DOI: 10.15514/ISPRAS-2025-37(6)-7

Fast Calls and In-Place Expansion: A Hybrid Strategy for VM Intrinsics

D.V. Zavedeev, ORCID: 0009-0009-1477-5249 <zdenis@ispras.ru>
R.A. Zhuykov, 0000-0002-0906-8146 <zhroma@ispras.ru>
L.V. Skvortsov, ORCID: 0000-0002-1580-1244 <lvs@ispras.ru>
M.V. Pantilimonov, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. This research proposes a hybrid approach for implementing performance-oriented compiler intrinsics. Compiler intrinsics are special functions that provide low-level functionality and performance improvements in high-level languages. Current implementations typically use either in-place expansion or call-based methods. In-place expansion can create excessive code size and increase compile time but it can produce more efficient code in terms of execution time. Call-based approaches can lose at performance due to call instruction overhead but win at compilation time and code size. We survey intrinsics implementation in several modern virtual machine compilers: HotSpot Java Virtual Machine, and Android RunTime. We implement our hybrid approach in the LLVM-based compiler of Ark VM. Ark VM is an experimental bytecode virtual machine with garbage collection and dynamic and static compilation. We evaluate our approach against in-place expansion and call approaches using a large set of benchmarks. Results show the hybrid approach provides considerable performance improvements. For string-related benchmarks, the hybrid approach is 6.8% faster compared to the no-inlining baseline. Pure in-place expansion achieves only 0.7% execution time improvement of the hybrid implementation. We explore two versions of our hybrid approach. The "untouched" version lets LLVM control inlining decisions. The "heuristic" approach was developed after we observed LLVM's tendency to inline code too aggressively. This research helps compiler developers balance execution speed with reasonable code size and compile time when implementing intrinsics.

Keywords: intrinsics; LLVM compiler infrastructure; virtual machines.

For citation: Zavedeev D.V., Zhuykov R.A., Skvortsov L.V., Pantilimonov M.V. Fast Calls and In-Place Expansion: A Hybrid Strategy for VM Intrinsics. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 121-134. DOI: 10.15514/ISPRAS-2025-37(6)-7.

Быстрые вызовы и раскрытие на месте: гибридная стратегия для встраиваемых функций виртуальных машин

Д.В. Заведеев, ORCID: 0009-0009-1477-5249 <zdenis@ispras.ru>
Р.А. Жуйков, 0000-0002-0906-8146 <zhroma@ispras.ru>
Л.В. Скворцов, 0000-0002-1580-1244 <lvs@ispras.ru>
М.В. Пантилимонов, 0000-0003-2277-7155 <pantlimon@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Данное исследование предлагает гибридный подход к реализации встраиваемых компиляторных функций, направленных на улучшение производительности. Встраиваемые компиляторные функции – особые функции, которые предоставляют доступ к низкоуровневым возможностям или улучшают производительность. Текущие реализации, как правило, используют либо раскрытие на месте, либо подходы, основанные на вызове. Раскрытие на месте может избыточно увеличить размер кода и время компиляции, но создать более эффективный код по времени исполнения. Подходы, основанные на вызове, могут проигрывать по производительности в связи с вызовом функции, но выигрывают по размеру кода и времени компиляции. Мы рассматриваем реализации встраиваемых функций в нескольких современных компиляторах виртуальных машин: в виртуальной машине Java HotSpot и в Android RunTime. Мы реализуем гибридный подход в LLVM-компиляторе для виртуальной машины Ark. Ark – это экспериментальная байткодная виртуальная машина со сборкой мусора, динамическим и статическим компиляторами. Мы сравниваем наш гибридный подход с раскрытием на месте и подходом, основанном на вызовах, на большом наборе бенчмарков. Результаты показывают, что гибридный подход по времени исполнения показывает значительное улучшение. Строковые бенчмарки выполняются быстрее на 6.8% по сравнению с подходом, основанном исключительно на вызовах, в то же время, чистое раскрытие на месте быстрее на 0.7% гибридного подхода. Мы рассматриваем две версии гибридного подхода. "Untouched" версия позволяет LLVM самому принимать решение о встраивании функции или выборе вызова. Подход "heuristic" мы разработали после того, как заметили, что LLVM в "untouched" подходе производит излишне агрессивное встраивание функций. Данная статья поможет разработчикам компиляторов найти баланс между временем исполнения, размером кода и временем компиляции при реализации встраиваемых функций.

