Hybrid Execution: Combining Ahead-of-Time and Just-in-Time Compilation

Christoph Pichler
christoph.pichler@jku.at
JKU Linz
Austria

Paley Li
paley.li@oracle.com
Oracle Labs
Czech Republic

Roland Schatz
roland.schatz@oracle.com
Oracle Labs
Austria

Hanspeter Mössenböck
hanspeter.moessenboeck@jku.at
JKU Linz
Austria

Abstract

Ahead-of-time (AOT) compilation is a well-known approach to statically compile programs to native code before they are executed. In contrast, just-in-time (JIT) compilation typically starts with executing a slower, less optimized version of the code and compiles frequently executed methods at run time. In doing so, information from static and dynamic analysis is utilized to speculate and help generate highly efficient code. However, generating such an efficient JIT-compiled code is challenging, and this introduces a trade-off between warm-up performance and peak performance.

In this paper, we present a novel way to execute programs by bringing together the divergence that existed between AOT and JIT compilation. Instead of having the JIT compiler analyze the program during interpretation to produce optimal code, critical functions are initially executed natively with code produced by the AOT compiler in order to gain a head start. Thus, we avoid the overhead of JIT compilation for natively executed methods and increase the warm-up performance. We implemented our approach in GraalVM, which is a multi-language virtual machine based on the Java HotSpot VM. Improvements in warm-up performance show a speed-up of up to 1.7x.

CCS Concepts: • **Software and its engineering** → **Dynamic compilers**; **Just-in-time compilers**; **Interpreters**; *Source code generation.*

VMIL '23, October 23, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0401-7/23/10.
https://doi.org/10.1145/3623507.3623554

Keywords: ahead-of-time, just-in-time, virtual machine, performance

ACM Reference Format:
Christoph Pichler, Paley Li, Roland Schatz, and Hanspeter Mössenböck. 2023. Hybrid Execution: Combining Ahead-of-Time and Just-in-Time Compilation. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3623507.3623554

1 Introduction

Two fundamentally different approaches have dominated the way programs are compiled and executed. One approach is to compile source code to machine code ahead of time (AOT). This native code is then directly executed on the machine. For a long time, this has been the most common approach for a wide range of programming languages. A vast amount of effort and research has been invested in AOT compilation, resulting in compilers that are able to produce code with impressive performance. This has helped establish languages such as C and C++ that are among the most widely used programming languages [23].

AOT compilation relies on static analysis to produce optimized machine code, which has limitations. Alternatively, code can be compiled during execution, which is called just-in-time (JIT) compilation, allowing it to utilize run-time information for optimization. This approach employs a typical (but not necessary) two-tier architecture for a JIT compiler: While either execution is blocked, or an interpreter is engaged, an optimizing JIT compiler compiles the most frequently executed functions into machine code. This leads to increased peak performance as soon as the optimized machine code is ready. As an advantage, JIT-compiled code can revert to the interpreter and/or can recompile when needed, making it more popular among languages where static analysis is difficult, such as dynamic languages [8]. However, the use of run-time information comes with the drawback of a worse warm-up performance compared to statically compiled code. To counter the innate performance issue of dynamic languages as well as to leverage the benefits

of native code, there has been a noticeable rise in projects in dynamic languages utilizing native libraries.

For such extensions written in C, warm-up performance is improved if AOT compiled native code is used rather than using interpretation and JIT compilation. Especially for a polyglot world, this leads to the question how managed and native execution can be brought closer together.

In our work, we tackle this issue: We introduce a *hybrid execution mode* in GraalVM, a multi-language ecosystem including a JIT compiler. In our *hybrid execution mode*, functions in the source code can be either interpreted and JIT compiled by the GraalVM compiler, or they can be compiled ahead of time and the resulting native code is executed directly. The resulting performance heavily depends on how we select which code is run natively and which code is interpreted and JIT compiled. The *hybrid execution mode* aims to improve the warm-up performance of traditional JIT compilation. The native code generated by the AOT compiler is usually far more optimized code than the initial code interpreted by JIT compilers in their initial warm-up stage.

The remainder of the paper is structured as follows: In Section 2, we explain the basics of LLVM, compilation, the GraalVM ecosystem, and the different GraalVM components. In Section 3, we describe our approach of mixing AOT- and JIT-compiled code. In Section 4, we look at the implementation of our approach. Before we conclude our paper and discuss future work in Section 6, we evaluate our approach in Section 5.

2 Background

In this Section, we will explore the intermediate representation of LLVM, the GraalVM ecosystem, and the interplay of ahead-of-time and just-in-time compilation in these two ecosystems.

2.1 LLVM and Native Execution

LLVM [15] is a cross-platform compilation framework. Its intermediate representation, LLVM IR, is statically typed and in static single assignment (SSA) form [12]. LLVM IR is platform-independent and often used as a universal IR for different platforms and architectures. It comes in a dense bitcode form as well as an equivalent human-readable format, and is usually compiled ahead of time.

Clang [1], a popular C/C++ compiler, uses LLVM IR as its intermediate representation and performs platform-independent optimizations on it. These optimizations are performed on the IR before different platform-specific LLVM backends produce efficient machine code. This is useful if some other LLVM backend is used, such as Sulong, the LLVM component of the GraalVM ecosystem. Most LLVM backends compile LLVM IR down to machine code, which is then executed directly on the machine. This is called native execution.

For implementing our *hybrid execution* approach, we used the clang compiler to generate LLVM IR and native code from given C/C++ source code.

2.2 Managed Execution and JIT Compilation

Besides native execution, there also exist languages such as Java, Python, or JavaScript, which require *managed execution*. This means execution with services such as garbage collection or JIT compilation.

