruction performs the addition of two
        single precision floating point numbers.
    </description>
    <arguments />
    <stack-arguments>
        <float32 name="first" />
        <float32 name="second" />
    </stack-arguments>
    <stack-results>
        <float32 name="result" aliased="true" />
    </stack-results>
    <semantic language="Smalltalk/Cog">
        self AddRs: second Rs: first.
        self ssPushNativeRegisterSingleFloat: first.
    </semantic>
    <semantic language="Smalltalk/StackInterpreter">
        result := first + second.
    </semantic>
</instruction>
```

Using Call Inline Primitive. For being able to use Lowcode and normal Pharo instructions, we take advantage of a single bytecode instruction known as *call inline primitive* which was introduced in a new extensible bytecode set (Béra and Miranda 2014). The call inline primitive instruction is intended to implement primitive operations that omit the checking performed in normal Smalltalk primitives, and are invoked directly, without sending messages, resulting in much simpler, shorter and faster code sequences than for normal primitives. The call inline primitive bytecode is followed by the number of the inline primitive that has to call. For practical purposes, inline primitives are actually a mechanism for extending the bytecode instruction set. By using this mechanism for encoding our extended instructions, we can reuse the normal instructions such as message send, and we can mix them freely with our custom instructions.

Instructions. The Lowcode instruction set is composed by instructions encoding several kinds of operations:

- conversions between objects and native data types,
- local method frame allocation for storing local variables with native data,
- manipulation of memory pointers,
- load and store of native values from memory,
- direct access to the instance variables present in any object,
- pointers into the first field or the first indexable field of an object. These instructions for getting a pointer into an object field are specially useful for manipulating ByteArray or any object with a byte, or non-object data layout.

Lowcode Stack. Since the virtual machine uses the object stack for tracing objects that are garbage collected, it is not possible to put the native data that is operated by the Lowcode instructions into the same stack that is used for Smalltalk object pointers. For this reason, the Lowcode instructions operate in a separate stack for native data which will not be traversed by the garbage collector. Some Lowcode instructions such as *oopToFloat32* operates in both stacks. This particular instruction receives an object pointer in the object stack, which is converted into a native *float32* value that is pushed into the native data stack.

Most of the Lowcode instructions do not perform type checks of their parameters. Type checking of parameters is left to the user by using traditional Smalltalk instructions such as sending a testing message, or by using some special Lowcode instructions to query about the layout of an object. This simplifies the implementation of the Lowcode instructions themselves and improve performance by removing redundant type checking.

Taking advantage of instruction specification. Lowcode instruction specification is also used to implement the instructions in both modes (interpreted and Just-in-Time) of the Virtual Machine execution. In the case of the interpreter, by taking advantage of the instruction signature specification, we generate automatically the code for fetching arguments from the stacks or encoded in-line in the instruction stream. This signature is also used to generate the interpreter code for pushing instruction result back into the stacks. By generating automatically this bookkeeping code, the code that for an instruction that has to be written manually is reduced considerably. In many cases, the actual implementation of a Lowcode instruction written manually by a programmer is only a single line of code for the interpreter version of the virtual machine.

For the JIT variation, Lowcode instructions are implemented in a similar fashion to the way they were implemented in the interpreter. Instead of generating code for pushing or poping values from one or two stacks, in this case we are generating code for doing register allocation via two simulation stacks. The just in time compiler uses a "simulation stack"[1] to keep track of how parameters are passed between instructions, and to keep track of which native CPU registers are going to hold a parameter for an instruction and in what moment during the execution of a method these values are present in a specific register. In other words, the simulation stack used by the virtual machine is a greedy scheme for doing register allocation.

Similar to the implementation for the interpreted execution, the implementations of these instructions in the just in time-compiler version are short. Most of them are only one or two lines of code, since the register allocation code is generated automatically. The implementation assume that the instructions are always operating with registers, which helps simplify their implementation and fits well with the automatically generated allocation of registers. Lowcode instructions only operate directly with constants, when some of their parameters are specified as constants that are encoded with the instruction.

Being able to extend easily the virtual machine with new bytecode instructions is only one part of the problem. The other part is actually generating Smalltalk compiled methods in the image that actually use these instructions. The next sections deal exclusively with such part of Lowcode.

4. Designing a Type System

For starting to use the Lowcode instructions our first task was doing a bytecode assembler for generating some compiled methods using these instructions. This approach was useful for making sure the implementation of the instructions was correct, but it is far for being user friendly.