Ключевые слова: встраиваемые функции; компиляторная инфраструктура LLVM; виртуальные машины.

Для цитирования: Заведеев Д.В., Жуйков Р.А., Скворцов Л.В., Пантилимонов М.В. Быстрые вызовы и раскрытие на месте: гибридная стратегия для встраиваемых функций виртуальных машин. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 121–134 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–7.

1. Introduction

Intrinsics are functions that compilers handle in a special way. They serve two primary purposes: first, providing programmers access to low-level functionality within high-level languages like C or C++ (such as the `__rdtsc` function, which returns cycle count for a CPU) and second, enhancing application performance (such as optimized implementations of `Math.log` in HotSpot JVM [1]). This paper focuses specifically on performance-oriented intrinsics.

Compilers typically implement intrinsics using two main strategies: in-place expansion, where the intrinsic call is expanded into a sequence of machine instructions directly at the call site; and the call approach, which generates a call to precompiled or runtime-generated code. Our survey shows that modern compilers utilize both approaches, though implementation practices vary across different compiler systems.

Virtual machine (VM) compilers often expand intrinsics without code size limitations, which can lead to excessive memory usage and potential performance degradation. Conversely, some VM compilers never use in-place expansion, instead relying exclusively on efficient calls. While this approach preserves memory and reduces code size, it might introduce performance penalties. On the other side, Damásio et al. propose using an inlining to reduce code size [2].

In this paper, we propose a hybrid approach that combines both in-place expansion and efficient calls. This method aims to deliver performance benefits while maintaining reasonable code size and compile time.

We implement the hybrid approach and evaluate its performance within the LLVM-based [3] compiler of Ark Virtual Machine (Ark VM) [4]. Ark VM is an experimental bytecode virtual machine with garbage collection, dynamic (Just in Time – *JiT*) and static (Ahead of Time – *AoT*) compilers. It executes bytecode compiled from ETS, a statically typed language similar to TypeScript (but other frontends are possible to the VM's bytecode).

The hybrid approach has experimental status, and shows promising results: for string-related benchmarks execution time is 0.932 (6.8%) times of the no-inlining baseline, while the always *in-place* expansion is only 0.993 (0.7%) times of the *hybrid* approach performance.

We have been working on LLVM-based compilation for Ark VM, thus we can experiment with intrinsics optimizations with relative ease in the LLVM-based compiler.

In this paper, we begin by introducing Ark VM in the 2nd section, where its essential components are described. Next, we survey various ways to encode intrinsics in the HotSpot Java Virtual Machine [5] and Android RunTime [6], while also including Ark VM for comparison in the 3rd section. The 4th section then presents a hybrid approach that combines both *call* and *in-place* expansion techniques.

Following this, the 5th section compares the *in-place*, *calls*, and *hybrid* approaches to intrinsics encoding in Ark VM, focusing on code size, application performance, and compile time. To provide empirical evidence, we run a comprehensive set of benchmarks, demonstrating how performance varies across a wide range of scenarios. Additionally, a dedicated micro-benchmark for the `StringEquals` intrinsic is developed, allowing us to analyze in detail the advantages and disadvantages of each approach.

We also discuss two flavors of the hybrid approach: *untouched* and *heuristic* because our setup reveals that LLVM tends to be overly aggressive with inlining, as will be shown in the 5.4 section. The paper concludes by summarizing the results.

2. Ark VM

2.1 Execution in Ark VM

Ark Virtual Machine is a bytecode virtual machine.

Compilers like Clang or GCC transform programs in a high-level language like C into a sequence of machine instructions specific to the CPU. In this approach a program compiled for x86_64 CPU cannot be run on aarch64 CPU directly.

We illustrate it in Fig. 1: compiler takes a program in C as an input, and produces an executable file, which in turn runs on x86_64 CPU.

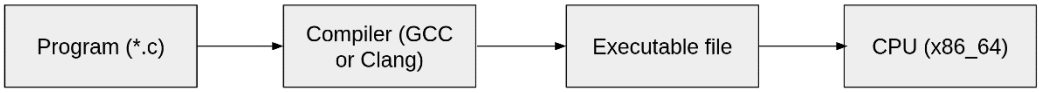

Fig. 1. Compilation and execution for compilers producing directly executable files.

