On Automating Hybrid Execution of Ahead-of-Time and Just-in-Time Compiled Code

Christoph Pichler
Johannes Kepler University
Linz, Austria
christoph.pichler@jku.at

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

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

Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

Abstract

The divergence between Ahead-of-Time (AOT) and Just-in-Time (JIT) compilation techniques presents a unique predicament when trying to achieve optimal performance in software applications. AOT compilation offers efficiency by precompiling and optimizing code, while JIT compilation enhances peak performance through dynamic optimization and speculation. However, the improved peak performance achieved by JIT compilation is offset by the poor warm-up performance due to the overhead caused by analyses and optimizations at run time. Previously, we proposed blending these two compilation techniques, aiming to maintain high peak performance while enhancing warm-up performance. Since the programmer had to manually select functions for AOT compilation, it required familiarity with the code base and with compilers in general.

This paper presents a strategy for blending these two compilation techniques automatically. We provide an overview of language implementation features which have to be considered when implementing such an automated approach. We also propose a call-graph based analysis when determining whether certain code should be replaced by its AOT-compiled equivalent.

We implemented our approach within GraalVM, a multi-language virtual machine based on the Java HotSpot VM. The results from different benchmarks show our approach leads to a speedup of 1.48× on average for data setup and up to 2.6× for warm-up and 3.5× for peak performance. Moreover, our automated approach is able to find optimizations which have easily been missed by manual annotations.

This work is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International License.

VMIL '24, October 20, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1213-5/24/10
https://doi.org/10.1145/3689490.3690398

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

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

ACM Reference Format:
Christoph Pichler, Paley Li, Roland Schatz, and Hanspeter Mössenböck. 2024. On Automating Hybrid Execution of Ahead-of-Time and Just-in-Time Compiled Code. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '24), October 20, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3689490.3690398

1 Introduction

There are two common approaches for compiling and executing programs. The first approach is to compile source code into machine code prior to execution, known as Ahead-of-Time (AOT) compilation. The native code is then run directly on the machine. Historically, AOT compilation has been the prevalent method across numerous programming languages. Extensive research and development efforts have been directed towards AOT compilation, leading to compilers that generate highly efficient code. This has significantly contributed to the popularity of languages such as C and C++, which are among the most widely used programming languages today [6, 22]. As a popular example for an efficient C/C++ compiler, clang can be mentioned, which uses the LLVM bitcode format as a low-level, intermediate representation [1].

The second approach is Just-in-Time (JIT) compilation, where execution starts with a lesser optimized, slower version of the code. At run time, the JIT compiler translates the most frequently executed functions into efficient machine code. This process enhances peak performance once the optimized machine code becomes available. A notable benefit of JIT compilation is that it can switch back to the interpreter or recompile code as needed, making it particularly favored for dynamic languages where static analysis is challenging [8].

However, JIT compilation relies on run-time information and thus has the drawback of inferior warm-up performance compared to statically compiled code. Several dynamic languages, such as Ruby or Python, make use of native libraries or native extensions written in C [5, 10]. In such a scenario, executing those C functions via AOT-compiled code enhances warm-up performance compared to relying on interpretation and JIT compilation. However, combining these two distinct worlds introduces certain frictions in their language and run-time models. As an example, certain data accesses can prevent code to get pre-compiled.

In our previous work, we provided a proof of concept for combining AOT and JIT compilation, which can achieve both fast warm-up and high peak performance [16]. That approach, however, required the programmer to manually select functions for AOT compilation. Our goal in this paper is to automate the process of selecting functions for AOT compilation, which eliminates the need for programmers to annotate their programs. This has not only the advantage that programmers need less knowledge about their code base, but also allows for more sophisticated selection heuristics. We implemented our approach of an automated hybrid execution model in GraalVM, a polyglot ecosystem that comes with a highly optimizing JIT compiler. Based on a heuristic, code can either be compiled to machine code ahead of time and executed directly, or it can be interpreted and JIT-compiled by the GraalVM compiler.

Our main contributions are:

- We investigate what needs to be taken into account when building an ecosystem that blends different execution modes (Section 3).
- We propose a heuristic for making decisions on hybrid execution (Section 4) and evaluate it (Section 5).
- We discuss current limitations of our approach (Section 6).

We finally conclude our work in Section 7.

2 Background

In this section, we discuss ahead-of-time and just-in-time compilation within the GraalVM ecosystem, as well as their respective properties.

2.1 LLVM

LLVM is a cross-platform compilation framework. Its low-level intermediate representation, LLVM IR, is used by various compiler front-ends and back-ends and uses static typing and single static assignment (SSA) form [12]. Usually, LLVM IR is compiled ahead of time (AOT). One popular example is the clang front-end, which outputs LLVM IR from which efficient machine code can be generated [1]. LLVM IR also comes with a compact bitcode format, LLVM bitcode.

2.2 Just-in-Time (JIT) Compilation

Just-in-Time (JIT) compilation usually comes in a two-tier architecture, where lesser optimized code is executed when the application is started. Meanwhile, JIT compilation optimizes and compiles frequently executed code, known as *hot* code, into efficient machine code. Execution can switch to the higher-optimized code once it is has been produced by the JIT compiler. This process involves the JIT compiler focusing its optimization efforts on the most frequently executed sections of code, while, for example, an interpreter collects performance data throughout execution to enhance code efficiency. Leveraging runtime profiling information allows the JIT compiler to do more advanced optimizations than those possible in AOT compilation. However, this approach initially slows down execution speed due to the need for JIT compilation and its various analyses [8].

Dynamically-typed languages such as Python and JavaScript often make use of a managed runtime with a JIT compiler. This presents some unique challenges to their performance [19]. Since these languages lack static type information, variables can hold values of different types, complicating the JIT compiler's ability to perform static type analysis and apply traditional compiler optimizations. This makes type checks and method dispatches necessary at run time, affecting optimization opportunities and introducing additional run-time overhead. As a consequence, dynamic languages such as Python or Ruby often make use of native extensions [10].

