Waterfall: Primitives Generation on the Fly

Removed for blind review

Abstract

Modern languages are typically supported by managed runtimes (Virtual Machines). Since VMs have to deal with many concepts such as memory management, abstract execution model and scheduling, they tend to be very complex. Additionally, VMs have to meet strong performance requirements.

This demand of performance is one of the main reasons why many VMs are built statically. Thus, design decisions are frozen at compile time preventing changes at runtime. One clear example is the impossibility to dynamically adapt or change primitives of the VM once it has been compiled.

In this work we present a toolchain that allows for altering and configuring components such as primitives and plug-ins at runtime. The main contribution is Waterfall, a dynamic and reflective translator from Slang, a restricted subset of Smalltalk, to native code. Waterfall generates primitives on demand and executes them on the fly. We validate our approach by implementing dynamic primitive modification and runtime customization of VM plug-ins.

1. Introduction

Modern high-level languages usually rely on runtime systems (Virtual Machines) which provide abstract execution models that promote portability, automatic memory management and enforce certain security properties. VMs are complex pieces of software that historically, for efficiency reasons, were developed in static low-level languages. Over the last years a new branch of VM implementations written in high-level languages appeared [8]. By relying on better abstractions the VMs would be easier to develop, debug and modify. The downside of the gain in abstraction is a further separation from the low-level execution model of the underlying hardware. It requires substantial efforts to make VMs written in high-level languages efficient [17].

Self-hosted VMs [7] are written in the same language they support. A common way of tackling them was to restrict the actual semantics for the code targeted to the VM construction. This is the path taken in many implementations like Squeak [11] for Smalltalk and PyPy [17] for Python. The code of these VMs is written in a subset of the language they provide. This subset typically has the same syntax, but the semantics are restricted in order to statically bind everything at compile time. Essentially, this prevents polymorphism and dynamic method dispatch which are substantial features of high-level host languages. A quite similar approach is observed in popular research VMs like Jikes [2] or Jalapeño [1] for Java.

VMs written in high-level languages have improved considerably and it has become a prolific area [7, 20]. One promising technique is by promoting low-level high-level programming frameworks [9] which aims at solving the problem of the abstraction mismatch of metacircular or self-hosted VMs. With these elaborate tools and others like runtime code specialization the performance penalty can be considerably mitigated.

However we identified that most of these VMs still suffer from an important limitation:

It is not possible to change or configure many of the most important design decision at runtime. Key components are frozen at compile time and hidden at runtime

One example of this limitation are the primitive methods. They are used by VMs to perform basic operations such as arithmetic operations and object allocation or to accomplish performance demanding task more efficiently [10]. They live at VM-level and are statically bound at compile time. In certain cases a developer may be interested in creating or adapting a primitive, for instance for instrumenting an application. In some languages this can be achieved by exploiting reflective capabilities [18]. But the overhead imposed by actual

implementations of reflection make them impractical for performance demanding tasks [13].

Similarly, some VMs support the concept of plug-ins to enable the efficient implementation of recurrent general operations (e.g, file access, floating point operations, compression, etc). Plug-ins also provide some features that can not be implemented fully at language-side. Moreover, only after releasing, developers may identify a hotspot in their application and improve its performance by providing a plug-in for it. Unfortunately, in many cases this implies that every user of the application will need to recompile its VM in order to update the plug-in. Other option is to ship the plug-in as a library and depend on the user's platform compilers or dynamic linkers. Clearly, giving end-users responsibilities, or worse, requiring them to update their VM every time a new plug-in is released is undesirable.

To overcome this limitations we conceived a toolchain which solves the aforementioned problems. Its main component is Waterfall, a runtime translator for code in a syntax equal the the language the VM supports. We take the approach of high-level low-level programming one step further and use it dynamically at runtime to change VM behavior. Concretely, Waterfall enables to dynamically modify primitives and plug-ins from language-side, outperforming existing approaches that are purely reflective. We use a single-language approach instead of using Foreign Function Interfaces (FFI) [12, 14] to access external libraries. This enables to debug, inspect and change the code at runtime with the same tools the developers use for their general purpose tasks. Exploiting Waterfall capabilities, developers and experienced users are able to tune their VMs at runtime without the need of external tools.

The contributions of this paper are:

- A toolchain, written in the language of the VM, that enables to compile and activate code at runtime.
- A proof of concept that uses the toolchain for adapting VM behavior at runtime, without the need of a system restart, considerably outperforming the reflective solutions.
- An empirical validation demonstrating the approach is feasible and the penalty in terms of performance is reasonable penalty.