For VMs like Ark it is different: a compiler for high-level language transforms a program into a sequence of *bytecode* instructions. *Bytecode* is a set of instructions for *artificial* processor that does not exist but is a convenient abstraction. The VM emulates such a processor. Thus, a bytecode program runs anywhere where the VM is available, despite environment.

Fig. 2 is specific to Ark VM: ETS is a programming language, ETS frontend is a program that transforms ETS to bytecode, it takes a program in ETS language, and produces bytecode file (`.abc`), which then runs on Ark VM.

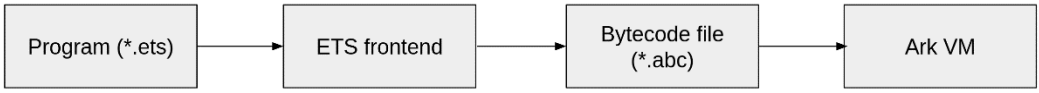

Fig. 2. Compilation from high-level language to bytecode file and execution in Ark VM.

Fig. 1 and Fig. 2 look similar but there is a huge difference: in the first case, the executable file runs directly on CPU, in the second case Ark VM acts as an emulator for CPU (Ark VM is a program itself). To emulate a CPU VM has an *interpreter* – a component which reads instructions from bytecode file one by one and emulates their behavior. Such an interpreter in its simplest form is implemented in the language of VM, e.g., C++, using a dispatch table. Modern VMs employ different optimizations techniques to make interpretation faster, e.g., HotSpot JVM has a template interpreter generated at runtime [7].

Despite improvements made to the interpreter, interpretation remains slow. To alleviate this, VMs employ compilers. VM monitors itself during the execution, and for frequently executed functions *Just in Time* compiler produces native code for the CPU where the VM runs at *runtime*. VMs also employ *Ahead of Time* compilers which produce native code *before* the execution. *AoT* compiler is used in different scenarios when upfront compilation is more profitable: e.g., to compile parts of the standard library, or parts of an application. Ark VM has both *JiT* and *AoT* compilers.

2.2 Ahead of Time compilation in Ark VM

AoT compiler in Ark VM transforms bytecode file into a file with instructions executable by CPU directly.

In Fig. 3 we show the inputs for Ark VM: Ark VM requires a file with bytecode for execution, and optionally AoT compiled files – `*.an` with machine code. Ark VM upon start loads the AoT compiled files, and registers the machine code internally: for each AoT compiled function, the VM changes its entry point so that instead of interpretation, the AoT compiled machine code is executed.

If there is no registered machine code for a function, then VM starts with interpretation of the function. If the function is executed often enough, *JiT* compiler produces native code dynamically.

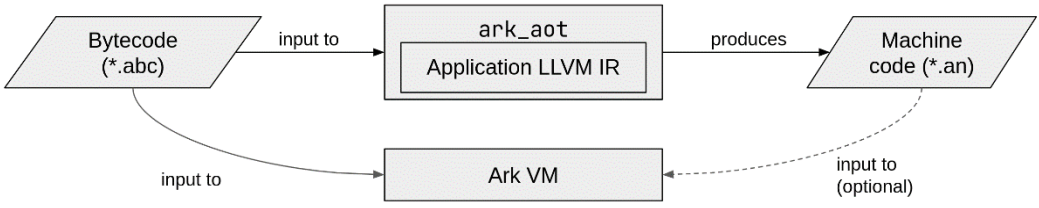

Fig. 3. Inputs into Ark VM: bytecode file (mandatory), AoT-file (optional).

In Ark VM there are two kinds of *AoT* compilers: stock, and LLVM-based. Stock compiler is written from scratch for the VM, and LLVM-based compiler transforms bytecode into LLVM IR, then LLVM produces native code for that LLVM IR. In this paper we focus only on LLVM-based AoT compiler.

2.3 Intrinsics in Ark VM

In Ark VM there are two kinds of intrinsics:

1. Regular intrinsics which are encoded using various strategies. E.g., to get a length of a String compiler uses special Intermediate Representation (IR – form of code between the source language and machine code). Also, there are intrinsics which are replaced by calls to the standard library function calls like `sin`.
2. FastPaths – precompiled intrinsics or VM routines.