In general, various approaches aim to balance fast warm-up with high peak performance. Many JIT compilers, such as Google's V8 engine [18] and the GraalVM JIT compiler [2], implement varying levels of optimization, applying fewer optimizations initially and increasing them as functions become more frequently executed. Another strategy involves reusing profiling data from prior executions to enhance performance [14].

2.3 GraalVM

GraalVM is a high-performance and polyglot virtual machine designed for applications written in Java, JavaScript, Ruby, Python, C, and many other languages [2, 24]. Its ecosystem is shown in Figure 1. The Java HotSpot VM serves as a basis, which can also use the aggressively and dynamically optimizing GraalVM JIT compiler.

2.3.1 Truffle Language Implementation Framework.

To be able to implement runtimes for different languages, there is the Truffle language implementation framework on top of the Java Hotspot VM and the JIT compiler [23], which also enables direct interoperability between Java and other supported languages. With Truffle, self-optimizing language interpreters can be built, which are based on abstract syntax trees (ASTs) or bytecode. The framework is built in a way such that the GraalVM JIT compiler can produce highly

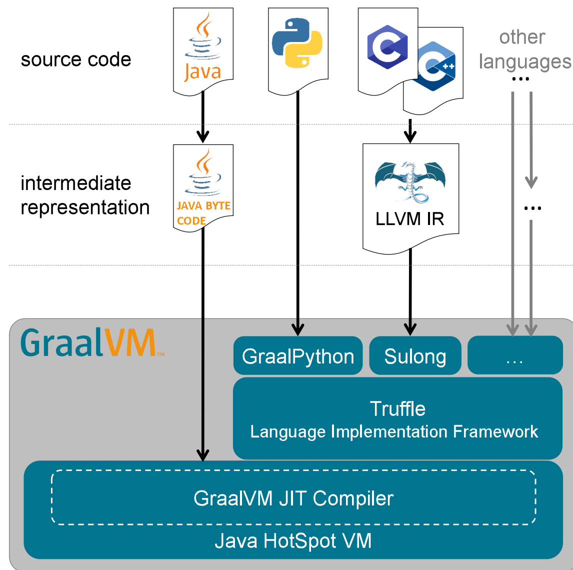


Figure 1. GraalVM and its polyglot execution engine

optimized machine code from Truffle's intermediate representations.

An example of a Truffle-based runtime is GraalPython [3], which serves as a runtime for Python source code. Due to the GraalVM JIT compiler, GraalPython outperforms the reference implementation CPython [3].

Since different language interpreters are based on Truffle, it is possible to perform cross-language calls within this framework. As all interpreters are based on the same framework, cross-language execution does not inflict any language barrier overhead in GraalVM after the JIT compiler has emitted machine code. Also, optimizations of the JIT compiler can easily be performed across different languages [23, 24].

2.3.2 Sulong. Sulong [15, 17] is another example of a runtime based on Truffle. It is the runtime for executing LLVM bitcode, as shown in Figure 1. As C/C++ code can be compiled to LLVM bitcode using the clang compiler [1], Sulong can be seen as the managed runtime for C/C++ source code on GraalVM.

To conveniently work with Sulong, it has its own toolchain, which is based on clang. It first compiles C/C++ code to LLVM bitcode, which is then further compiled to machine code. The Sulong toolchain, however, also adds the LLVM bitcode as a special section to the generated machine code file. This section can later be used by GraalVM to execute this code in managed mode.

2.3.3 Truffle Native Function Interface. The Native Function Interface (NFI) within Truffle facilitates the direct invocation of AOT-compiled functions within a Truffle runtime, as shown in Figure 2. This interface acts as a conduit between the managed execution environment provided by

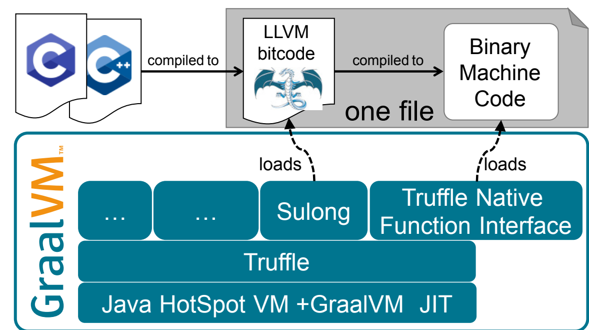


Figure 2. Two ways to execute C code on GraalVM

GraalVM and native execution capabilities. Its primary usage is to execute code from native libraries, system APIs, or performance-critical segments written in languages outside the Truffle framework [7]. The NFI thus allows the invocation of native C/C++ functions.

2.4 C Code on GraalVM

The two common ways to execute C code on GraalVM are shown in Figure 2 and will be explained now.

- Managed Execution via Sulong: This is the default execution mode for C/C++ source code on GraalVM. As described above, the Sulong toolchain is used to compile C/C++ code to a file containing both LLVM bitcode and machine code. If Sulong receives such a file with an LLVM bitcode section, it creates a Truffle AST from the LLVM bitcode, which is then forwarded to the GraalVM JIT compiler, as shown in Figure 2.
- Native Execution via the Truffle NFI: If Sulong receives a file without an LLVM bitcode section, it uses the Truffle NFI to execute the compiled machine code directly, as shown in Figure 2. In contrast to managed execution, cross-language calls are not possible in native mode, as no Truffle data structure is built to represent the program. Also, there is no optimization by the GraalVM JIT compiler involved [7].

2.4.1 Native and Managed Global Data. When combining managed and native execution, it is important to consider how and where global data objects are stored and accessed. We described different options how such accesses can happen in GraalVM [16]:

Functions in managed mode can access data stored in both, managed and native heap. Also, functions in native mode can directly access data in native heap.