2. Statically Defined Primitives

One of the main components of VMs are primitives. Primitive routines provide the runtime with the capability to perform essential operations that the language could not supply by its own. For instance, create objects or provide access to low-level structures. On the other hand, they are also used to optimize some critical bottlenecks [10, ch.3, p.52]

Each time the runtime activates a method that is a primitive, it swaps the execution mode. The VM, instead of interpreting bytecodes or performing message dispatching, directly executes the binary code of the primitive. Primitives typically are already statically compiled, further optimized and in general they are written in the same language used to implement the VM. Changing or extending primitives is profitable in the case where core features of a language needs to be inspected or modified. For instance, if a VM accesses its instance variables through a primitive, immutability could be easily and efficiently achieved by changing that primitive.

The problem is that primitives are deeply coupled to the VM building process which is a complex and time-consuming task. Moreover, changing primitives may not be possible since it requires access to the VM source code, which is not always open. Even in that case, primitives are written on a different abstraction level than the high-level language the VM supports. This is a complex barrier for the common developer of the host language since he needs to work in two abstraction levels at the same time and deal with different development environments and tools [9]. As a consequence it is observed that the set of available primitives is statically defined and frozen at compile time, making them difficult or even impossible to change.

Self-hosted VMs try to overcome this two-language problem by implementing the VM using a language with a similar syntax as the host language, mitigating several of the problems mentioned previously. But, even though the syntax is usually similar, or even equal, the developer still has to be aware that the semantics are different from the host language. For example, in Squeak and PyPy the VM is implemented in a language that is almost identical to the host language but eventually the semantics are reduced to C-like expressions.

Another example similar to primitives in high-level dynamic environments is the Pharo VM [4, 16] plug-in infrastructure. Pharo provides a large number of plug-ins tailored towards specific but heterogeneous tasks such as algebraic matrix operations, floating point precision computations, file management, etc. All these tasks have in common that they are intensively used for repetitive and performance demanding tasks that are not fulfilled by the standard runtime execution model. In essence, plug-ins consist of a set of primitives isolated on separated modules.

Plug-ins are, in many cases, built and deployed with the VM. This is unpleasant and involves exactly the same limitations as static primitives in a larger scale. The other scenario is to build them as dynamic linked libraries (DLLs), like modules outside VM, and link them dynamically. In an case, plug-ins imply bigger

binary footprints, more complexity and overhead for deployment. Plug-ins force users to work with VM binaries with a large amount of statically built code that they perhaps never use. Concerning runtime adaption, the only option to change and modify plug-ins is by using the second approach (DLLs) and by loading them dynamically with external OS tools, possibly after unloading a previous version. Relying on external compilers brings its own set of problems. Additionally to the given CPU architecture difference, compilers and linkers differ on each operating system.

Below we summarize the limitations of current VMs, concerning runtime adaptation, that we want to address:

1. Primitives are statically defined and frozen at compiled time.
2. Primitives can not be efficiently changed at runtime.
3. It is complex, heavy-weight and error prone to extend or upgrade VM modules at runtime.

3. Context

Smalltalk is a good representative of a dynamic, object-oriented and reflective high-level language. We choose Pharo[1] [4, 16], an open-source Smalltalk-inspired environment, for developing the proof of concept used to validate our approach. Pharo features a self-hosted VM with primitives accessible at language-side. Moreover, the VM provides a powerful plug-in infrastructure with a simple interface, enabling its extension with efficient low-level functionalities, which works very similar to primitives.

In the rest of this section we provide context for understanding the Waterfall approach presented on next section.

3.1 Pharo VM

The VM for Pharo is the Cog VM [15], which has a unique construction process that was inherited from the Squeak VM [11]. The language under which Cog is developed is a subset of Smalltalk known as Slang. This has the advantage that Slang programs can be managed and explored in the same way as any other code from the host language. The main difference is that Slang code is not executed by the runtime. Instead there is a compiler that translates Slang to C, wherefrom a standard C compiler takes over. This is why Slang basically has the same syntax as Smalltalk but it is semantically constrained to expressions that can be resolved statically at code generation time. Hence Slang's semantics are closer to C than to Smalltalk.

[1] `http://pharo.org`

In Cog, primitives are defined at VM-side using Slang and they are frozen at compile time. Following a similar approach as primitives the Cog VM can be customized with plug-ins which are also written in Slang and use the same compilation strategy already explained. Plug-ins can be seen as modules that encapsulate a set of particular primitives. One of the main differences is that plug-ins can be compiled in external files, like independent libraries, and then linked to the VM at runtime.