Usually, an AOT-compilation result of a managed language is not compiled to native code and thus not executed on the bare machine directly. Rather, another program, a *virtual machine* (VM), interprets (AOT-)compiled intermediate representation. In managed execution, the virtual machine usually applies automatic garbage collection and further optimizations at run time.

A common optimization is JIT compilation, where frequently executed code is optimized and further compiled to efficient machine code at run time. The rest of the program is still interpreted by the VM. As soon as optimized code is available, it replaces the previous interpreted version [8].

A JIT compiler focuses its effort on only optimizing frequently executed code, called *hot* code. The interpreter collects profiling data during execution and uses that information to optimize the code. Thus, the execution speed of managed code is usually lower at the start due to the need for many different kinds of analyses. Python and JavaScript are two popular and widely-used examples of languages where a managed runtime with JIT compilation is used [3, 5, 13].

2.3 Challenges with JIT Compilation

The inherent challenge of JIT compilation is the trade-off between the time spent on optimizations and the resulting peak performance: Optimizing more heavily can increase peak performance at the cost of longer application times. There will always be an innate struggle between favoring a fast warm-up against achieving excellent peak performance [8].

As JIT compilers are often used for dynamically-typed languages, another challenge is the lack of static type information. Variables of dynamically-typed languages can hold values of different types. This makes it harder for the JIT compiler to perform static type analysis and apply traditional compiler optimizations. The compiler needs to handle type checks and method dispatches dynamically, impacting the optimization opportunities and introducing additional runtime overhead [22].

Nowadays, high-performance JIT compilers use speculation, whereby runtime profiling data are used to make assumptions about the executed code. Based on these assumptions, the code is optimized with the effect that it might no longer be semantically equivalent to the source code. For example, array index checks can be omitted if the array is always accessed within range. To ensure correct execution, the assumptions made by the compiler are regularly checked

at run time. In case of a violation, the execution of the optimized code has to be removed, and program execution has to revert back to slower correct code. This deoptimization may happen rarely but can have a quite negative impact on run-time performance.

There have been several attempts to solve this deoptimization performance problem. One is a flight forward, whereby when an assumption is invalid, the compiler does not deoptimize but looks for another possible specialization instead [14].

2.4 Python and Its Implementations

Python [24] has recently become a very popular programming language and is heavily used in machine learning [23]. Of the several language implementations, CPython, which is written in C, serves as the reference implementation.

While CPython prioritizes ease of use, its performance is mediocre, especially for computationally-intensive tasks. A solution is to use the native extensions via the C API, allowing high-performance C or C++ code to be executed from Python programs. Furthermore, natively executed libraries such as NumPy provide optimized numerical operations, further enhancing their performance.

Another possibility to improve the performance of CPython is JIT compilation, as it has been done with GraalPython [3] or PyPy [6]. On average, GraalPython is 3.4x faster and PyPy is 4.8x faster than CPython. GraalPython is the Python component of the GraalVM ecosystem and will be explained in more detail in Section 2.6.5. PyPy [6] uses meta-tracing, which can transform an interpreter into a tracing JIT compiler [10]. As is typical for tracing JIT compilers, linear sequences of operations in the interpreter are compiled. This concept can compile loops containing linear sequences of code into highly-efficient machine code. PyPy is written in a restricted subset of Python called RPython.

Python's existing C API is optimized for CPython. If another Python implementation such as PyPy is used, the usage of the C API leads to a higher overhead. The HPy [4] project addresses this problem by introducing a new API layer that sits between the native extension and the Python runtime. Using HPy reduces the complexity of native extensions in Python.

In our work, we use GraalPython as the Python runtime, using HPy.

2.5 Recent JIT Compilers

The idea of combining the advantages of native execution and JIT compilation was first raised in the 1980s [19].

Recently, several state-of-the-art projects started to introduce native code to an environment with managed execution.

A widely-used JavaScript engine that uses both native execution and JIT compilation is Google's V8 engine [17]. It uses multiple compilation levels and two different JIT compilers to generate highly efficient native code. In addition,

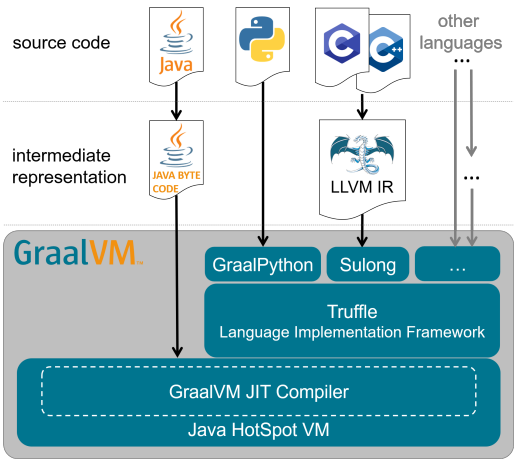


Figure 1. Ways to run languages on GraalVM

Google decided to simplify and speed up the compilation process by first producing less optimized native code, which is then optimized further at run time [21]. In contrast to GraalVM, the V8 engine has 16 ms to complete the entire compilation to keep the user response time acceptable.

As with the V8 engine, Ruby has evolved its internal compiler components over the years. Resulting in the release of the YJIT compiler in 2021 to provide maximum compatibility with existing Ruby code [11].

One of the fastest Ruby implementations is TruffleRuby, which is part of the GraalVM ecosystem.

2.6 GraalVM

GraalVM [2, 26] is an advanced virtual machine based on the Java HotSpot VM. Its core is an optimizing JIT compiler, and its polyglot execution environment supports a variety of programming languages, such as Java, JavaScript, Python, Ruby, and C/C++. Managed execution on GraalVM includes an interpreter and its highly optimizing GraalVM JIT compiler.