The interesting situation happens when a native function wants to access managed data: Granting direct access to the managed data would violate language semantics, as described in detail in [16], and is therefore not possible. The way how such an access continues depends on the managed data itself:

- If the managed data can be easily transformed to a native representation, for example, when it is a C struct, then the managed data is transformed to native memory. By that, functions both in managed and native mode can access the transformed data without any problems. To avoid unnecessary work, this conversion to native memory is done lazily at the first access of a native function [16].
- If the managed data cannot be transformed to native memory, for example, when it originates from a foreign language such as JavaScript, then access cannot be granted, and execution cannot continue [16]. Thus, if managed data cannot be converted to native memory, we have to make sure to execute all accessing functions in managed mode as well. Note that since the attempt to convert managed data to native memory happens only when this data is accessed, pointers to managed data can also be passed to native functions, as long as the data itself is not accessed.

To summarize, if certain managed data is accessed by a function, then this function must be executed in managed mode. Therefore, it is not possible to execute every C function in native mode.

2.4.2 Manual Hybrid Execution. C code can be run on GraalVM in managed mode via Sulong or in native mode via the Truffle NFI. The managed execution under Sulong comes with a rich toolset and many opportunities for optimization at run time, but has the drawback of long warm-up times. In contrast, native execution does not offer access to interoperability or dynamic optimization, but enables a fast start-up by not requiring JIT compilation.

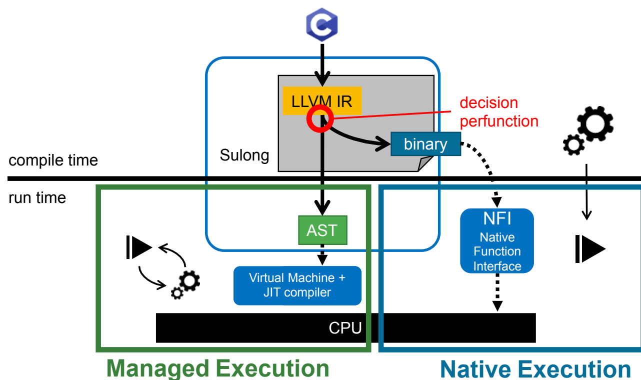


Figure 3. Hybrid execution mode as proposed in [16]

Therefore, we proposed to combine those two execution modes: Instead of letting the JIT compiler analyze the whole program, which is shown on the left-hand side in Figure 3, existing native code is used to speed up the warm-up process of GraalVM [16]. This is done by Truffle's Native Function Interface (NFI), which calls native code that has already been compiled ahead of time (shown on the right-hand side in Figure 3).

To mix native and managed execution, we suggested to modify the (managed) runtime: By default, a function call in managed execution results in the callee being executed in managed mode, while in native execution, the callee is executed in native mode. The approach of our previous work is to modify the managed runtime such that a caller in managed mode can also call the callee's native code, as indicated by the red line in Figure 4 [16].

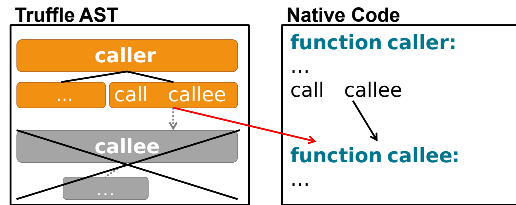


Figure 4. By default, execution mode of caller and callee are identical. However, managed call instructions can be redirected to native execution of the callee [16].

However, it is not possible in our previous approach to call a managed function from native code [16]. Therefore, all calls from a function in native mode result in the callee being executed in native mode as well. This introduces a major limitation, since functions which access managed data must be executed in managed mode. Even worse, all direct and indirect callers of functions that access managed data cannot be executed in native mode as well.

The decision of which code to execute in which mode, is taken per function and has to be specified manually in our previous approach [16]. This manual decision is another major limitation, as the programmer is required to know the code base and its internal dependencies.

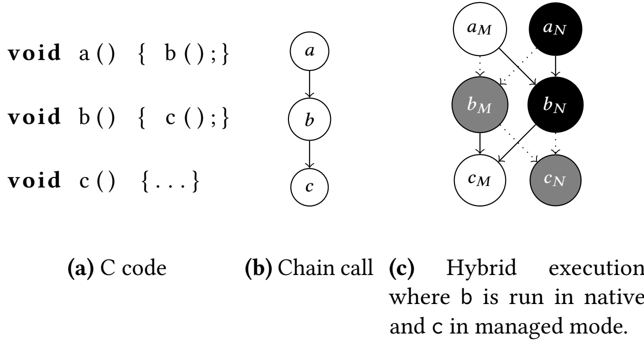
Thus, the goal of our work is to extend the possibilities of hybrid execution and improve this decision process per function, which will be covered in the next sections.

3 Hybrid Execution

In this section, we explore the principles of our hybrid execution mode and resulting circumstances with regard to implementation.

In general, our hybrid execution mode offers for each function both managed execution via Sulong, and native execution via the Truffle NFI. A simple example with three functions is shown in Figure 5a and 5b.

Each function exists in two modes and can thus be executed in managed mode (e.g. a_M) or in native mode (e.g. a_N). When b is set to run in native mode, managed callers of b (in our case: a_M) call the native version of b (b_N) instead of the managed version b_M. Native callers of b (a_N) do not need a modification, as they call the native version of b by default.



(a) C code **(b)** Chain call **(c)** Hybrid execution where b is run in native and c in managed mode.

Figure 5. Hybrid Execution and its basic concept

Similarly, managed execution of c needs a change in the native callers of c. The native caller b_N has to forward the call to the managed runtime, such that c_M is called instead of c_N. The final result can be seen in Figure 5c, where solid lines show actual calls, and dotted lines show possible, but replaced calls.