3.2 Benzo Framework

Waterfall relies on a framework called Benzo [5] that provides dynamic high-level low-level programming techniques. Benzo provides a language-side assembler, a dynamic code generator and a set of generic primitives for activating native code from language-side.

Instead of building a separate infrastructure for generating and activating native code we rely on this generic framework. Benzo is based on 5 generic primitives to activate native code and access VM symbols. Benzo uses its own language-side assembler to generate native code without external help. Once the native code is ready Benzo uses a dedicated primitive to active the code. This very basic interface is enough to open doors for new language-side implementations of typical VM-level tools. Benzo is already successfully used in production for a language-side FFI implementation [6].

4. Waterfall Compiler

Waterfall's main task is to translate Slang to native code at runtime. The compiler was designed as a chain of transformations that receives as input a high-level representation of code (Slang) and returns as output a lower-level one (binary code) going through different intermediate representations in the process.

4.1 General Architecture

A high-level graphical description of the processes involved in generating native code is shown in Figure 1. The Waterfall toolchain begins by parsing Slang to an Abstract Syntax Tree (AST) representation. Then the AST representation is translated to native code enforcing C-like Slang semantics. Waterfall generates assembler instructions at language-side and executes them using the Benzo framework. Waterfall can also translate the AST into an intermediate representation (IR) featuring an hybrid IR between Three Address Code (TAC) and Static Single Assignment form (SSA) as a pre-step before native code generation.

Even though Slang is syntactically similar to Smalltalk it is actually closer to C. It is completely bound at compile time and most of their constructs can be directly translated to C. Considering that fact, the binary code that Waterfall generates follows similar code

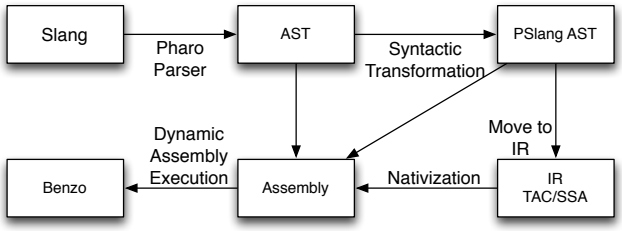

Figure 1: All Waterfall stages for Slang to native compilation.

conventions used by C compilers. Concretely, Slang variables and arguments are pushed on the native stack and Slang messages are treated very similar to C function calls. Waterfall also sees every variable just as a word in memory leaving its semantics interpretation (type) to the developers.

Figure 2 exposes a high-level diagram of the core classes involved in the translation. Waterfall's main component is the **ClosureNativizer** that has as collaborators: a set of **SendNativizers**, a static set of **Primitives** of the language and a chain of **Converters**.

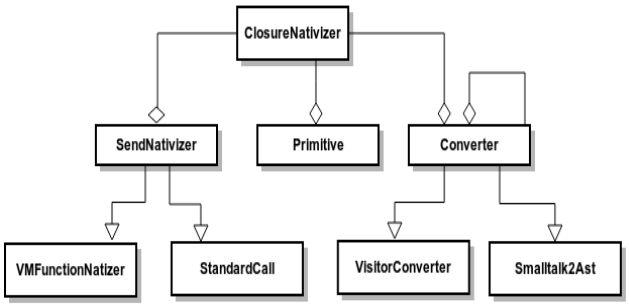

Figure 2: Waterfall high-level class diagram

Primitives. Certain operations at Slang-level can not be expressed as calls to other Slang methods and thus are defined as assembler templates. Much like in the normal Pharo environment, Slang relies on certain primitive operations that can not be expressed at the same level. For instance, algebraic operations such as $+$, $-$, bit manipulation or direct memory access operations have to be treated differently. Instead of generating their native code from a Slang method we inline an already nativized template (instance of the **Primitive** class).

SendNativizer Hierarchy. The **SendNativizer** objects have the responsibility of handling the different type of calls (calling convention) which we describe in more detail in Section 4.4.3.

Converter Hierarchy. Converter objects essentially receive as input a representation of the code, apply transformations and emit a modified output. A chain of these transformations defines the compilation phases that Waterfall will actually execute. **VisitorConverter** is the class that represents the behavior for walking through nodes in a visitor fashion. All transformations that traverse ASTs doing something different for each kind of tree node are implemented as visitors and collaborate with a **VisitorConverter**. With this architecture it is quite simple to define new conversions such as optimizations or further IRs, and link them to the chain. As an example of the reuse of standard environment tools, **Smalltalk2AST** converter is instantiated with the Pharo standard parser. The converter is named Smalltalk since their syntax is the same, but Waterfall uses it for translating Slang.