In this subsection, we describe the various parts of the GraalVM ecosystem that are relevant to the implementation of our *hybrid execution mode* approach.

2.6.1 Truffle. An important component of GraalVM is *Truffle* [25], which is a framework for implementing self-optimizing language interpreters using either byte code or abstract syntax trees (ASTs) as their intermediate representation. The GraalVM JIT compiler can produce highly efficient machine code from Truffle's intermediate representation.

There are different ways of using a language on GraalVM, as can be seen in Figure 1:

- Languages such as Java, Scala or Kotlin can be compiled down to Java byte code and then used directly with the GraalVM JIT compiler.

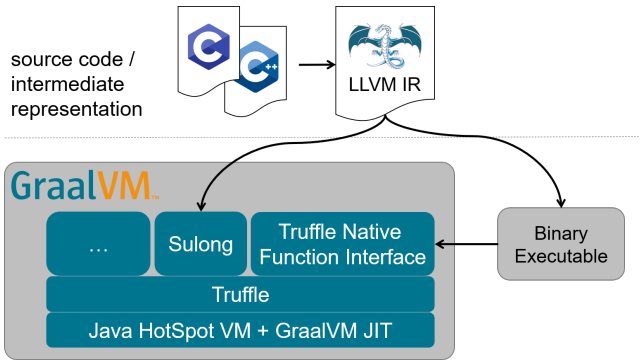


Figure 2. Truffle Native Function Interface: Calling native code within GraalVM

- For every other language, the Truffle language implementation framework is used. Language implementations using this framework create an AST or produce a custom byte code format from:
  - The source code, such as Python, Ruby or JavaScript, with the implementations GraalPython, TruffleRuby or GraalVM JavaScript, respectively.
  - An intermediate representation, such as LLVM IR with Sulong as its language implementation.

  This AST or byte code is then interpreted and JIT-compiled by the GraalVM compiler.

To run C/C++ programs on GraalVM, the source code has to first be compiled into LLVM IR. Sulong transforms LLVM IR to an AST, which is then finally interpreted and JIT-compiled in GraalVM.

As all Truffle language implementations use the same runtime, they can interact with each other. This Truffle interoperability concept will be shown in more detail in Section 2.6.6.

2.6.2 Truffle Native Function Interface. Truffle's Native Function Interface (NFI) [7] enables calling and executing AOT-compiled functions natively from a Truffle language runtime, as shown in Figure 2. Thus, the NFI can serve as a bridge between managed execution on GraalVM and native execution. It is mostly used for running code of existing native libraries, system APIs, or performance-critical code written in languages other than the Truffle language itself.

Internally, the NFI is implemented as another Truffle language that can be accessed via Truffle's interoperability mechanism.

2.6.3 Sulong. Sulong [18, 20] is GraalVM's LLVM language implementation, as shown in Figure 1. As it uses the Truffle framework, the LLVM IR is translated to an AST and then interpreted and JIT-compiled.

Sulong can run arbitrary LLVM IR. However, when a library in native code without LLVM IR is accessed via Sulong, such as any specific C libraries, Sulong uses the Truffle NFI to access and run the corresponding native code of the library. Since code accessed via Truffle NFI is run natively, no access to GraalVM features such as JIT compilation or Truffle language interoperability is possible. This is a difference to the situation when libraries are available as LLVM IR, since features of GraalVM can then be used.

Most of the changes we performed to implement our *hybrid execution approach* are located within Sulong.

2.6.4 Sulong Toolchain. Sulong comes with its toolchain, which is based on clang. It takes C/C++ code and emits both LLVM IR and native code within one file. First, it uses clang to produce LLVM IR and native code. The LLVM IR is then added as a special section to the resulting native code such that the compilation result is a file containing both native code and LLVM IR. When Sulong receives a file containing both formats (LLVM IR and native code), Sulong prefers LLVM IR. The fact that LLVM IR and native code are in the same file is essential for our approach of hybrid execution.

2.6.5 GraalPython. GraalPython [3] is the Python language implementation on GraalVM, using the Truffle framework. The language runtime is based on Java, which is different from other popular Python implementations. As mentioned before, on average GraalPython outperforms the reference implementation CPython [3].

However, the reference implementation achieves higher performance when it is combined with native extensions, especially with the NumPy library. This is because most of the runtime for NumPy is in the AOT-compiled native code, whereas the performance of the JIT-compiled Python code only has a minor influence in such applications. Due to the wide distribution of the NumPy library, its code is highly optimized.

Given that GraalVM can achieve similar peak performance as the reference implementation with C extensions, JIT compilation of GraalPython trades high peak performance for a slower warm-up. Therefore, it is hard for GraalVM to achieve the same overall performance for such pre-optimized code.

However, we believe that the performance gained by the reference implementation using native extensions can also be achieved by GraalPython. Through using the HPy project together with GraalPython, a more efficient access to native extensions can be realized.