Since writing bytecode manually is a tedious task, our next objective was to extend Pharo to generate these instructions when it makes sense to use them. Lowcode instructions are strongly related with the notion of primitive data types. Because Smalltalk does not have a static type system, we need to extend Smalltalk with the creation of a type system.

For the creation of our type system we made two important observations:

[1] The simulation stack is used by the JIT to model the state of compilation as it compiles the stack-oriented byte code to machine code. The simulation stack holds values modelling arguments and local variables, which may have been assigned to registers, parameters pushed on to the stack by push byte codes, and values modelling the results of computations. It is the key structure that allows the JIT to compile stack-oriented byte code into register-oriented machine code

- In Smalltalk everything is an object.
- Lowcode instructions add the ability to operate with the data types present in the C programming language.

From these two observations, our type system needs at least to support all of the data types present in the C programming language, with the inclusion of a single different type: *object*. The *object* type represents a normal Smalltalk object, and it also makes sense to let *object* be the default type of everything. Variables or values should have type different than object only when there is some kind of type annotation or information that tells their type.

The data types present in C (Kernighan and Ritchie 1978) fit into two big families: primitives data types and derived data types. Primitive data types represent small data values that are usually supported directly by the CPU *bool*, *int* or *double*. Derived types are built upon the primitive data types, and these types are of three kinds: pointers, structures and unions.

- A pointer is a memory address that points to a particular C type, that may be any C type, including void, the type for "opaque" pointers. This is why a pointer type is a derived type.
- A structure type is a sequence of different types that are laid linearly in memory. An union type is similar to a structure, but their values are placed at the same memory location.
- Unions are used to save memory or to reinterpret the binary representation of a value written in a given type using a different type. Unlike structure types, union types are rarely used.

Since we are building our type system in Smalltalk, it makes sense to take advantage of the symbols and of literal array present in Smalltalk to make a type notation. We added the *asLowcodeType* message to the the Symbol and Array classes to build our type notation. Table 1 presents a sample of our notation for primitive types, pointers and the special object type.

In our type notation, we are using #(void pointer) instead of #(void*) because we wanted to retain a Smalltalk based notation for our types. Types notations are parsed by evaluating the first element of the array literal looking for an identifier or nested type. The rest of the elements in the array literal are used for sending the respective Smalltalk message into the base primitive type. With this scheme, we have a flexible type system that can also be represented as literal value.

The only missing types and the trickiest to define are the structure and union types. Since they are very similar, we will only discuss the structure type.

In Pharo 5.0, slots (first class instance variable) were introduced (Verwaest et al. 2011). Classes can have slots and a layout that describes how the slots are composed. By making a custom structure layout and a custom slot class with type information, we support class definitions with a structure type. The instances of these classes are able to hold the data of a C structure as if they were instances of Smalltalk ByteArrays. For union types, we used the same approach as the structures.

Abstract Type	C Type	Lowcode Type	Size in Bytes
#void	void	NA	NA
#char	int8_t	int32	1
#short	int16_t	int32	2
#int	int32_t	int32	4
#long	int64_t	int64	8
#float	float	float32	4
#double	double	float64	8
#(void pointer)	void*	pointer	4 or 8
#(int pointer)	int*	pointer	4 or 8
#(float pointer)	float*	pointer	4 or 8
#object	NA	oop	NA

Table 1. Samples of the Lowcode type system notation.

For example, the definition of a structure called *Vector3* with three instance variables of x, y and z each one with the data type *#float* looks as follows:

```
NativeStructure subclass: #Vector3
    layout: StructureLayout
    slots: { #x &=> #float.
             #y &=> #float.
             #z &=> #float }
    classVariables: { }
    package: 'LinearAlgebra'
```

Supporting boxing. The notation for the *Vector3* structure type is *#Vector3*. In a similar way, the notation for the type of a pointer to this structure is *#(Vector3 pointer)*.

Since with this we can also have a Smalltalk object boxing the structure data, and we are only able to pass these objects as arguments during a message send, it makes sense to have a type boxed values. For this reason, we added the type notation *#(Vector3 object)* to denote an object that only contains the data of a *Vector3* structure in its interior. We use these types later to improve the generated code by the compiler when accessing the fields present in these structures, without having to unbox them from the object.