4.2 Replacing Primitives with Generated Code

Once Waterfall generated the code, there still remains the step of actually installing and executing it at runtime. In the case of changing a primitive, the generated code must be executed instead of the original static primitive. Below we describe a summary of the main steps involved in this task:

1. Exploiting reflective properties of the environment with meta programming techniques, Waterfall identifies whenever a plug-in or primitive is called. In the Pharo environment these methods are identified by a special **pragma** as first statement.
2. The pragma is removed and instead Waterfall installs code that invokes the generation of a new primitive on demand from Slang code.
3. Finally, Waterfall relies on Benzo for executing the native code. Subsequent calls to the same primitive function will not execute step 2, avoiding the compilation overhead.

4.3 Replacing Plug-ins with Generated Code

There is only a marginal difference when using Waterfall for generating plug-ins since plug-ins are in some way modules encapsulating a set of related primitives. They are more general. For dealing with them there exist a class **PluginNativizer** whose instances receive as input (collaborators) a set of methods (primitives) from a plug-in. Additionally **PluginNativizer** knows the class where the new dynamically nativized primitives should be installed. This nativizer is responsible for calling the translator on each of the methods and installing the code generated as new primitives on the corresponding class.

With this approach it is quite simple to mark a plug-in from language-side as dirty for later recompilation.

Concretely, marking it as dirty implies to trigger a call to the nativizer with only the changed method as input. Since the interface described above is written at language-side, and Smalltalk is a reflective language, all this behavior can be dynamically activated. Plug-ins developers can also define their plug-ins as language functions that trigger the nativizer whenever called for the first time. It is also easy to make changes to a plug-in or create a different version and let the user decide to use the one that better fits his requirements.

4.4 Compilation Steps

In the rest of this section we describe the standard steps of transformations (**Converters**) needed for translating Slang code to native instructions.

4.4.1 Transforming Slang to AST

This step benefits from the cooperation of high-level tools at language-side provided by the Pharo environment. Concretely, since Slang is a subset of Smalltalk, Waterfall uses the standard Smalltalk parser to generate the AST. The parser itself is written in Smalltalk and resides at language-side, which makes it an easy target for debugging and possible extensions. Hence Waterfall does not need a special parser.

4.4.2 Computing the Set of Reachable Methods

Before executing a Slang method Waterfall translates its code into machine instructions. However looking at an isolated method itself is not enough. Slang programs can be decomposed in several methods so Waterfall also has to translate all the methods reachable for all possible executions starting at the method under translation. Essentially, every potentially executable method has to be nativized. However it must remain clear that Waterfall currently does not support polymorphism, only modularization.

4.4.3 Transforming AST to Native Code

The actual nativization step consists of a visitor that receives an AST as input and returns a stream of bytes which then is loaded into memory for execution. For abstracting from the binary code generation, we use an already developed abstract assembler called ASMJit[2] that is an independent tool inside the Benzo framework.

The Slang AST has few kind of nodes. Each node is translated into a set of native instructions. As an example, we briefly explain how to translate nodes corresponding to variable references (more details about the translation of other nodes in section 5). The visitor checks what kind of variable is referenced (i.e., temporary, argument, global, low-level symbol) and it finds its memory address: for temporaries and arguments it will be a native stack location, for VM globals the position where they were loaded in memory. Finally, the translator interacts with ASMJit for pushing the gathered memory address into the stack.

4.4.4 Dynamically Executing Generated Native Code

After generating the native instructions there still reamins the task of activating the native code and passing it down to the VM. This is not directly possible since in general VMs strongly separate the language from the low-level environment. Due to this barrier it is simpler to ensure portability and security properties for a VM. However, in our case this poses a limitation since we want to dynamically execute instructions at VM-level which were created at language-side. As already explained, Waterfall is supported by Benzo framework for overcoming this limitation.

5. Implementation Details

The main goal of Waterfall is to provide support for changing low-level (VM) behavior at runtime. Due to the existing restrictions of the VM we apply certain simplifications or rely on external tools to accomplish our goal.