2.6.6 Truffle Cross-Language Interoperability. Truffle enables running code written in different source languages within a single runtime. Thus, it is possible to call functions across different languages. For example, a cross-language function call is shown in Figure 3: The main method is written in Python (cf. Figure 3a). Before invoking code from a foreign language, the code itself has to be loaded in lines 2 and 3. After that, a parser written in JavaScript is called, that parses text and returns a JavaScript 2D point object (cf. Figure 3b). This JavaScript object is then stored in a Python

```
1 #fetch references
2 jsFile   = polyglot.evaluate(lang: "js",
        src: "pointParser.js")
3 llvmFile = polyglot.evaluate(lang: "llvm
    ", src: "pointLib.so")
4
5 #call a JS parser (returns JS object)
6 p = jsFile.parseTextTo2DPoint(inputFile)
7
8 #call a C function from Python with JS
    obj
9 llvmFile.mirror(p)
```

(a) `main.py` – content of main method, written in Python

```
1 function parseTextTo2DPoint(inputFile) {
2   ...; return {x: xval, y: yval};
3 }
```

(b) `pointParser.js` – JavaScript code

```
1 struct Point { int x; int y; };
2
3 void mirror(struct Point *p) {
4   p->x = -p->x; p->y = -p->y;
5 };
```

(c) `pointLib.c` – C code which is then compiled to LLVM IR/machine code (`pointLib.so`)

Figure 3. Example for Truffle interoperability: Python code calls JavaScript and C functions.

variable p. Then, in line 9 of Figure 3a, a C function is called with the JavaScript object p as a parameter (cf. Figure 3c). This C function is not called natively, but interpreted via Sulong and JIT-compiled by the GraalVM compiler.

In contrast to other polyglot systems, Truffle language objects are not modified when passed to another language. In our example, the point object from JavaScript is neither converted to a Python object when it is stored in p, nor to a C struct when it is passed to the C/LLVM function. It is kept as the original Truffle object.

The code shown in Figure 3 can also be JIT-compiled by the GraalVM compiler. This shows how different source languages can interact seamlessly, since cross-language function calls are compiled down to a single piece of machine code.

3 Hybrid Execution

The principle behind the *hybrid execution mode* is to incorporate native and managed execution (i.e., AOT and JIT compilation) within a single ecosystem, in order to harness the benefits offered by both.

The goal of our *hybrid execution mode* is to improve the overall performance of GraalVM when C/C++ code is executed, by improving warm-up performance while maintaining its already high peak performance. We aim to achieve this by increasing the amount of native code being run natively, in particular during the warm-up phase of the GraalVM JIT compiler.

The common scenario which we consider is a polyglot application containing native code. An example is parsing JSON or XML data into JavaScript objects using a Python library (e.g. libXML [16]). Internally, libXML accesses a C library, which GraalVM does via a cross-language interoperability access to the Sulong runtime of GraalPython. In such a case, it is not possible to run all C code natively, which will be explained in more detail in section 4.3.

The machine code produced by the AOT compiler becomes the basis for the execution of the application. This allows the applications to be initially run mostly on the AOT optimized code, even if this code is usually still slower than the code produced by the JIT compiler, while providing time for the JIT compiler to be able to perform its optimizations. Once the JIT compiler has been properly warmed up, the application will be switched to the code produced by the JIT compiler.

The Sulong toolchain, as mentioned in Section 2.6.3, ensures that both LLVM IR and machine code of every native code are available. Which is critical to our ability to be able to switch between native and managed execution modes.

A challenge for the *hybrid execution mode* is to determine which functions in the code should be executed natively and which functions should be run on GraalVM in managed mode with JIT compilation. For applications that are all C/C++ code, running everything natively may be the way to go. For polyglot applications containing native code, we aim for a mixture of native execution and managed execution, with an emphasis on native execution. Presently, a rudimentary selection process is used to identify the code candidates that are run natively. We explore these decisions in the next section.

4 Implementation

In this section, we explore the decisions behind our implementation of the hybrid execution mode.

4.1 Classifier Domain

One of our first decisions was to decide at what granularity to execute either natively or in managed mode. We settled at the function level, whereby a source function can either be executed natively or in managed mode.

There are several reasons for deciding at the function level: In well-written code, every function or method performs a coherent task. Since the decision of where to run code is based on the type of operations and the type of accessed

```
1  static struct Point *globalObj = init();
2
3  void fetchPoint(struct Point *p) {
4    foo(p->x);
5    *p = *globalObj;
6  };
7
8  void foo(int par) {...};
```

Figure 4. C code containing two functions, where one is called from the other. The function `fetchPoint` accesses a global variable.

data, splitting the execution mode by functions and methods is a reasonable way.

For calls between different execution modes, a way has to be found how such cross-execution-mode calls should be performed. The Truffle native function interface offers a convenient way to call native functions from managed code, thus a function-based decision allows using an already implemented mechanism.

Alternatively, one could decide on a higher level, e.g., per file or library, how code should be executed. However, we consider this classification as too high-level: A library can contain functions and methods with different tasks. While one function might be suitable to run natively, another one might need access to GraalVM features. A single access to managed data (which is not possible in native execution) would require the whole library to run in managed mode. This can cause significant slowdowns, especially if the remaining library code could be executed natively without any access problems. Thus, a more fine-grained decision than on the file/library level is needed.

On the other hand, also decisions on a more fine-grained level do not seem to be optimal: If we decided for every basic block or for every instruction how it should be executed, it would be necessary to provide access to all local symbols in both execution modes. However, due to JIT compilation and its optimizations, symbols might live at different locations on the stack, might be stored in different formats, or might have been eliminated altogether. Thus, the costs of achieving consistent execution would wipe out the performance gain.

Therefore, we aim for a decision based on the function level.

4.2 Switching Execution Mode: Method and Function Calls

The execution mode, between native and managed, is specified manually for every function. Function calls constitute the boundaries between the two execution modes.

Such a point can be seen in line 4 of Figure 4, where `foo` is called from the `fetchPoint` function. By default, the callee is executed in the same mode as the caller. In other words, if

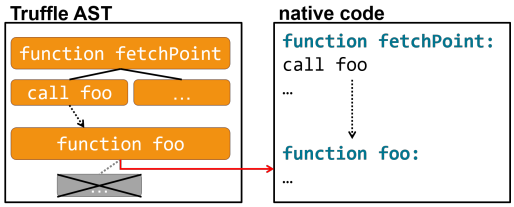


Figure 5. If `foo` should be executed native, then its call node in the LLVM Truffle AST has to be modified.