3.1 Execution Mode Dependencies

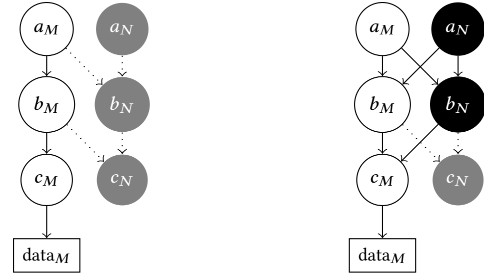
As explained in Section 2.4.1, not every function can be executed in native mode. The major reason why a C function must be executed in managed mode is that it accesses managed data which cannot be converted to native memory. For example, assume that function c of Figure 5 accesses managed data, as shown in Figure 6. This access requires c to be executed in managed mode, i.e., native execution of c is not possible.

In our previous approach, it is not possible to call managed functions from native callers [16]. In other words, native callers can only call native callees. In the scenario of Figure 6a, this leads to a major restriction: Managed execution of c requires all of its direct and indirect callers (a and b here) to be also executed in managed mode. Thus, our previous hybrid execution model is rather limited.

In contrast to our previous approach, we now allow managed functions to be also called from native callers, which opens more opportunities for hybrid execution. In the example of Figure 6b, a and b can also be executed in native mode, even if c is restricted to managed execution.

3.2 Calling Managed Functions from Native Code

When native code is linked and loaded dynamically, as it happens with the Truffle NFI, function calls cannot be resolved at compilation time. Instead, the compiler replaces every function call by a look-up into a so-called procedure linkage table (PLT) [11]. This table is created and filled with placeholder values when native code is loaded. The first time a function is called, its address is resolved and stored into the PLT. For example, `caller` in Figure 7 calls `callee`, and a jump instruction to the resolved address of `callee` (shown by the dotted gray line) is written into the table. Subsequent



(a) Execution model in [16]: Managed execution of c implies managed execution of a and b. Thus, no hybrid execution in this case.

(b) Our execution model: Although function c is managed, a and b can be executed in either mode (native / managed).

Figure 6. Hybrid execution models for function a calling b, b calling c, and c accessing managed data. Dotted lines and gray nodes indicate that the call/method cannot be used.

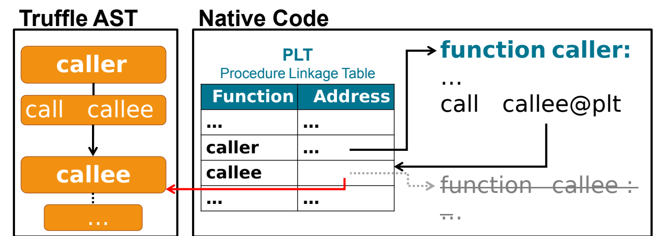


Figure 7. A change in the procedure linkage table (PLT) can make native callers call managed callees.

calls to `callee` then only need a jump instruction to the table entry, which then invokes the corresponding callee.

This indirection can be used to trigger managed execution from native execution: Instead of resolving the address of the callee in native mode, we modified the Sulong runtime such that execution automatically resolves the address of the same function in managed mode. Sulong then stores a jump to this address into the PLT, such that subsequent calls also invoke the callee in managed mode. This injection of the address of the function in managed mode is indicated by the red arrow in Figure 7.

In contrast to changes within GraalVM, this modification of native code is platform-dependent. Therefore, we implemented this patching of the PLT only for code on x86-64 machines as a proof of concept. Other architectures require a similar procedure.

By being able to call managed functions from native code, we eliminate one of the major limitations of our previous hybrid execution approach [16], as shown in Figure 6a.

3.3 Symbol Resolution

Another aspect to consider is how symbol resolution happens in native code: The order in which native libraries are loaded and initialized might differ from the managed loading order,

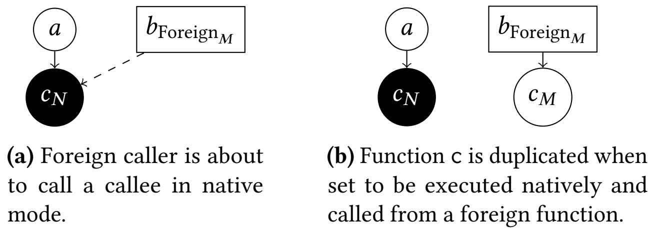
especially if some libraries are not used for native execution at all. In our previous approach, the programmer manually decided in which mode functions are executed and whether global variables are stored in native or managed memory [16]. Thus, the programmer can also implicitly control the loading order of libraries and can detect or prevent errors resulting from library dependencies.

For an automated decision, this is not possible any more. Thus, symbol resolution has to be aware of the fact that symbols might be about to be resolved when their target library has not been loaded yet.

To overcome this problem, we modified the loading of libraries such that native symbols are resolved lazily. In other words, native symbols are only resolved at their first use, instead of when their library is loaded.

3.4 Foreign Caller calls Native Callee

In our context, managed functions are C functions executed on Sulong. However, there might be other managed functions (e.g., Python functions) that are not executed on Sulong. We call such functions foreign functions. As mentioned in Section 2.3.1, it can happen that a C function `c` is called by such a foreign function $\mathrm{b}_{\mathrm{foreign}}$ and that our automated heuristics recommend to run `c` in native mode, as shown in Figure 8a.



(a) Foreign caller is about to call a callee in native mode.

(b) Function `c` is duplicated when set to be executed natively and called from a foreign function.

Figure 8. Functions may get duplicated — Call structure (Fig. 8a) and a possible execution scenario (Fig. 8b)

However, it is not possible to simply call a native C function directly from a foreign language, as the Sulong runtime performs extra work such as adding context parameters. In such a situation, we create a duplicate version of `c` on demand: If invoked by a managed caller `a` via Sulong, `c` runs in native mode as recommended. If invoked by a foreign caller b_{Foreign} (not via Sulong), the managed version of `c` is called, as depicted in Figure 8b.

Unlike the situation where managed data is accessed from native execution, detecting when a native function is called from a foreign function can be done at run time. This means that there is no need to detect such calls ahead of time. Instead, the callee's code can be dynamically replaced at run time if an error occurs during the method call.