Now that we are having a suitable type system for taking advantage of Lowcode, our next step is to extend the compiler to use these types, check them and generate our extended bytecodes. Since we made our type system in Pharo, it is easy to add new type or extend the existing types if needed.

5. Extending the Compiler

After adding our extended bytecode instructions, and a type system implemented in Pharo along with a notation for these types, our next step is to extend the compiler to perform type checking and to generate our bytecode instructions.

The Pharo compiler is defined in Pharo and it is available as an object from the language itself. Classes understand the message called #compilerClass. Therefore we use the normal compiler by default and our extended compiler only when we want to improve the performance of some particular classes. This give us the ability to experiment with our extended compiler without breaking the complete image. In our setup we have a trait named *TWithNativeData* whose purpose is on enabling the Lowcode compiler in a specific class by overriding the #compilerClass method. The *NativeStructure* abstract base class uses this trait and provides some additional facilities for structures.

Pharo has a modular compiler known as the *Opal Compiler* (Béra and Denker 2013). The Opal Compiler has a pipeline architecture whose main stages are: The parser, the semantic analyzer, the bytecode translator, the bytecode intermediate representation builder and the bytecode encoder. The parser generates a generic AST. The nodes that compose this AST can be annotated with arbitrary properties by subsequent compiler stages. In Pharo it is possible to add metadata tags to the beginning of a method that are known as *pragmas* (Ducasse et al. 2016). By taking advantage of pragmas and the ability to annotate the AST, we implemented our extended compiler without having to change the Smalltalk parser, or the Smalltalk syntax.

The following code listing shows an example method that computes the dot product between two 2D vectors, by using typed local variables and typed arguments:

```
Vector2 >> dot: other
    <argument: #other
         type:#(Vector2 object)>
    <var: #result type: #float>
    | result |
    result := (x * other x)
             + (y * other y).
    ^ result
```

Our extensions to the compiler are concentrated in the semantic analyzer and in the bytecode translator. The bytecode translator takes the annotated nodes that are generated by the extended semantic analyzer to emit the most appropriate instructions for a given node in the AST. The translator trusts blindly the annotations made by the semantic analyzer. The semantic analyzer extensions takes care of the following tasks:

- Parsing pragmas with type information.
- Annotating each AST node that returns a value with the type of this value.

Conversion message	Target Type
#asObject	#object
#asNativeInt	#int
#asNativeFloat	#float
#asNativeDouble	#double

Table 2. Some special type conversion messages.

- Performing type inference when possible.
- Type check implicit and explicit type conversions in compile time.
- Identify special messages that perform type conversions.
- Identify messages that can be inlined, such as primitive data arithmetics, structure accessor and constructors.

Type conversion. Having type annotations is not enough to define an usable type system. We also need a way to perform type conversions. For doing type conversions we added special messages that convert objects into primitive values, and messages for converting primitive values into objects. Table 2 shows the special messages that we added to do these conversions. These messages are implemented mostly as no-operations in Smalltalk (i.e by returning self), or by calling the closest object conversion message to it. For example the *#asNativeFloat* message is implemented by just sending the *#asFloat* message to *self*. The utility of these special messages is not their implementation, but in providing more type information and the ability to Lowcode instructions to inline these special messages in the bytecode translator. Since most of the types can be converted into the *#object* type implicitly, using this message is usually redundant.

Inlining simple messages. The most important messages provided by objects that are representing structures are simple accessors messages, and trivial constructors. We define simple accessors methods as methods that are used to only to retrieve the encoded value of a structure, or to change a field present in a structure. In a similar way, we define trivial constructors as methods that create a new structure, and then set some of its fields. These methods do not have additional secondary side effects in addition to create the structure object or manipulate its fields. Since sending a Smalltalk message has an overhead, and when dealing with native primitive we have to perform a marshalling operation for sending the message, and then an unmarshalling when the message is receiver. Due to this overhead, we inline these special structure messages by using custom bytecodes for these operations.

To inline structure field accessors we mark them with the <*accessor*> pragma, to tell the compiler that this compiled method is a trivial accessor. As for trivial constructors, we mark them with <*constructor*>. In addition the selector name for getters has to be the same name as the field it is getting. In a similar way, the selector of a trivial setter method has to be composed by the name of the field followed by a colon. As for the selectors of trivial constructors they are composed by the names of the fields that are going to be set separated by colons. This matches standard conventions in Smalltalk for accessor methods and simple constructor methods.