5.1 Resolving External Symbols

Code written by developers may reference other code or data that already exists in the managed runtime, like global variables or VM functions. In these cases, the nativization procedure has to find the actual addresses in memory where these static references lie and inline them into the generated native code. All this is necessary because Slang eventually resides at VM-level when executed and thus it is not possible to access global objects directly. While executing a primitive it is not possible to interact with the high-level environment. For instance accessing an object using the standard Smalltalk way by sending a message would cause recursion problems.

To access the position of VM internal symbols referenced in Slang methods, Waterfall relies on an existing Benzo API to interface with C libraries which is based on **dlsym**. We developed also a parser of the **nm** Unix command for gathering the positions not accessible to **dlsym** in the VM binary. As a consequence of the external tools selected, the complete functionality of the compiler can be only obtained on Unix platforms. However, it should not be difficult to develop a parser for a tool similar to **nm** on Windows or other platforms.

5.2 Special Parameters

Pharo primitives receive their arguments on the stack. The VM is responsible for pushing them right before

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

calling the primitive. Low-level primitive code special has to be careful when accessing these arguments. The Cog VM uses a moving Garbage Collector (GC) which requires careful access to high-level objects when in VM-level code. The GC is not aware of the C stack. Therefore, if a GC pass happens during primitive activation, Waterfall pointers to language-side objects held on the C stack are not updated resulting in dangling pointers that would cause severe troubles. Waterfall simplifies the access to high-level objects by using a single statically known position for all parameters. This way only a single memory address has to be registered at the garbage collector for not moving it. Since the VM is single threaded and no two primitives can be executed at the same time it is safe to rely on a single global argument position.

5.3 Slang Purification

The complete Slang language supports special syntax for inlining C expressions. That means actually that it is at least as expressible as C. It also allows for other expression such as type **pragmas** which contain type information and are directly translated to C types. Waterfall currently does not fully support these two support special Slang expressions. However to validate our approach we reimplement these two features in a simple fashion.

In the case of types there exists a special converter that walks over the **pragmas** of the nodes, parses them and finally assigns basic type information to the variables that were reified during the previous transformations. The converter also marks if their arguments are used as value or as reference. Concerning the inlined C macros, we realized that they were mostly used to call external functions so we decided to provide a special language construction only for that cases.

We finally provide a special converter that performs simple string substitutions for the Slang features we implement differently.

5.4 Managing Stack Frames

The most complex node of the AST (Section 4.4.3) to deal with by Waterfall is the one for messages in Smalltalk which is reduced to function calls in Slang. The generation of native code in this case implies defining a calling convention for argument passing and register preserving (similar to C in this case): pushing correctly the arguments, preserving the needed registers, calling to the right place and resuming control at return. For each function call, a *Context* object is instantiated which represents the stack frame and has a pointer to its parent. This context is responsible for determining the actual position on the stack for every reference to a variable. Since Waterfall allows blocks (lambda functions), it must manage very carefully the context stack since a variable reference could be in another context far from the current one. Finally, as an example of the different calling conventions, calling VM functions implies sticking to a C ABI, whereas for Slang internal calls there is a receiver of the function (implicit parameter) that is always pushed on the stack.

6. Validation

We present two case studies: an essential primitive and a language-side plug-in untied from the VM building process. The first experiment validates that our solution efficiently addresses the first two limitations identified in Section 2 concerning primitives. The second experiment shows how Waterfall overcomes the limitation regarding language-side VM extensions. We run all the benchmarks for this paper with the SMark[3] benchmarking tool. On each benchmark we measure 50 runs and take the average time.

6.1 Essential Primitives Instrumentation

We distinguish two types of primitives: essential and non-essential. Essential primitives are required for the bootstrap and vital operations of the language, such as creating a new object or activating a block. Such primitives can not be easily implemented at language-side. The second category of primitives are mainly used for optimization purposes and could be replaced by language-side code.

In this particular case study, we focus on essential primitives. Instrumentation of essential primitives is an error-prone task falling in many cases in non-termination due to recursive loops. A difficult candidate is the **basicNew** primitive, which is responsible for instantiating new objects. Even a very simple instrumentation task such as printing the address in memory of the created object is problematic. If during the printing process another object is created, the very same instrumented **basicNew** primitive would be triggered.

Using reflective techniques it is possible to avoid this loop. Essentially one would have to inspect the current stack for previous activations of the modified primitive before using it. If the primitive has been used before, the code jumps to the original unmodified primitive. Thus, breaking the recursive loop. However this approach imposes a considerable overhead.

6.1.1 Experiment

We present the code of two approaches for instrumenting the object creation primitive: a pure language-side solution and one translated and executed by Waterfall.