4 Setting up Heuristics for Hybrid Execution

The previous section described the principles and circumstances of interaction between managed and native execution. In this section, we describe an automated approach for determining whether certain functions or variables should be run, or stored, in managed or native mode.

4.1 Tracking Function Calls

When a C function is called, there is the choice whether it should be executed in native or in managed mode. To obtain data for this decision, we added a simple instrumentation to the Sulong runtime and perform a run of the program under Sulong in pure managed mode. By that, every call to a function and every access to a variable is tracked.

With that data, we can build a call graph of parts of the executed application. Every node represents a function or a variable, while a directed edge represents a function call or an access to a variable. A simple example has already been shown in Figure 5, where the code in Figure 5a has been transformed to the call graph in Figure 5b.

In constrast to a call tree, where one function is represented as multiple nodes if it is called from more than one call site, we decided to build a call graph. Thus, every function only has one corresponding node in the graph.

The following subsections show how such a call graph can be used to design heuristics for hybrid execution in order to improve the performance of a program.

4.2 Execution Time Estimation for Single Functions

Since our heuristics determines the execution mode for each function, one important aspect is the execution time of a function in managed mode and in native mode. The most accurate way to get the execution time is to measure it. However, we do not really measure the execution time of each function in managed and native mode, but rather estimate it based on static code size and control flow information.

4.2.1 Execution Time Estimation Without Control Flow. In general, the execution time of a function depends on the number and time of executed instructions. Ignoring control flow (loops and branches) initially, there is linear dependency between the code size of a function and its execution time. To validate this, we performed measurements with functions of different sizes that do not contain loops or branches.

The measurements consist of micro-benchmarks, where functions with different sizes of LLVM bitcode are called from Sulong. This also means that the measurements for native execution of the called function include the call overhead for the call from Sulong to native execution. Although this overhead will be considered in more detail in Section 4.3, we decided to include it in these measurements as well: Setting

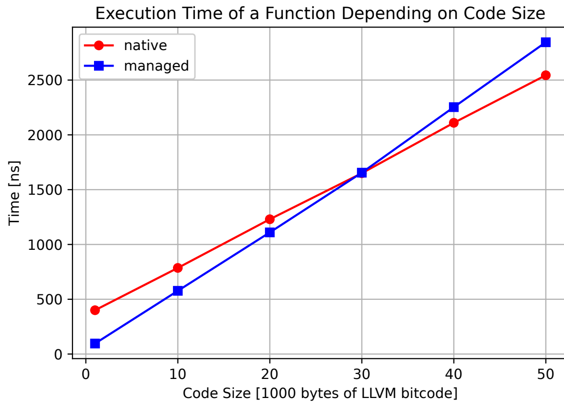
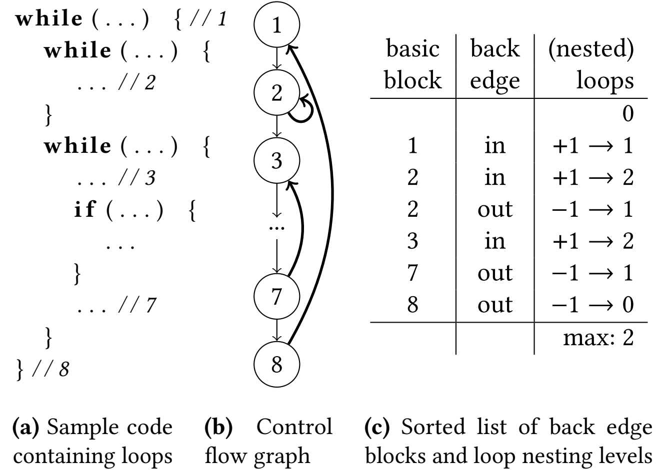


Figure 9. Execution time of a function called from Sulong without branches and loops, based on its code size of LLVM bitcode



basic block	back edge	(nested) loops
		0
1	in	$+1 \rightarrow 1$
2	in	$+1 \rightarrow 2$
2	out	$-1 \rightarrow 1$
3	in	$+1 \rightarrow 2$
7	out	$-1 \rightarrow 1$
8	out	$-1 \rightarrow 0$
		max: 2

(a) Sample code containing loops **(b)** Control flow graph **(c)** Sorted list of back edge blocks and loop nesting levels

Figure 10. Calculating the maximum level of nested loops

a function to native mode will still have the effect of causing boundary costs.

The results are shown in Figure 9, which suggests a break-even point at a code size of ≈ 30 kB. Therefore, we suggest native execution for functions with an LLVM bitcode size larger than 30 kB.

4.2.2 Control Flow Considerations.

As mentioned above, control flow can heavily influence the execution time of a function: While branches (e.g. `if` instructions) skip parts of the code during execution, loops lead to multiple execution of the same code. Therefore, the proposed heuristics of taking 30 kB of code size as a threshold is too inaccurate.

Branches affect the ratio of executed code by less than 50 % on average, which is an amount we ignore for our heuristics. However, loops that are executed multiple times can change the amount of executed instructions by more than one order of magnitude. Therefore, we have to include information about loops into our heuristics.

As our code base is LLVM bitcode, which is structured as a linearized control flow graph with basic blocks, it is not trivial to detect loops and the maximum loop nesting level out of the box. Therefore, we look at back edges in the control flow graph of the LLVM bitcode, and assume every back edge to indicate a loop. Back edges can be found easily, as every basic block in LLVM is annotated with an ascending block index. For example, Figure 10a shows source code containing loops. The corresponding control flow graph is shown in Figure 10b, where back edges are shown in bold.

When parsing the bitcode, we traverse the control flow graph and check every jump between basic blocks. If the block index of the target block is smaller than the index of the current block, we have found a back edge. Each block with an incoming or outgoing back edge is stored in a sorted list (cf. left column of Figure 10c), which is then traversed from top to bottom. Every incoming back edge means that a loop is entered, so we increment the loop nesting level. Conversely, every outgoing back edge decrements the loop nesting level, as shown in the right column of Figure 10c. The maximum number during traversing this list is therefore the maximum loop nesting level of the corresponding code.