2.4 FastPaths in Ark VM

To call a FastPath compiler produces a *fast-call* in Ark VM.

At the low-level, to call a function the compiler must obey a set of rules which define how to pass arguments to a function, how to return a value from a function, which registers must be preserved across calls (callee-saved registers), and are not guaranteed to be preserved (caller-saved registers or call-clobbered registers). This set of rules is *calling convention* in compiler terminology.

Fast-call means using a special calling convention which preserves almost all registers across calls. It uses `ArkFast` calling convention in our patched LLVM 15 version.

The purpose of the calling convention is to make the code for caller as cheap as possible, so the caller does not need to preserve registers before making a call. The `ArkFast` calling convention is supported for aarch64 and x86_64. The calling convention has also caller saved registers: 4 for aarch64, and 6 for x86_64. Parameters, and return value registers are caller or callee-saved depending on their presence. For comparison, the default calling convention for arm64 – aapcs64 defines more than 20 of the registers as call-clobbered [8].

2.5 Intrinsic FastPaths example: StringEquals

Consider the following code in ETS language in Listing 1.

```
function bar(a: string, b: string): boolean {
    return a === b;
}
```

Listing 1. String comparison in ETS language.

The code in the Listing 1 uses the `'==='` operator to check if two strings are equal. Two strings are equal if their lengths are equal and characters composing these strings are equal. In Ark VM the performance of such a comparison is so critical that string comparison is an intrinsic function – `StringEquals`.

`StringEquals` intrinsic in AoT code is encoded as a fast-call to precompiled machine code.

2.6 Categories of FastPaths

Ark VM has 90 FastPaths. There are FastPaths for high-level operations like comparing Strings which directly map onto standard library methods. Also, there are low-level FastPaths, e.g., allocation routines. We categorize all FastPaths in Table 1.

As we can see, most of the FastPaths are related to strings in Ark VM.

2.7 Summary

1. VM emulates artificial CPU to make programs for the VM cross-platform. Instructions for such cpu are *bytecode* instructions. Ark VM is one of such machines.

2. When executing the bytecode Ark VM starts with interpretation, then *JiT* compiler can produce machine code if the function is frequently executed, *AoT* compiler can produce machine code before execution.
3. There are two kinds of intrinsics in Ark VM: regular and FastPath. Regular intrinsics are encoded using various strategies. FastPath is precompiled machine code for VM routines or intrinsics. Most of the FastPaths are string related in Ark VM.

Table 1. Categories of FastPaths in Ark VM.

Category	Count	Comment
Allocation	5	Allocation routines for objects and arrays
Garbage Collection (GC)	9	GC pre-, post-write barriers
String	61	String routines: creating a substring, appending to string, etc.
Array copy	11	Various routines to support ArrayCopy operation in ETS
Misc	4	Interface lookup cache, monitor lock and unlock, check cast

3. Industry approaches to intrinsics

In this section we survey approaches to intrinsics encoding. We do it for HotSpot JVM (openjdk version "21.0.1" 2023-10-17), Android RunTime (specifically dex2oat aml_art_351110180, Nov 2024), and Ark VM.

We observe the following strategies of intrinsics encoding in production VMs:

1. *Call replacement*. VM compiler replaces a call to intrinsic function with a call to another equivalent function but more performant. Android RunTime, Ark, HotSpot JVM perform call replacement for sin function: Android RunTime [9] and Ark replace it with a call to the `std::sin` function, HotSpot JVM replaces it with a call to the runtime generated `_dsin` function [10].
2. *Special IR*. In this case, compilers replace a call with a special sequence of IR instructions. Example of such strategy is encoding of `java.lang.String::length` in Android RunTime [11]. Ark also replaces calls to get string length with several compiler IR instructions, these instructions become a part of the function's body, and are optimized together.
3. *In-place expansion*. In this case, compiler replaces a call to an intrinsic function with a sequence of machine instructions at the call site. We observed that behavior in HotSpot JVM for `java.lang.Arrays::equals(byte[], byte[])` method [12]. Android RunTime also employs *in-place* expansion for `java.lang.String::equals` method [13].
4. *Fast-call*. A special case for call replacement. Ark VM employs this approach for FastPath intrinsics. As we described in the 2.4 section, in this case a call to an intrinsic is encoded as a call with calling convention that preserves almost all registers.
5. *Calling other intrinsics*. OpenJDK 21 employs this approach to improve the performance of `String::equals` method: in OpenJDK 21, the method is not intrinsic itself, but it calls `StringLatin1::equals` intrinsic. Júnior Löff et al. [14] propose leveraging that method, and suggest replacing several intrinsics written in assembly with implementation in Java using Java Vector API.