`fetchPoint` is run on Sulong in managed mode, then also `foo` is run on Sulong (when called from `fetchPoint`). On the other hand, native execution of `fetchPoint` would lead to a native execution of `foo`.

4.2.1 Managed Caller, Native Callee.

If `foo` should be executed in native code, a call from managed execution to native execution is needed. The call itself can be performed by the Truffle NFI.

To switch execution mode in this case, we have to modify the LLVM Truffle AST, as shown with the red arrow in Figure 5: The call node in the AST does not contain the callee's AST as a child but gets a reference to the native function instead. This replacement, which ensures native execution of the callee, can be performed during the symbol initialization phase.

4.2.2 Native Caller, Managed Callee.

When a natively executed function invokes a callee that should be run in managed mode, the situation is not as straightforward: For code being interpreted and JIT-compiled on Sulong, it is possible to modify the AST. However, modifying native code introduces more challenges.

A relatively simple workaround to trigger managed execution of a callee (when invoked from a native caller) is to use a global function pointer in the source code that is initialized to a function in the LLVM Truffle AST. Its managed execution is even triggered if the function pointer is invoked from native execution. As a drawback, this workaround implies a modification of the source code, as every function call has to be replaced by a call of the corresponding function pointer symbol.

However, our initial results show that performance gains can be achieved by running these functions in managed mode that are towards the top of the call graph. Functions that should be executed natively occur mostly towards the bottom parts of the call graph. Moreover, functions called by a natively executed function should also be executed natively in most cases. Therefore, situations where managed callees should be invoked from native callers happen rarely.

4.3 Objects and Variables

Since running code on GraalVM can happen in different execution modes, there are also different possibilities for

storing data. Native execution allocates and stores data in native memory, whereas GraalVM uses the Java heap to store objects. When functions that access the same object are executed in different modes, problems might arise depending on how the object is stored and accessed: Data stored in native memory might have to be accessed from managed execution, and vice versa.

4.3.1 Managed Objects and Native Memory. This subsection describes why accessing the managed heap from native execution may violate language semantics. Higher-level languages usually have their own runtime, which can perform additional semantic checks when objects are accessed. Java, for example, performs out-of-bounds checks for array accesses, or checks if pointers are not `null` before they are dereferenced. In contrast, execution of lower-level languages like C directly accesses memory without any checks.

When foreign language objects are accessed from C code in Sulong, Sulong forwards the access via Truffle's interoperability framework to the corresponding language runtime of the foreign object [25]. This way, language-specific access checks are performed automatically even in polyglot applications, and language-specific semantics are preserved. Thus, if a Python or JavaScript object is accessed in Sulong, the access itself is done via the GraalPython or GraalVM-JavaScript runtime, respectively.

If one converted an object from a higher-level language like Python to native memory, the raw data would be accessible from native code. Checks by the higher-level language runtime would then be missing and the semantics of the higher-level language would be violated. Therefore, managed objects from a foreign language can neither grant direct access from native code nor can they be transformed to native memory automatically. In general, any managed object must be treated as a black box in native code.

4.3.2 Access Scenarios. In the following, we will show different access scenarios and explain why not every function can be executed natively.

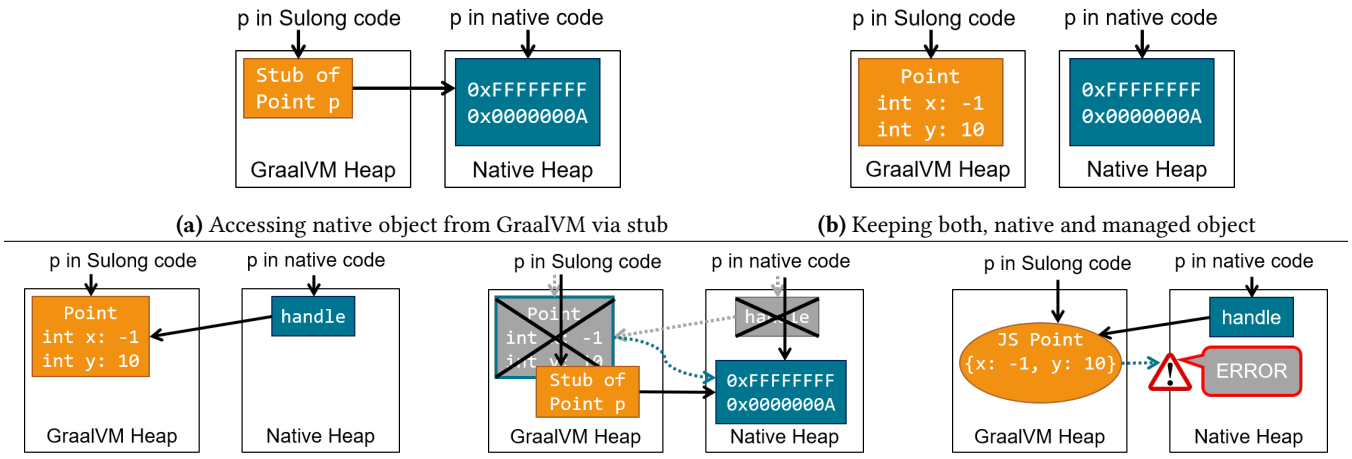
- `globalObj` can be stored on the native heap. In this case, native execution is trivial. For access from GraalVM, a stub is created on the managed heap which forwards all operations to native memory, as shown in Figure 6a. The access from GraalVM to native memory also works via the Truffle NFI.
- `globalObj` can also be stored twice: Once in native memory and once on the Java heap, as shown in Figure 6b. As each execution mode can access its own instance of the object, the global object has to be constant/read-only to avoid inconsistencies.
- The remaining case is where `globalObj` is stored on the managed heap only, which is shown in Figure 6c: Native execution gets a handle to `globalObj` that can be stored into variables and passed to other functions. As long as it is not accessed/dereferenced in native code, nothing has to be changed. However, when native code accesses such a handle by dereferencing it, the managed object on the Java heap has to convert itself to a native memory object.
  - If the conversion to native memory is possible for the corresponding managed object, as shown in Figure 6d, the situation is similar to Figure 6a, and program execution continues. This is the case for all managed objects that originate from C code, i.e., for all objects that have been created by Sulong and managed LLVM execution. Therefore, the `Point` object in Figure 6c can also be transformed to native memory (given that it has been created by Sulong) as no language semantics are violated.
  - Otherwise, the object cannot be converted to native memory as shown in Figure 6e, which occurs when foreign language objects are accessed from native code. Thus, functions that access objects originating from a managed language cannot be run natively, but must be executed in managed mode.
  - If the object is a function pointer, the function is executed in managed mode when invoked, and only the result is converted to native memory (as described in Section 4.2.2).