For a precise calculation of the execution time, one would need the code size inside the loop as well as the number of loop iterations during execution, which is not decidable by static code analysis. Since we only deal with heuristics, we simplify our approach and assume that each loop level causes the function to run 10 times as long as without loops. Although we are aware that this factor 10 varies for different programs and even varies for different loop levels within one program, assuming 10 as a multiplication factor for each level turned out to be a reasonable choice.

Therefore, our heuristics which sets large methods to native execution changes from from $s(f) > 30\,\text{kB}$ to $s(f) \cdot 10^{\text{loopDim}} > 30\,\text{kB}$, where `loopDim` denotes the maximum loop nesting level. For example, if the function in Figure 10 had an LLVM bitcode size of 400 B, the size would be estimated to be $400\,\text{B} \cdot 10^2 = 40\,\text{kB}$. While the size of f itself (400 B) would be too small to trigger native execution, the loops in f make it large enough to suggest native execution.

4.2.3 Effect on Warm-Up Performance.

Although looking on code size rather than investigating how single functions affect warm-up performance, we assume larger functions to cause a higher compilation effort. By setting large functions to native execution, we reduce the effort for JIT compilation and thus expect a better warm-up performance.

4.3 Function Call Overhead of a Single Function

Although the execution time of a function, as described in Section 4.2, is important, the costs of the function call overhead also have to be taken into account. Especially when a

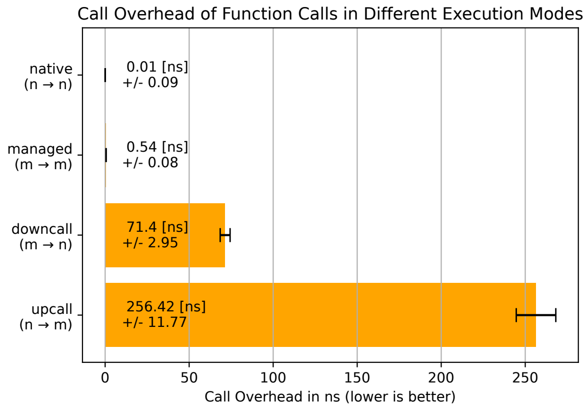


Figure 11. Function call overhead in different execution mode scenarios

managed function calls a native function and vice versa, context information of the managed runtime has to be added or removed, respectively. Thus, we also ran micro benchmarks which measure the function call overhead within and across execution modes. The results, depending on the execution mode of the caller and the callee, are shown in Figure 11.

When both the caller and the callee are executed in the same execution mode, a function call typically involves not more than a couple of machine instructions. Thus, the time for calling and returning from a method can be neglected in this case.

However, a downcall, which is calling a native function from a managed caller, significantly increases the call and return overhead by two orders of magnitude compared to purely managed execution. An even higher overhead is imposed by an upcall, which is when a native function calls a managed callee. Therefore, it is necessary to take a look at function call overheads when assigning execution modes to functions. Otherwise, the worst case could be a scenario where execution switches between native and managed mode back and forth, causing high overhead costs.

As a first step, we take a look at the call overhead for every function `f` that has no hard constraints of being executed in native or managed mode. Call overheads happen due to callers and callees of `f`, where both (callers and callees) can be executed in native or in managed mode. Combining these possibilities leads to four different cases: We can distinguish between managed and native functions on the one hand, and callers and callees of `f` on the other hand, as shown in Figure 12. The edge weights (in_m, out_m, in_n, out_n) denote the absolute frequencies of the calls.

If `f` is executed in native mode, we get the following costs, where d and u denote the costs of a downcall and upcall, respectively: Managed callers of `f` perform a downcall, thus the costs are $in_m \cdot d$. Managed callees of `f` cause an upcall, thus the costs are $out_m \cdot u$. Native callers and native callees

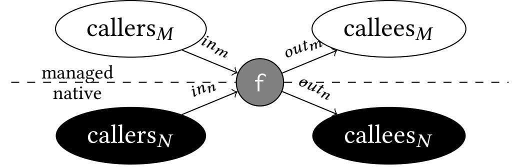


Figure 12. Function `f` can be executed in managed mode as well as in native mode. Since overhead costs for upcalls and downcalls of `f` also influence execution time, they are taken into account.

hardly cause call overheads (cf. Figure 11) and are neglected. For callers and callees that have no execution mode assigned yet, we assume that half of the calls will be native and half will be managed. Therefore, the amounts of managed calls are increased by half of the amount of the undetermined calls (named as in_u and out_u). Thus, the call overhead costs when `f` is native can be computed as in Equation 1.

$$t_{\text{call}}(f, n) = \left(in_m + \frac{in_u}{2}\right) \cdot d + \left(out_m + \frac{out_u}{2}\right) \cdot u \qquad (1)$$

If `f` is executed in managed mode, we get the following costs: Native callers of `f` perform an upcall ($in_n \cdot u$), whereas native callees of `f` cause a downcall ($out_n \cdot d$). Managed callers and callees can be neglected for our heuristics. Half of the undetermined calls are added to the native calls this time. Thus, the call overhead costs when `f` is managed can be computed as in Equation 2.

$$t_{\text{call}}(f, m) = \left(in_n + \frac{in_u}{2}\right) \cdot u + \left(out_n + \frac{out_u}{2}\right) \cdot d \qquad (2)$$

4.4 Heuristic

From the above observations, we infer the following heuristics for a hybrid execution mode: First, we mark all functions that access managed data for managed execution. Then, we mark all functions that are large enough for native execution. Finally, we set the execution mode of the remaining functions such that the call overhead is as low as possible.