Each of the approaches has its advantages and disadvantages. *Call replacement*, *in-place expansion*, and *fast calls* are used to implement complex high-level operations, like comparing strings. *Special*

IR is usually used for simple operations like getting a string length. *Fast-call* is similar to *call replacement* since those approaches are basically calls.

We are interested in the *in-place expansion* and *call* approaches because using solely either of approaches has its own advantages and disadvantages. We list them in Table 2.

We see that HotSpot JVM, Android RunTime, and Ark VM lack flexibility in encoding intrinsics: for some of the intrinsics their approach is *black-or-white*, with no option to adapt or vary the encoding method.

We propose an approach that allows the encoding strategy to change based on potential benefits.

Our hybrid approach aims to be the approach taking the best of the *in-place* and the *call* approaches: reasonable code size, improved performance, and manageable compile time.

Table 2. In-place and call approaches compared.

Criteria	**In-place**	**Call**
Code size	Usually grows indefinitely	No code size growth
Performance	Can produce more performant code due to (1) absence of a call instruction, (2) possible optimizations after inlining	Can lose at performance
Compile time	Takes more time to compile	Less time to compile

4. Hybrid approach to intrinsics encoding in Ark VM

4.1 VM before introducing hybrid approach

As of current state, Ark VM produces precompiled machine code for FastPaths – VM routines and high-level operations like comparing strings. We illustrate it in Fig. 4: FastPath compiler produces a `fastpath.o` file with machine code from the source code, then the `fastpath.o` file becomes part of the runtime library.

In Fig. 5 we show how AoT compiler transforms bytecode file into a file with machine code. When compiling calls to FastPaths, compiler produces calls to functions from the dynamic library (`libarkruntime.so`) – a part of VM runtime. AoT compiler has no definitions of the FastPath functions, and thus cannot inline the FastPath into the application code.

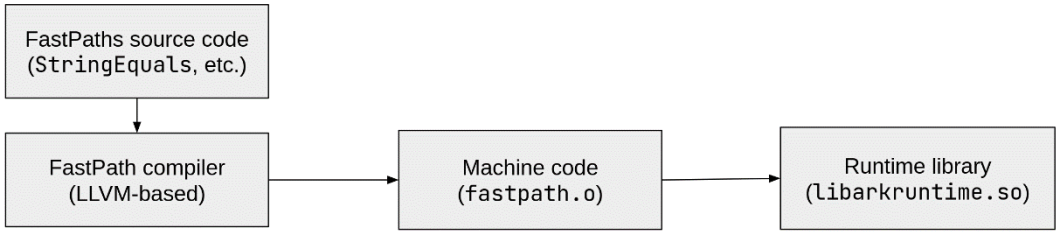

Fig. 4. FastPath compilation before hybrid approach.

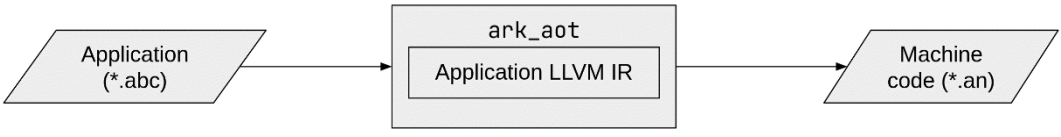

Fig. 5. AoT compilation before hybrid approach.

4.2 VM with hybrid intrinsics encoding

We involve joint work of two compilation phases. First, during VM build, LLVM compiles FastPaths and dumps LLVM bitcode. We show the difference in Fig. 6: now we produce not only a binary file with machine code (`.o`) but also a bitcode file with definitions of the FastPaths (marked blue in the diagram).