The language-side version of an instrumented **basicNew** looks as follows:

[3] `http://smalltalkhub.com/#!/~StefanMarr/SMark`

	Average Time	Relative Time	w.r.t Waterfall Instr.
Unmodified	0.28 ± 0.16 ms	1.0×	–
Unsafe reflective instrumentation	21.80 ± 0.33 ms	≈ 78×	≈ 2,8×
Secure reflective instrumentation	27.72 ± 0.40 ms	≈ 99×	≈ 3.6×
Waterfall-based instrumentation	7.72 ± 0.27 ms	≈ 28×	1.0×
Waterfall-based unmodified	7.08 ± 0.23 ms	≈ 25×	–

Table 1: Slowdowns comparison for instrumentations of the essential primitive **basicNew**.

```
Class>>basicNew
    | object |
    object ← super basicNew.
    FileStream stdout ifNotNil: [ :stream |
        stream << object nbAddress << String lf ].
```

However, printing on the standard output might easily fall in a recursive loop as described before. Hence the safe version needs an additional recursion guard:

```
Class>>basicNew
    RecursionGuard
        ifStackContains: #basicNew
        do: [ ↑ self unmodifiedBasicNew ].
    FileStream stdout ifNotNil: [ :stream |
        stream << self name << String lf ].
    ↑super basicNew
```

In Waterfall we define a new version of **basicNew** in a small Slang method which itself is a wrapper around the unmodified primitive:

```
WaterFall >> slangBasicNew
    | oop value |
    oop ← self stackAt: 0.
    self
        callVMFunction: #printOop
        withArguments: { oop }.
    ↑self
        callVMFunction: #primitiveNew
        withArguments: {}.
```

Much like the reflective solution, the instrumented Slang version of the primitive delegates the main functionality to the original one. The reflective version uses a normal message send to call the original primitive. In Slang, after calling a low-level function that is responsible for printing the object memory address, it performs a function call to the VM-level **basicNew** function.

6.1.2 Results

In Figure 3 we compare the run times for different instrumentation approaches of the **basicNew** primitive. We make different comparisons, measuring the creation of 100 and up to 1000 objects. We run the experiment for: the standard primitive, a version with an unsafe reflective instrumentation, a safe version of a reflective instrumentation with a secure guard, a primitive compiled by Waterfall for creating objects and finally an instrumented Waterfall version. Table 1 illustrates the relative slowdown factors.

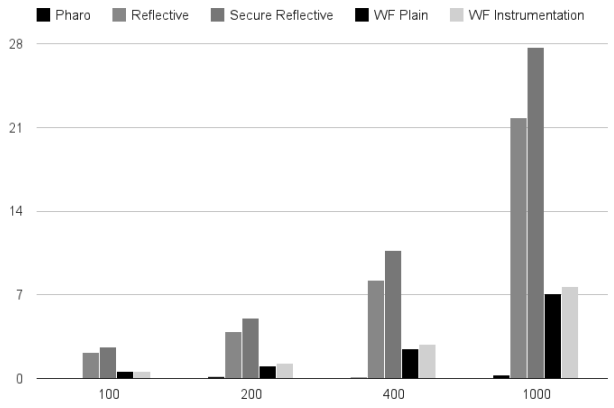

Figure 3: Instrumenting the creation of objects with reflection and with Waterfall

Some interesting conclusions can be inferred from the chart. First of all, it can be seen that in most cases the time for creating objects with the standard Pharo primitive is almost negligible. This is because creating objects is an essential task in an object-oriented environment and thus the primitive is highly optimized. Then it is also visible that just the overhead of calling Waterfall is quite high. This is because Waterfall is still a prototype and we have not focused on optimizations yet. Actually, is not our goal to compare Waterfall with a fully optimized C compiler.

It is also worth noting that the cost of instrumentation using Waterfall is negligible compared to the reflective techniques. As the chart exhibits, plain Waterfall versions are similar in time to Waterfall instrumented version while the table shows that reflective instrumentation imposes a slowdown factor of 78× and 99×. This is encouraging since it shows that if Waterfall is further optimized it would get closer to the pure static times.

Finally, concerning the analysis only for instrumented versions, the chart and table also exhibit that the reflective solutions are considerably slower than the ones based on Waterfall (a slowdown factor is between 3 and 4 depending on the approach). We compare Wa-

terfall with two reflective approaches for completeness reasons but, since Waterfall avoids the recursive loop, a fair comparison will be with the safer version. The final cost of this operation is correlated with the size of the call stack it must traverse. Since this slowdown factor of almost 4× was obtained using the benchmarking framework that generates a pretty shallow call stack, we conjecture that a real application may suffer more overheads, favoring our approach of using Waterfall. Moreover, the implementation in Waterfall of simple optimizations like function inlining will surely enlarge even more the differences.