In detail, the heuristics look as follows: Given F as the set of all C functions and F_M and F_N as the set of all C functions in managed and native mode respectively, heuristics for an automated hybrid mode can be seen as a partitioning function $F \longrightarrow \{F_M, F_N\}$. The heuristics work as follows:

1. Mark all C functions with managed parameter types or managed return types for managed execution.
2. For each remaining C function f, check if f is large enough for native execution (cf. Section 4.2). I.e., if $s(f) \cdot 10^{\text{loopDim}(f)} > 30$ kB, mark f for native execution.
3. Add all neighbors (callers and callees) of functions marked as managed (step 1) or native (step 2) to a queue Q. While Q is not empty:

a. Remove a function from Q and calculate $t_{\text{call}}(f, n)$ and $t_{\text{call}}(f, m)$ as in Equations 1 and 2.
b. If $t_{\text{call}}(f, m) > t_{\text{call}}(f, n)$, i.e., if the call overhead is higher for managed execution, then set f to native execution. Else, mark f for managed execution.
c. Add all neighbors (i.e., callers and callees) of f without an assigned mode to Q.

5 Evaluation and Results

In our hybrid execution mode, all accesses to managed variables that cannot be converted to native memory are required to happen in managed mode. Therefore, we aim for a scenario where the primary application is written in a high-level language such as Python but uses native extensions written in C, such as an XML or JSON parser. While managed objects are indeed accessed by C/C++ functions, which thus have to be executed in managed mode, there are also C/C++ library functions which can run in native mode.

Although Python comes with a standard performance benchmark suite [4], we did not select these benchmarks, as our hybrid execution mode operates on C source code and requires native extensions.

Both of the following benchmark suites were executed on an x86-64 machine with a 6-core 12th Gen Intel Core i7-1255U, using Ubuntu 22.04 and GraalVM 24. Concerning language versions, we used Java 23, Python 3.10 and LLVM 16.

5.1 LXML: Results Depend on Code

As a benchmark suite, which was also used in our previous work [16], we decided to use the LXML Python package for parsing XML data to objects [13]. This python package also comes with a benchmark harness. It first parses and allocates XML data structures in memory during its setup process. Then over several iterations, the parsed XML tree is modified in roughly 20 different tasks. While the main program of the benchmark is written in Python, the tasks themselves are written as native extensions, which is what we are interested in.

To evaluate our automated hybrid execution mode, we are interested in the following questions: How does our hybrid execution perform with respect to:

1. The particular allocation of objects and variables, i.e., whether they are allocated in native or in managed memory? For this question, we measured the total data setup time.
2. The warm-up performance of the different benchmark tasks?
3. The peak performance of the different benchmark tasks?

For all three questions, we used fully managed execution on Sulong as the baseline and compared it to the execution times of our automated hybrid execution mode, as well as

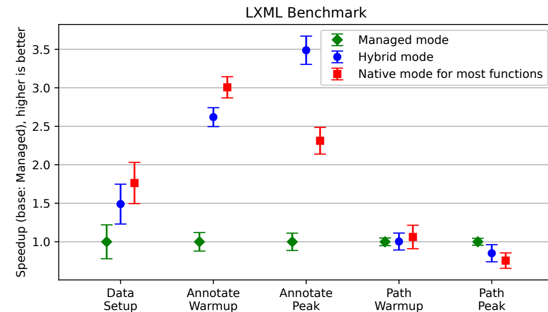


Figure 13. Results of LXML Objectify Benchmark

a mode where all functions are native that do not access managed data. We grouped the tasks into annotating tasks (which perform modifications on the XML tree) and tasks that try to find certain paths within the tree.

Before we take a look at the results, let us take a look at the outcome of the heuristic: Concerning global variables, all C variables are stored in native memory, while 22 % (804 out of 3637) of all C functions are selected for native execution.

The results of the three measurement questions from above are shown in Figure 13, where the speedup of our hybrid execution mode is shown in blue. The results where all possible candidates for native execution are executed in native mode are shown in red. Given that the end of the warm-up phase of VMs is not always easy to determine [9], we see that similar benchmarks using the GraalPython runtime need about five to ten iterations to warm up, which corresponds to our measurements.

One can see that storing C variables in native memory leads to a speedup of 48 % for the setup time of the given benchmark on average. Thus, for question 1, we see an advantage of hybrid execution mode compared to managed execution. Concerning warm-up and peak performance of the LXML benchmark, the answers to questions 2 and 3 highly depend on the kind of task: On the one hand, tasks that perform annotations on the XML tree achieve an average speedup of 2.6× for warm-up and even 3.5× for peak performance in hybrid mode. Compared to the baseline (all functions executed in managed mode on Sulong), executing some functions in native mode clearly pays off. Running as many functions in native mode as possible implies even less optimization work for the JIT compiler, which leads to a slightly higher speedup for warm-up, but then results in a slower peak performance.

For the tasks that aim to find certain paths within the XML tree, the situation is different: While the warm-up performance in hybrid mode does not show a significant difference to managed execution, there is a small drop in peak performance with a slowdown of 0.85×, and an even worse drop for running as many functions natively as possible. In this case, native execution of certain functions did not speed up

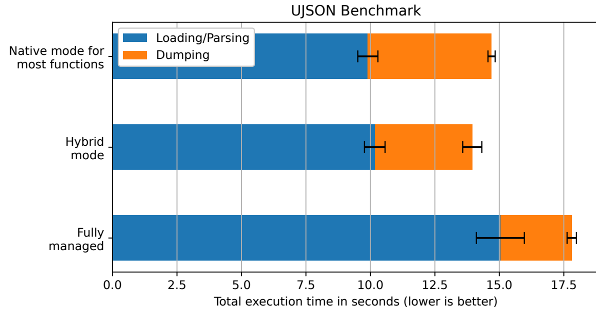


Figure 14. Results of the UJSON Benchmark

the execution process. A possible reason is that the GraalVM JIT compiler is able to optimize the managed functions better than if they were executed in native mode, e.g. since managed execution enables the GraalVM JIT compiler to inline managed data accesses.