The following code listings, shows examples of these methods for a 2D vector. It shows an example of a getter, a setter, a trivial constructor and a method that makes use of them along, with how type inference allows getting something that looks a lot like standard Smalltalk code with additional type annotations:

```
Vector2 >> x
    <accessor>
    ^ x

Vector2 >> x: value
    <accessor>
    <argument: #value type: #float>
    x := value

Vector2 class >> x: x y: y
    <constructor>
    <argument: #(x y) type: #float>
    ^ self new x: x; y: y; yourself

Vector2 >> + other
    <argument: #other
         type: #(Vector2 object)>
    ^ self class
            x: x + other x
            y: y + other y
```

The extended version of the bytecode translator receives an AST annotated by the semantic analyzer extended with these type annotation and type checking. With these type annotations, the bytecode translator generates the normal bytecodes used by Pharo when dealing with normal objects. The translator uses our extended bytecode instructions for translating the special messages for type conversions, native integer and floating point arithmetics, trivial accessors and trivial constructors. When translating a non-special message send into a native data value, the bytecode translator will generate the bytecodes for converting the native data value into an object, and then the translator will generate the bytecodes for a normal message send.

6. Benchmarks

For testing our hypothesis we developed a basic library with two simple mathematical objects that are normally used in 3D graphics: a three dimensional vector and a 3x3 matrix. The interest of these objects comes because we measured in a profile of Woden, a 3D graphics engine written in Smalltalk, that 90% of its time is spent doing matrix multiplications and inversions in a very dynamic scene.

6.1 Setup

We performed three basic benchmarks:

- 3x3 matrix with 3x3 matrix multiplication,
- 3x3 matrix with 3D vector multiplication,
- and 3D vector normalization.

For each one of these benchmarks, we make two versions of the basic linear algebra library, one version using the normal compiler, and another version using our extended compiler. We also execute both versions of the benchmarks in two different versions of the Pharo virtual machine, one an interpreter-only version and one including the just-in-time compiler. Each one of our benchmarks is executed by sending a message that contains the number of elements to compute. For the number of element we pick the values between 10^5 and 10^6 in increments of 10^5. We also execute the benchmarks for each number of elements ten times to be able to compute and average and a standard deviation for each number of elements.

We also compute the speed up factors for the different number of elements by dividing the time spent by the version using normal Smalltalk objects, with the time spent by the version using native data. For each benchmark, we only present the average of these speed up factors for the two versions of the virtual machine.

Since we are trying to measure a constant speed up factor, it seems that this benchmarking protocol could seem a bit complicated. However, the garbage collector can be triggered by creating objects containing a boxed structure for the results of the benchmarked methods. Because the garbage has to perform a linear scan of all the created objects, the tested algorithms can be affected by the number of objects used.

6.2 Measures

3x3 matrix with matrix. The 3x3 matrix with matrix benchmark creates two matrices. These two matrices are multiplied the requested number of times. Each of the results are immediately discarded. In Figure 1 we plotted the times taken by this benchmark. The version using native data is 2.96 times faster on average when using using the just in time compiler. When using the interpreter based virtual machine, the native version is only 1.05 times faster on average.

3x3 matrix with 3D vector multiplication. As for the 3x3 matrix with 3D vector multiplication benchmark, this benchmark creates an array with random vector which are multiplied by a matrix, and then stored into another array. The time taken for creating the array of random vectors is not measured by this benchmark. The resulting times of this benchmark are plotted in Figure 2. This benchmark is 4.73

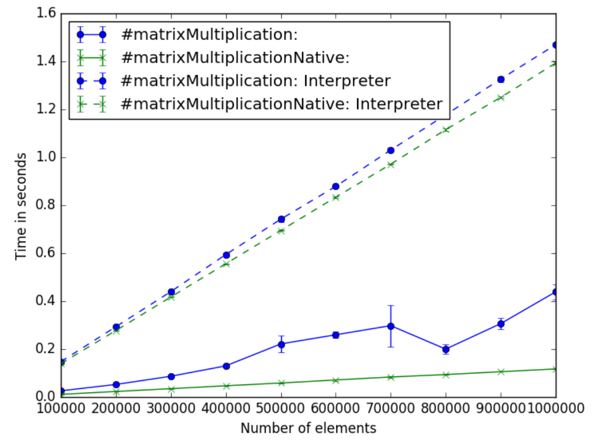

Figure 1. Benchmark results for 3x3 matrix-matrix multiplication.