The case where conversion to native memory is not possible would not happen very often, since managed execution via Sulong is preferred when both LLVM IR and native code are available. Therefore, most accesses to and from compiled C code (LLVM IR/native code) happen on the managed heap, which can deal with objects of foreign languages without any problems. However, using *hybrid execution mode*, which increases the amount of natively executed code, the scenario where natively executed code accesses managed data from other languages happens more often. Thus, we should try to avoid this case.

A way of reducing the number of conversions from managed to native memory is to apply them lazily. As an example, we consider a managed C array containing pointers to elements. When the array reference is passed to native code, only the array itself gets converted to native memory, while the references to the objects can still point to handles of the corresponding native objects. Thus, if a native array containing managed objects is passed to native code, objects are not converted to native memory if they are not accessed. This saves unnecessary conversions that might have caused problems as described above.

4.4 Determining the Execution Mode for Certain Functions

For every function in the native source code, we have to decide if GraalVM should choose to interpret and JIT-compile it, or if the AOT-compiled code should be called. Thus, we



(a) Accessing native object from GraalVM via stub

(b) Keeping both, native and managed object

(c) Managed object: Native object receives a (native) handle which on access triggers a conversion to native memory

(d) Accessing managed object from native: Object is transformed to native memory, a stub is created on the GraalVM side

(e) Same scenario as in Figure 6c, but now the managed object cannot be converted to a native one, which results in an error

Figure 6. Memory object handling in different scenarios

have to provide an oracle that takes a function name as an input, and determines which execution mode (native or managed) to run.

To achieve a high warm-up performance, it sounds reasonable to execute as much code natively as possible. However, there are scenarios where native execution is not possible, as we have shown in Section 4.3.

To make matters worse, it cannot always be determined statically if a function can be executed natively. This is because every pointer in C and C++ code can possibly point to a managed object. For a given C array of pointers to objects, it is impossible to statically detect whether accessing certain elements has to happen in managed mode.

4.5 Classification Heuristics

As an initial step, we implemented a manual classification. The developer can specify those functions that should be executed natively by hand. By setting a run-time option, GraalVM reads a file that contains the library and the name of each function that should be executed natively. During symbol initialization, the corresponding call nodes in the Truffle AST are then modified, as shown in Figure 5.

We implemented a manual classification because it gives us something dynamic to experiment upon to find an automated heuristic. To get a feeling for which functions should be executed natively, the application can be fully run in managed mode on Sulong/GraalVM together with a profiler. This profiler is already implemented in GraalVM and can find those functions that get the most CPU time. Usually, these functions are towards the bottom part of the call tree, mostly even leaf functions.

For a fully-automated approach, which is our long-term goal, there are several ways to go. One possibility is to decide the execution mode based on the function type: A function is executed natively if all its types in the signature are primitive. By that, it is ensured that at least parameter and return values can be accessed from native code without any problem. However, access to managed global variables can still happen from native code. Therefore, run-time information has to be used to check if a function only accesses elements that can be converted to native code.

One of the open aspects concerning a classification heuristic is the boundary when switching to native execution is faster. Managed execution can be even faster during warm-up than a call to native execution via the Truffle NFI. This happens when the overhead introduced by the Truffle NFI is higher than the speed-up gained by avoiding the slower warm-up of managed execution. Thus, the heuristic will have to also consider the expected run time of a function.

Later on, more sophisticated heuristics will also change decisions at run time. Our long-term goal is to still reach peak performance of managed execution, while using the native code in the beginning to get a higher warm-up performance. Therefore, some if not all functions which are executed natively in the beginning will be switched to managed execution during a later stage in execution.

4.6 Native Symbols

When providing access to native symbols in managed execution, the following aspects have to be considered:

- Native symbol information: To access the necessary symbol information (such as the name and the type of a symbol), it is not necessary to access native code. As both the LLVM IR and the native code section originate from the same source code, the existing data structure

for the LLVM symbol information can be used, which is already implemented in Sulong.

- Static symbols: The `static` keyword in C keeps symbols local to a file, i.e., static functions or fields are not accessible or callable from the outside, also not for the Truffle NFI. This leads to a problem if Sulong calls a `static` function of the same file in native mode.
  To retain access to the native symbol, we need to calculate its native address during run time, which consists of the following parts:
  – The symbol table of every native file contain the relative symbol offsets within the library.
  – To determine the library's base offset in memory at run time, a lookup call on the link map is conducted.
  By summing these two numbers for each symbol, we can compute the pointers to native memory, even in situations where accessing symbol information is not possible for the Truffle NFI.