Second, during AoT compilation we supply the dumped bitcode file for joint optimizations purposes. Fig.7 now shows that *AoT* compiler has LLVM IR from bytecode, and for FastPath.

Now, LLVM *AoT* compiler has the definitions of the FastPath functions, not only their declarations, and can inline the FastPath functions into *AoT*-code.

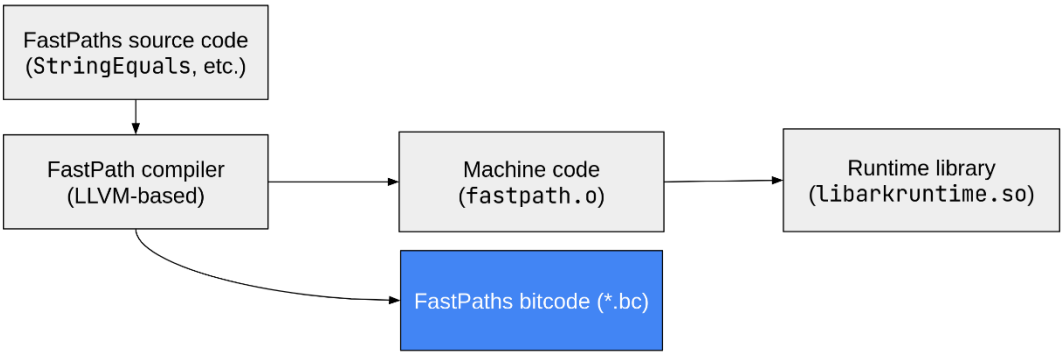

Fig. 6. FastPath compilation with hybrid approach.

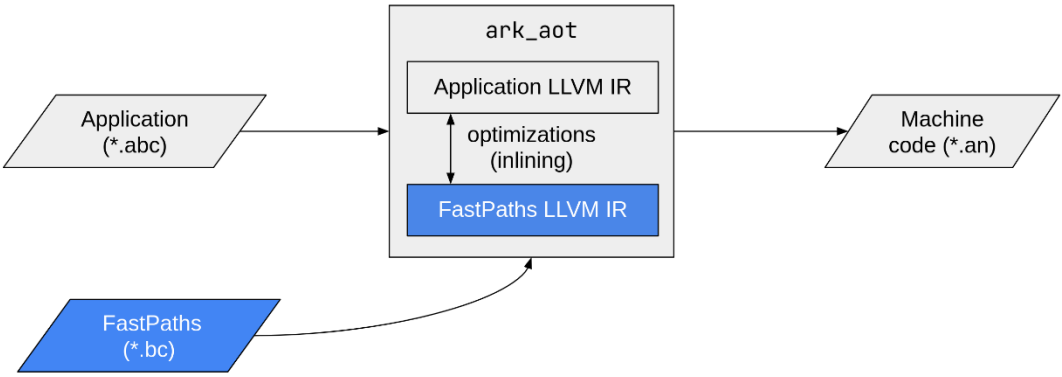

Fig. 7. AoT compilation with hybrid approach.

New parts are blue in these diagrams. All grey components in the diagrams are already implemented. VM uses the produced `.an` file as usual when started: it loads the `.an` file upon start, registers compiled machine code, and executes more optimal compiled code.

4.3 Controlling inlining

To allow fine-tuning of inlining we introduce an LLVM pass which forces or forbids inlining of FastPaths. `fastpath-max-always-inlines` option controls the number of forcefully inlined FastPaths into AoT compiled function:

- N > 0 – forces inlining of the first N FastPaths met in the function, forbids inlining of other FastPaths in function
- 0 – forbids inlining of all FastPaths in a function – equivalent of default mode
- -1 – neither force, nor forbid inlining of FastPaths. In this case, the decision is up to LLVM.

A FastPath can call another FastPath in Ark VM, in this case, we inline the FastPath unconditionally.

5. Evaluation

In this section we evaluate configurations of FastPath inlining listed in Table 3.

We introduce the "heuristic" scenario in our measurements because we know that LLVM is too aggressive at inlining in advance, as it will be evidenced in the 5.4 section.

Table 3. Inlining configurations.