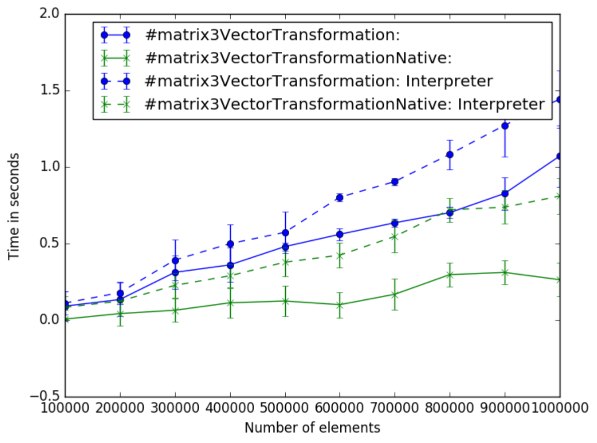

Figure 2. Benchmark results for 3x3 matrix vector multiplication.

times faster on average with the just in time compiler, and it is 1.63 times faster on average with the interpreter.

3D vector normalization. Finally, for the 3D vector normalization benchmark we are also creating another array of random vectors for creating the testing dataset. The time spent on creating the testing dataset is also not taken into account for this benchmark. The times taken for normalizing the vectors present in the dataset are plotted in Figure 3. This benchmark is 3.82 times faster with just in time compiler, and it is 1.72 times faster with the interpreter.

7. Result Discussions

7.1 General Remarks

Our results show clearly that we obtain a performance improvement in each byte code in both the interpreter and

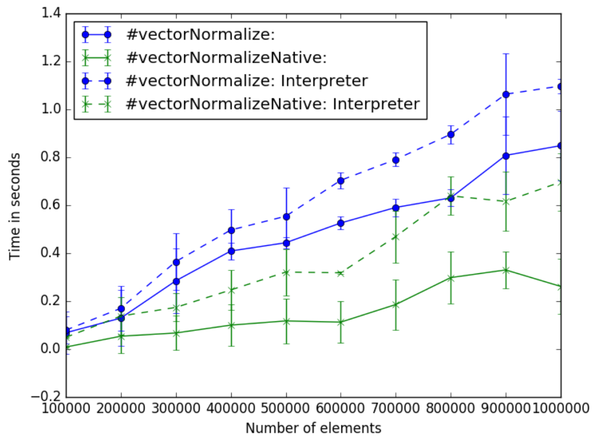

Figure 3. Benchmark results for 3D vector normalization.

the just-in-time compiler VMs. Getting a better performance with the just in time compiler was an expected result because the Lowcode floating point arithmetic instructions are translated into the native x86 SSE instructions for single precision floating point arithmetic. For the virtual machine interpreter-based version getting a modest performance improvement between 6-72% is quite a surprise. In the interpreter most of the performance is spent decoding the bytecode instructions instead of just executing them. The code generated by our extended compiler is usually a lot larger for a version using native types instead of object. Because there are lot more instructions that have to be decoded by the interpreter, we were afraid that the performance improvement of using native code could be outweighted by the reduced performance of the instruction decoding when using the interpreter. With these results we can conclude that our extended bytecode instructions and our extended compiler does not introduce performance regressions when using an interpreter only virtual machine.

7.2 About Floating Point Objects

The second aspect of this benchmark is related to the way that floating point objects are encoded in Pharo. Pharo uses an object model known as *Spur* (Miranda and Béra 2015) which has two variants: a 32 bits version and a 64 bits version. The biggest difference between these two versions is on the alignment of objects pointers. Pointers to object have to be aligned to 4 bytes in the 32 bits version of the object model, and it has to be aligned to 8 bytes in the 64 bits version. This alignment requirement means that the two or three least significant bits depending of the object model variant are always zero for objects. The Spur object model takes advantage of this property for encoding immediate objects such as SmallInteger in the pointers themselves. This object model is not able to support immediate floating point objects in the 32 bits version, but it is able to encode immediate float objects in the 64 bits variant. This means that each floating point operation in Pharo is creating an object that is allocated in the garbage collected heap for storing its results. Our extended bytecode instructions do not create Smalltalk objects for holding the results of the native floating point arithmetics, which removes the overhead of garbage collecting floating point values. In addition to the garbage collection overhead, our extended instructions do not perform run time type checking on their inputs, so they do not have the overhead associated to dynamic message lookup. This overhead is not present unless the native value is casted into object for sending a message that is not inlined or supported by our extended instructions and compiler.