6.2 Towards Dynamic Customizable VMs

In order to provide a stronger validation we choose an interesting and general enough plug-in, frequently used by most of the users. We basically strip it from the VM. Then, we use Waterfall for dynamically and lazily compiling and executing the functions which are actually used.

6.2.1 Experiment

In Pharo all file related operations are delegated to a plug-in named the **File Plug-ins**. We choose to experiment with this plug-in since it encompasses several functionalities and is exhaustively used by most standard users. It also is one of the most complex plug-ins, because it has a strong interaction with the OS.

The experiment consists on evaluating the feasibility of extracting the functionality from VM-side and using Waterfall for generating only the functionality required on demand.

6.2.2 Results

Following the same methodology already explained for the previous validation, we compare the execution time for creating directories with the standard static plug-in with the time it takes for creating them with the function dynamically generated with Waterfall. Figure 4 shows the results. The X axis represents the number of different directories the test created. Each one is a call to the primitive. The Y axis exhibits the average time.

It can be clearly observed that concerning the performance is almost the same with the static as with the dynamic approach. Also it is visible, and it seems like a contradiction, that the growth is non-linear. For instance, creating 2000 directories is more than twice than creating 1000. This is because the more directories created, the more the environment works, more objects created and then more garbage collection cycles.

We noted that we obtained very good timing results. We conjecture that this could be related to the fact that the main cost of file operations is expended by the OS. For other plug-ins, the Waterfall version may expose some performance degradation. However, the

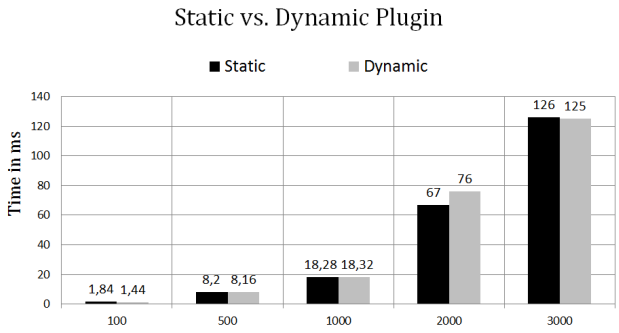

Figure 4: Static vs. dynamic plug-in comparison for creating files

main focus of this experiment is to show the feasibility of our approach on removing plug-ins from the VM. **File Plug-ins** is an excellent candidate for that goal since it is one of the most-used plug-ins. We managed to move it completely to language-side and show that it has no impact on the usability of the environment. The discussion about efficiency was already presented on the previous experiment.

7. Related Work

An approach more similar to ours is QUICKTALK [3]. It was conceived to compile Smalltalk code directly to binary code. Developers must include type annotations in order to bound all method invocations at compile time. This focus on performance and the development of a complex compiler for a new Smalltalk dialect. In contrast Waterfall creates a bridge between the VM and the language-side. Unfortunately, this tool is not available to perform an empirical evaluation.

Another noteworthy Smalltalk implementation is Smalltalk/X[4] that has excellent C integration built into the language. Using dedicated syntax, C expressions can be written inline. Smalltalk/X explicitly uses this approach to define primitives transparently at language-side (in C). Much like Waterfall primitives can be modified at runtime, however Smalltalk/X does not focus on a one-language approach where the VM and the dynamic primitives share the same language.

High-level low-level programming [9] encourages to use high-level languages for system programming. In this work authors present a low-level framework which is used as system interface for Jikes, an experimental Java VM. Methods have to be annotated in order to tell compiler to use the low-level functionality. Although this work is an step forward the use of high-level languages to build system software, the strong separation

[4] `http://www.exept.de/en/products/smalltalkx`

between low-level code and runtime does not allow for reflective extensions of the runtime.

Pharo benefits from Squeak[11] an early self-hosted VM for Smalltalk. More recent self-hosted approaches include Klein [19] for Self, PyPy [17] for Python, Maxine [20] for Java or Tachyon [7] for JavaScript. The Maxine VM stands out as it truly focuses on productivity and developer interaction. Maxine uses abstract and high-level representations of VM-level concepts and consistently exposes them throughout the development process. Inspectors at multiple abstraction levels are readily available while debugging, giving insights to the complete VM state. Compared to Maxine, Waterfall currently lacks the debugging tools which would enable a truly seamless interaction with the low-level world. However, Maxine focuses on Java, a language with inferior reflective capabilities compared to Pharo. Hence the live interaction with the VM is only presented in the development phase and not exposed to the language-side. Maxine would be an excellent candidate to implement our approach for Java.