5 Evaluation and Results

For evaluating our *hybrid execution mode*, we are aware of the fact that it cannot be applied to any given application yet, as there might be a situation where natively executed code accesses managed objects. Therefore, we aimed for a scenario where the main application is written in a higher-level language like Python but uses a C/C++ library like a JSON or XML parser. Although also managed objects can be passed to code of the C/C++ libraries, we select functions to run natively that only deal with data that can be converted to native memory.

Although measuring warm-up for virtual machines can be hard [9], other existing GraalPython benchmarks for warm-up times in similar size show that it usually takes 3 – 5 iterations to warm up, which is why we always measured the first 10 iterations [3]. We executed each benchmark (consisting of 10 iterations each) 10 times and provide average and maximum interquartile length.

The system features an Intel Core i7-9700 with 8 cores, 32 GB of RAM and Ubuntu 22.04. The C code of both benchmarks was compiled with the Sulong toolchain of GraalVM 23.1.0. To see differences in warm-up and peak performance, the same GraalVM version (23.1.0) was used for running the benchmark.

5.1 UJSON: Comparing the Hybrid Mode with as Much/Little Native Execution as Possible

To assess our approach described in Section 3, we compare our *hybrid execution mode* to two existing execution modes. On the one hand, running all code on GraalVM (and nothing via native execution) should be seen as a baseline. On the other hand, executing as much code as possible natively shows the other end of the spectrum.

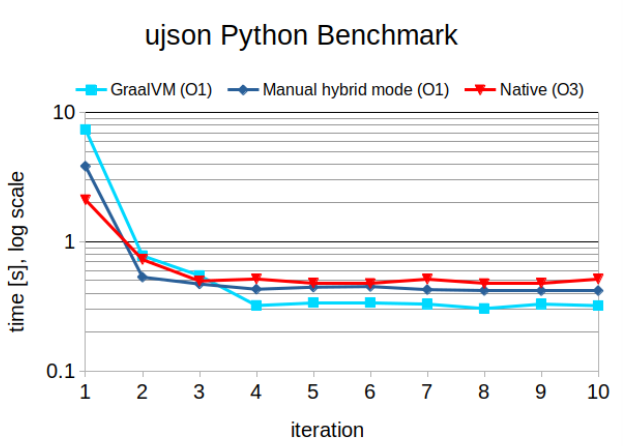


Figure 7. Warm-up behavior in different execution modes (time measurement → lower is better). The interquartile range for the first iterations is up to 1.5 seconds, while peak performance interquartile range is around 0.1 seconds.

For the benchmark itself, we use existing UJSON benchmarks, which are JSON parser benchmarks written in Python. To stay within GraalVM, we thus use GraalPython as a language runtime for the benchmark code. Concerning the modes, we compare three different scenarios:

- GraalVM: All available code (Python and C parts) are run on GraalVM, i.e., they are interpreted and JIT-compiled and make use of the Truffle framework (GraalPython and Sulong). Inlining is enabled.
- Native: All Python code is run on GraalVM, whereas all C code is executed natively, which means that implementations of the Truffle framework are used less frequently.
- Hybrid: We selected the three C functions of the benchmark with most execution time to be executed natively, while all other C functions are run in managed mode under Sulong. In the UJSON benchmark, the functions with most execution time are leaf and pure functions, i.e., functions that do not call other functions, and functions without side effects. Should a native function not be a leaf function, all other functions called from it are still going to be executed natively. Also, native functions are currently always executed natively, i.e., their execution mode is not going to be switched to managed during execution even when managed execution is faster.

The results of this JSON processing benchmark are shown in Figure 7. While the version where everything is executed in managed mode achieves the highest peak performance already after a few iterations (1.3x), its warm-up time is quite high. In contrast, in the scenario where all the C code of the benchmark is run natively, less JIT compilation leads

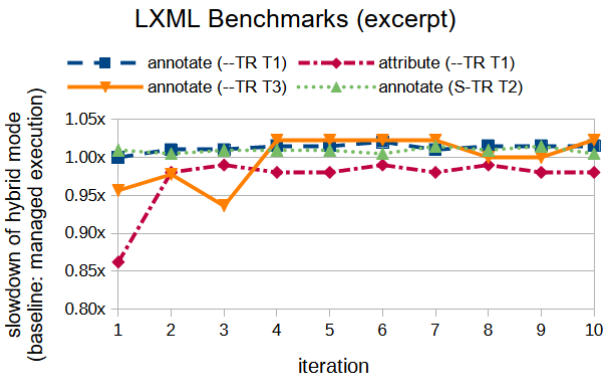


Figure 8. XML parser: relative time of manual hybrid mode. The baseline is managed execution (time measurement → lower is better). The interquartile range is 0.02x on average, 0.021x at max.

to slightly lower peak performance. However, its run time during the first iterations is the shortest (1.7x faster than manual hybrid mode). As expected, when executing only the three functions with most execution time natively, the performance of this hybrid mode is mostly in between the fully managed and the fully native mode. Thus, we can see that mixing native execution with managed execution can achieve a good trade-off between fast warm-up times and high peak performance.

5.2 LXML: Comparing Different Tasks with the Same Classifier

LXML [16] is a Python package for parsing XML data to objects. Similar to the JSON parser in the UJSON benchmark, it uses code of C libraries internally. We decided to use the `bench_objectify` benchmark harness, which can be found in the official source directory [16]. The `bench_objectify` benchmark harness contains different tasks of processing XML data and operating on its structures, and the time for each task is measured. Since execution times of this benchmark are task-specific, the results will be shown as a relative time of our hybrid execution mode, where the baseline (1.0x) is standard managed execution on GraalVM. The measurements of the different tasks all showed an interquartile ranges lower than 0.021x, and 0.02x on average.