If we were using the Spur 64 bits object model that supports immediate floating point object, we expect a reduced performance gap between the normal Smalltalk benchmark and the benchmark extended with type annotations. Unfortunately, when we did this work the 64 bits version of Spur along with its virtual machine was incomplete and in an experimental state. This means, that we are currently not able to compare the performance difference for floating point arithmetics against a Smalltalk with support for immediate[2] floating point objects.

8. Related work

Quicktalk (Ballard et al. 1986) is a Smalltalk dialect that achieves a similar purpose. It provides the ability to define primitives in the image side by adding adding type annotations to a restricted version of Smalltalk and generating machine code directly. Unlike Lowcode, Quicktalk generates actual primitive methods that can fail and are executed in a separate context to the context that is used by Smalltalk. These restrictions implies that Quicktalk has to inline all of the message sends. Due to the ability of mixing normal bytecode instructions with Lowcode instructions in a single method, we do not have to inline all of the messages because we can perform a normal Smalltalk message send.

NativeBoost (Bruni et al. 2013a) is a framework for writing Smalltalk primitives in the image side by generating platform specific machine code in the image side. The main use of NativeBoost is in building a foreign function interface for calling external C function. For doing a FFI callout, NativeBoost provides a nice interface where the user only has to specify the called C function signature. The main advantage of NativeBoost is that its implementation requires minimal changes in virtual machine because machine code generation is done in the image side. However, NativeBoost has the following issues:

[2] Immediates are values that can be encoded in an object pointer. In 32-bits, SmallInteger (values in range $[-2^{29}, 2^{29-1}]$) and Character (values in range $[0, 2^{30-1}]$) are represented as immediates. In 64-bits SmallFloat is added which represents 64-bit floats whose exponent is the middle 8-bits of the 11-bit range, that of double-precision floats.

- NativeBoost is very hard to port to different platforms. It requires a full assembler for the CPU specific ISA in the image side.
- The methods generated by this framework bypasses completely the virtual machine. A NativeBoost method cannot perform a Smalltalk message send, except by calling a callback function to the virtual machine.
- When an user needs to use a custom marshalling or callout behavior, he has to write platform specific assembly code.
- By writing platform assembly code instead of Smalltalk code with some type annotations, it is possible to optimize our simple linear algebra library using NativeBoost instead of Lowcode. This is error prone and not portable at all.

Yet another approach is the one provided by Nabujito (Bruni et al. 2014a). Nabujito is a just in time compiler written in the image side. Nabujito receives bytecode instructions and generates the same machine code that is generated by the just in time compiler present in the Pharo VM. By being in the image side, Nabujito can have a high level of interaction with the objects present in the image side such as the Smalltalk compiler. It should be possible to implement Lowcode using Nabujito. The problem of Nabujito is that it is not a complete virtual machine, and in fact Nabujito relies in the runtime provided by the Pharo VM. Implementing Lowcode required some changes in the runtime support of the virtual machine in addition to the changes to the interpreter and the just in time compiler.

9. Future work

We have not completed the support for pointers yet. Our next task is to implement full support for dealing with pointers and manipulating memory. By having support for pointers, we believe that we can get an even better performance improvement for these benchmarks. Instead of manipulating a Smalltalk array of boxed structures, we want to manipulate big byte arrays that encode the complete dataset. By doing this we believe we can avoid the object system and the garbage collector completely for getting even better performance.

Another benchmark and comparison we want to do is using the Spur 64 bits object model with the immediate floating point objects. We will perform these benchmarks when we have a production ready virtual machine with a just in time compiler for this object model. We are still expecting a performance improvement using native floats because Spur 64 bits should have a minor performance overhead in checking the immediate object tags, decoding the immediate operands for arithmetic message and then encoding the result as an immediate. The Lowcode instructions by taking advantage of the type annotations and type inference should not have this overhead, but we do not know how much different the performance gap will be for this case.