Apart from the VM community, Waterfall shares many ideas with research conducted in the high-level reflection domain. For instance, Röthlisberger et al. [18] present a tool to mitigate runtime adaptions with unanticipated partial behavioral reflection . Built on top of Smalltalk they present a tool to limit the computational overhead that dynamic runtime reification introduces. This contrast with Aspect Oriented Programming solutions where the runtime has to be modified upfront to allow for partial behavior reflection. In the scope of our work, the latter approach corresponds to static changes at VM-level. Waterfall on the other hand allows for unanticipated changes.

8. Conclusions

In this work we present a toolchain that allows for altering and configuring components such as primitives and plug-ins at runtime. The main component is Waterfall, a dynamic and reflective translator for Slang, a restricted subset of Smalltalk. Waterfall is implemented completely at language-side and it is integrated in Pharo. Waterfall generates primitives and plug-ins on demand and executes them on the fly.

Even though Waterfall still lacks substantial optimizations it outperforms dynamic instrumentation of primitives based on purely reflective approaches. It also enables to have dynamically generated plug-ins which perform reasonably well with respect to their statically compiled counterparts.

We believe this approach provides a flexible mechanism for adapting and evolving VMs and enables developers to deploy them with a smaller footprints. VMs can be later customized by users according to their needs, at runtime and without resorting to external tools.

Concerning conceptual improvements, with Waterfall we encourage the use of high-level low-level programming at runtime. Furthermore we advocate the importance of self-hosted VMs to control and modify essential parts of them from language-side.

We also managed to enhance quality properties of the environment by decoupling it from the low-level building infrastructure while modifying low-level behavior. Operating systems, compilers and linkers tends to impose important constraints. For instance, with Waterfall it can be completely avoided for altering some components at runtime the common low-level cycle: compile → link → save to file → load to memory. As a consequence, our approach also improves portability since the compilation infrastructure is in general very dependent on the platform.

8.1 Perspectives

Even current self-hosted or metacircular VMs freeze many aspects at compile-time. We think VMs should be easier to evolve and adapt dynamically, using reflective runtime capabilities. At the same time, the VM performance should be comparable with the solutions written in low-level languages. We believe the work presented in this paper together with other techniques such as gradual typing and type inference, opens up new perspectives about the possibility of approaching our vision.

In this setting we plan to work on improving the Waterfall back-end in order to produce more efficient code. This includes powerful static and dynamic analysis techniques targeted specially for highly dynamic environments.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing jalapeño in java. *SIGPLAN Not.*, 34(10):314–324, Oct. 1999.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44(2):399–417, Jan. 2005.

[3] M. B. Ballard, D. Maier, and A. W. Brock. QUICKTALK: a Smalltalk-80 dialect for defining primitive methods. *SIGPLAN Not.*, 21(11):140–150, June 1986.

[4] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[5] C. Bruni, G. Chari, S. Ducasse, and I. Stasenko. Reflective low-level programming. *Submitted to AOSD'14*, 2013.

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

[7] M. Chevalier-Boisvert, E. Lavoie, M. Feeley, and B. Dufour. Bootstrapping a self-hosted research virtual machine for javascript: an experience report. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 61–72, New York, NY, USA, 2011. ACM.

[8] B. Folliot, I. Piumarta, and F. Riccardi. Virtual Virtual Machines. pages 1–4.

[9] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[10] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[11] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *OOPSLA'97: Proceedings of the 12th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 318–326. ACM Press, Nov. 1997.

[12] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[13] J. Malenfant, M. Jacques, and F. N. Demers. A tutorial on behavioral reflection and its implementation. Proceedings of the Reflection '96 Conference, 1996.

[14] J. S. Miller and S. Ragsdale. *The Common Language Infrastructure Annotated Standard*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[15] E. Miranda. The Cog Smalltalk virtual machine. In *VMIL '11: Proceedings of the 5th workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*. ACM, 2011.

[16] Pharo Smalltalk environment. `http://pharo.org`.

[17] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM.

[18] D. Röthlisberger, M. Denker, and É. Tanter. Unanticipated partial behavioral reflection. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 47–65. Springer, 2007.

[19] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM.

[20] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, Jan. 2013.