Running this harness with our *hybrid execution mode* led to different results without showing a clear tendency. Therefore, we decided to only show four interesting out of the almost 20 different tasks, which can be seen in Figure 8.

Among the different task results, we show one where our *hybrid execution mode* could outperform managed execution (`attribute TR T1`). For this example, performance gains could be made especially during warm-up, as more native code leads to a reduction of JIT compilation at run time. We also included two results where our *hybrid execution mode* could not catch up with managed execution (`annotate TR T1` and `annotate S-TR T2`), although the slowdown is in the range of 2-3% only.

An interesting result, however, is given by the `annotate TR T3` example. During warm-up, our *hybrid execution mode* leads to better results, as the JIT compiler has to analyze less than for managed mode. However, the difference in the analysis also pays off at peak performance, where our *hybrid execution mode* is ≈ 3% slower than managed mode. The `annotate TR T3` example thus shows a similar result as the UJSON benchmark.

Overall, the results indicate that on the one hand, the performance gain of our *hybrid execution mode* depends on the application. On the other hand, also the selection of the functions, i.e., which functions to run natively and which to execute in managed mode, has an important impact on the output. This also shows that our *hybrid execution mode* is not trivial to apply generically to an application. For a better result, more work has to be done for finding a more universal heuristic.

6 Conclusions and Future Work

In conclusion, we introduce the concept of *hybrid execution mode*, a novel way to incorporate AOT-compiled native code into a VM with a standardized JIT compiler. By leveraging the readily available AOT compiled code, it elevates the quality of the code at the warm-up phase of the JIT complication cycle. We also have shown how the addition of a *hybrid execution mode* to GraalVM has the potential to improve its warm-up performance.

Currently, the selection of which code should be executed natively has to be done by hand in order to achieve optimal performance. Therefore, further research is necessary to automatically find classifiers for deciding which functions should be executed natively and which in managed mode. Such classifiers would most likely have an impact on the warm-up performance of the application.

Another limitation of our current approach is the fact that errors can happen at run time if the conversion from a managed object to native memory fails. We therefore need to find a way to identify the failures before they occur and switch back to managed execution. Another option is revoking the native execution when the memory error occurs, and the currently failing code has to fall back to managed execution.

References

[1] 2023. clang: C++ compiler. https://www.clang.org/
[2] 2023. GraalVM. https://www.graalvm.org/
[3] 2023. High-performance Modern Python – run your applications with GraalPy. https://www.graalvm.org/python/
[4] 2023. HPy: a better API for Python. https://docs.hpyproject.org/en/latest/
[5] 2023. PYPL PopularitY of Programming Language index. https://pypl.github.io/PYPL.html

[6] 2023. PyPy – A fast, compliant alternative implementation of Python. https://www.pypy.org/

[7] 2023. Truffle Native Function Interface – Oracle GraalVM for JDK 17. https://docs.oracle.com/en/graalvm/jdk/17/docs/graalvm-as-a-platform/language-implementation-framework/NFI/#truffle-native-function-interface

[8] John Aycock. 2003. A Brief History of Just-in-Time. *ACM Comput. Surv.* 35, 2 (jun 2003), 97–113. https://doi.org/10.1145/857076.857077

[9] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (oct 2017), 27 pages. https://doi.org/10.1145/3133876

[10] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy) *(ICOOOLPS '09)*. Association for Computing Machinery, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827

[11] Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. 2021. YJIT: A Basic Block Versioning JIT Compiler for CRuby. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Chicago, IL, USA) *(VMIL 2021)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/3486606.3486781

[12] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 25–35.

[13] David Flanagan. 2006. *JavaScript: the definitive guide.* " O'Reilly Media, Inc.".

[14] Olivier Flückiger, Jan Ječmen, Sebastián Krynski, and Jan Vitek. 2022. Deoptless: speculation with dispatched on-stack replacement and specialized continuations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM. https://doi.org/10.1145/3519939.3523729

[15] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[16] LXML. 2023. lxml/lxml – The lxml XML toolkit for Python. https://github.com/lxml/lxml

[17] Ross McIlroy. 2016. Firing up the ignition interpreter. https://v8.dev/blog/ignition-interpreter

[18] Oracle. 2023. Graal/Sulong at master · Oracle/Graal. https://github.com/oracle/graal/tree/master/sulong

[19] T. Pittman. 1987. Two-Level Hybrid Interpreter/Native Code Execution for Combined Space-Time Program Efficiency. In *Papers of the Symposium on Interpreters and Interpretive Techniques* (St. Paul, Minnesota, USA) *(SIGPLAN '87)*. Association for Computing Machinery, New York, NY, USA, 150–152. https://doi.org/10.1145/29650.29666

[20] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-Level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages* (Amsterdam, Netherlands) *(VMIL 2016)*. Association for Computing Machinery, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[21] Leszek Swirski. 2021. Sparkplug – a non-optimizing JavaScript compiler. https://v8.dev/blog/sparkplug

[22] Laurence Tratt. 2009. Dynamically typed languages. *Advances in Computers* 77 (2009), 149–184.

[23] Lionel Sujay Vailshery. 2022. Most used languages among software developers globally 2022. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

[24] Guido Van Rossum and Fred L. Drake. 2009. *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA.

[25] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. 13–14.

[26] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Georg Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming and software* (Indianapolis, Indiana, USA). ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

Received 2023-07-23; accepted 2023-08-28