We also want to add support for C function pointers to our type system, and the ability to call directly the C functions pointed by these pointers. The ability of having C function pointers and the C data types should allow us to bypass the FFI for communicating with third party libraries. By having the C data types and the Smalltalk objects in the same time, we can perform the marshalling from the Smalltalk objects into C objects manually. Currently, the FFI when calling a C function uses a fixed behavior for marshalling arguments for calling C functions which is determined by the signature of the C function provided to the FFI library. This fixed behavior can be wrong or make code complicated for some common C idioms such as returning multiple values by using passing pointers to local variable in the function caller.

Acknowledgments

We thank Eliot Miranda for supporting the concept of the Lowcode and providing guidance for modifying the just in time compiler of the virtual machine to implement the Lowcode instructions. We thank Clément Béra and Eliot Miranda for making the extended bytecode set that added the inline primitive support, and for working and documenting the Spur object model. We also thank Marcus Denker for the work on Slots and providing some ideas for making the structure layout.

References

M. B. Ballard, D. Maier, and A. Wirfs-Brock. Quicktalk: A smalltalk-80 dialect for defining primitive methods. *SIGPLAN Not.*, 21(11):140–150, June 1986. ISSN 0362-1340. doi: 10.1145/960112.28711. URL http://doi.acm.org/10.1145/960112.28711.

C. Béra and M. Denker. Towards a flexible pharo compiler. In *International Workshop on Smalltalk Technologies 2013*, 2013. URL http://rmod.inria.fr/archives/papers/Bera13a-OpalIWST.pdf.

C. Béra and E. Miranda. A bytecode set for adaptive optimizations. In *International Workshop on Smalltalk Technologies (IWST 14)*, Aug. 2014. URL http://rmod.inria.fr/archives/papers/Bera14a-IWST-BytecodeSet.pdf.

C. Béra, E. Miranda, M. Denker, and S. Ducasse. Practical validation of bytecode to bytecode jit compiler dynamic deoptimization. *Journal of Object Technology*, 15(2):1:1–26, 2016. doi: 10.5381/jot.2016.15.2.a1. URL https://hal.inria.fr/hal-01299371.

C. Bruni, S. Ducasse, I. Stasenko, and L. Fabresse. Language-side foreign function interfaces with nativeboost. In *International Workshop on Smalltalk Technologies*, 2013a.

C. Bruni, L. Fabresse, S. Ducasse, and I. Stasenko. Language-side foreign function interfaces with nativeboost. In *International Workshop on Smalltalk Technologies 2013*, 2013b. URL http://rmod.inria.fr/archives/papers/Brun13a-NativeBoostIWST.pdf.

C. Bruni, S. Ducasse, I. Stasenko, and G. Chari. Benzo: Reflective glue for low-level programming. In *International Workshop on Smalltalk Technologies*, 2014a.

C. Bruni, L. Fabresse, S. Ducasse, and I. Stasenko. Benzo: Reflective glue for low-level programming. In *International Workshop on Smalltalk Technologies 2014*, 2014b. URL http://rmod.inria.fr/archives/papers/Brun14a-IWST-Benzo.pdf.

G. Chari, D. Garbervetsky, S. Marr, and S. Ducasse. Towards fully reflective environments. In *Onward! 2015*, 2015. URL http://rmod.inria.fr/archives/papers/Char15a-Onward-ReflectiveVM.pdf.

M. Dias. *Towards Self-aware Virtual Machines*. PhD thesis, University Lille 1 - Sciences et Technologies - France, may 2014. URL http://rmod.inria.fr/archives/phd/PhD-2014-Bruni.pdf.

S. Ducasse, E. Miranda, and A. Plantecl. Pragmas: Literal Messages as Powerful Method Annotations. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, Aug. 2016. URL http://rmod.inria.fr/archives/papers/Duca16a-Pragmas-IWST.pdf.

A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. ISBN 0-201-13688-0. URL http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989. ISBN 0-201-13688-0.

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

B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978. ISBN 0-13-110163-3.

E. Miranda and C. Béra. A partial read barrier for efficient support of live object-oriented programming. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 93–104, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754186. URL http://doi.acm.org/10.1145/2754169.2754186.

T. Verwaest, C. Bruni, M. Lungu, and O. Nierstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In *Proceedings of 26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, pages 959–972, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048138. URL http://rmod.inria.fr/archives/papers/Verw11a-OOSPLA11-FlexibleObjectLayouts.pdf.