The results indicate that the performance of hybrid execution highly depends on the executed code, as well as on the data dependencies. This goes with the findings in our previous work, where the execution mode for functions and variables was manually chosen by the programmer [16]. The results of the same benchmark in our previous work also strongly differ based on the exact subtask of the benchmark. However, the speedup there was only 1.7× for warm-up performance, and close to none for peak performance [16]. On the one hand, this comes from the fact that in our previous work native functions cannot call managed functions, as we do now (see Section 3.1 and Figure 6). On the other hand, it also shows that an automated heuristic can find improvements for hybrid execution, while manual selection as in our previous work comes with a high manual effort and might miss certain optimizations.

5.2 UJSON: How Hybrid Mode Can Achieve Speedups

The basic principle how our hybrid execution mode can achieve speedups can be seen with the benchmark available in the GitHub repository of the python UltraJSON package [20, 21]. The package itself offers efficient loading, parsing and dumping JSON data. The benchmark loads and parses JSON data as a first task, which is then dumped again in a second task. The time of both tasks is measured within 10 fork runs, where each performs 500 iterations of loading and dumping. Our heuristics as described in Section 4 proposed 76 % (725 out of 949) of the functions to run in native mode. We ran this benchmark in three modes: Fully managed (1), our hybrid execution mode (2), and a mode where as many functions as possible are executed in native mode (3).

The results are shown in Figure 14. While managed execution on Sulong is faster for dumping, (mostly) native execution is faster for loading and parsing. Concerning execution times, our hybrid mode, which combines native and managed execution, is between managed and native execution. However, for the total time of both tasks, our suggested hybrid execution mode strikes a balance and is faster than fully managed or close-to-native execution.

6 Current Limitations and Future Work

Our current hybrid execution mode can lead to a high overall speedup. However, the speedup for a single function highly depends on its code. While some functions lead to a speedup when executed natively, for others, the optimizations due to JIT compilation overtake any speedup caused by native execution. Therefore, further research is necessary to more closely investigate the relationship between functions and the impact on performance when automated hybrid execution is enabled, which can also include linear programming or machine learning techniques. Being able to predict the impact of hybrid execution on warm-up and peak performance more precisely enables better overall results.

Also, the approach presented in this paper tries to minimize execution time during warm-up. Other interesting metrics to look at are the time spent on JIT compilation, or code and memory footprints.

Another possible future research is to reduce the risk of errors at run time when functions in native mode access managed data that cannot be converted to native memory. To address this, we must devise strategies for detecting these failures preemptively and for reverting such functions to managed execution. As an alternative, especially if such failures cannot be detected preemptively, we could rewind native execution upon encountering a memory error, forcing the problematic code to resume under managed execution.

Related to this, being able to change the execution mode of a function at run time can also enhance the peak performance in hybrid mode: If native mode is detected to be slower than managed mode for a function (see the path finding tasks in the LXML benchmarks in Figure 13 or dumping JSON data in Figure 14), the execution mode can be switched to managed mode at run time such that the GraalVM JIT compiler can optimize aggressively. Also, important insights may also be applied to more general settings where functions are executed in different contexts, such as heterogenous computing.

7 Conclusion

In this paper, we presented a hybrid execution environment that determines which functions of a program should run in managed mode, and which should run in native mode. Our system executes AOT-compiled native code via NFI and managed code in a VM using a JIT compiler. Our evaluation shows that integrating an automated hybrid execution mode into GraalVM can improve the performance of mixed-mode programs.

References

[1] 2024. clang: C++ compiler. https://www.clang.org/
[2] 2024. GraalVM. https://www.graalvm.org/
[3] 2024. High-performance Modern Python – run your applications with GraalPy. https://www.graalvm.org/python/
[4] 2024. The Python Performance Benchmark Suite. https://pyperformance.readthedocs.io/
[5] 2024. Python/C API Reference Manual. https://docs.python.org/3/c-api/index.html
[6] 2024. TIOBE Index for July 2024. https://www.tiobe.com/tiobe-index/
[7] 2024. Truffle Native Function Interface – Oracle GraalVM for JDK 21. https://docs.oracle.com/en/graalvm/jdk/21/docs/graalvm-as-a-platform/language-implementation-framework/NFI/
[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] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically composing languages in a modular way: Supporting C extensions for dynamic languages. In *Proceedings of the 14th International Conference on Modularity*. 1–13. https://doi.org/10.1145/2724525.2728790
[11] Michael Hicks, Stephanie Weirich, and Karl Crary. 2001. Safe and Flexible Dynamic Linking of Native Code. In *Types in Compilation*, Robert Harper (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 147–176. https://doi.org/10.1007/3-540-45332-6_6
[12] 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
[13] LXML. 2024. lxml/lxml – The lxml XML toolkit for Python. https://github.com/lxml/lxml
[14] Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. 2023. Reusing Just-in-Time Compiled Code. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 263 (oct 2023), 22 pages. https://doi.org/10.1145/3622839
[15] Oracle. 2024. Graal/Sulong at master · Oracle/Graal. https://github.com/oracle/graal/tree/master/sulong
[16] 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* (Cascais, Portugal) *(VMIL 2023)*. Association for Computing Machinery, New York, NY, USA, 39–49. https://doi.org/10.1145/3623507.3623554
[17] 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
[18] Leszek Swirski. 2021. Sparkplug – a non-optimizing JavaScript compiler. https://v8.dev/blog/sparkplug
[19] Laurence Tratt. 2009. Dynamically typed languages. *Advances in Computers* 77 (2009), 149–184. https://doi.org/10.1016/S0065-2458(09)01205-4
[20] UJSON. 2024. UltraJSON – Ultra fast JSON encoder and decoder for Python. https://pypi.org/project/ujson/
[21] UJSON. 2024. ultrajson/ultrajson – Ultra fast JSON decoder and encoder written in C with Python bindings. https://github.com/ultrajson/ultrajson
[22] Lionel Sujay Vailshery. 2022. Most used languages among software developers globally 2022. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/
[23] 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* (Tucson, Arizona, USA) *(SPLASH '12)*. Association for Computing Machinery, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723
[24] 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