Approach	Description	fastpath-max-always-inlines
No inlining	Inlining of all FastPaths into AoT code is forbidden	0
Always inline	Inlining of all FastPaths into AoT code is enforced	2 000 000 000
Heuristic	Only the first occurrence of FastPath in the function is inlined forcefully, inlining is forbidden for other FastPaths in the function	1
Untouched	Leave the decision up to LLVM	-1

We are interested in three metrics:

1. Execution time – how long it takes to perform an operation.
2. Code size – the size of a produced binary AoT file. In Ark these are files with `.an` extension.
3. Compilation time – how long it takes to compile a file.

To measure code size, and compilation time we compile the standard library file of ETS language. To measure performance, we run a large set of benchmarks. We also gather compile time and code size for benchmark files.

But first, we describe which FastPaths we inline and do not, and describe our setup.

In our evaluations we do not inline:

1. GC-related FastPaths because they are called by a pointer, thus it is impossible to know which FastPath is called.
2. "Interposer" FastPaths. They simply wrap a single function call, and never called by AoT code. There are 2 interposer FastPaths.

That is, we inline 78 FastPaths out of 90.

Another important note: to provide a fair comparison between approaches, we disabled safepoints in our experimental code. Safepoints function like scheduled checkpoints where running code briefly pauses to allow the virtual machine to perform maintenance tasks such as garbage collection. When FastPath code is compiled on its own, the compiler does not insert safepoints because they are not needed in this specialized code. However, when we inline this FastPath code into the main program, the compiler automatically inserts safepoint instructions into loops, treating it like regular code.

This automatic insertion creates an inconsistency: the same FastPath code performs differently when inlined versus when kept separate, not because of our hybrid approach but because of how the compiler handles it. By disabling safepoints throughout all code paths, we establish a consistent environment that allows us to measure the true performance differences between our approaches without this compiler introduced variable affecting our results.

While safepoints are important in production environments, disabling them for our comparative analysis ensures our research findings reflect the actual differences between implementation approaches rather than artifacts of the compilation process.

5.1 Setup

We run all measurements:

- On aarch64 Linux.

- Using Release built version of Ark VM and Release LLVM.
- Using vm-benchmarks framework [15].

5.2 Standard library: code size, and compile time

We compile `etsstdlib` – the bytecode of ETS standard library, resulting file with machine code is `etsstdlib.an`. The file has 10764 calls to FastPaths. Table 4 summarizes our measurements. In parentheses, we show score against the "No-inlining baseline".

Table 4. Different inlining strategies for standard library.

Approach	Size, bytes	Compile time, s
Always inline	7 489 992 (x1.39)	303 (x1.62)
No inlining	5 384 648 (x1)	187 (x1)
Heuristic	5 642 696 (x1.05)	200 (x1.07)
Untouched	6 101 448 (x1.13)	220 (x1.18)

As we could expect, the "Always inline" approach grows the size of binary file the most (x1.39 of original), compile time is also at the maximum. The "Untouched" approach grows the code size by x1.13 of the original, compile time is also lower than in "Always inline". Our heuristic approach provides minimal code size growth of the binary file, and minimal compile time growth.

5.3 Benchmarks: performance, compile time, and code size

We run a large set of benchmarks. As we described in the section 2.6, most of the FastPaths are string related, so we should keep an eye on benchmarks related to strings. We evaluate our approach on 2305 benchmarks. Those include 276 String benchmarks.

We show relative value which should be read as "x times of no inline baseline".

The relative values are geometric mean ratios across metrics: execution time, `.an`-file size, and compile time. For all cases "lower is better".

In our measurements, the VM uses AoT-compiled standard library. That is, if a benchmark calls a function from the standard library, then the called function is also *AoT*-compiled with `fastpath-max-always-inlines` set to match the approach under measurement.

5.3.1 String benchmarks

For string benchmarks the results are in Table 5. As we can see, the always inline approach has the best execution time: 0.925 of the no-inlining baseline, the worst compile time, and code size – 2.381, 1.313 of the original respectively.

The untouched approach is 0.957 of the original at execution time. Code size is almost the same as the original – 1.015. Compile time is less than for "always inline" approach.

The heuristic approach is closer in performance to the always inline approach, and does not increase compile and code size dramatically.

Table 5. String benchmarks results

Characteristic\Approach	Always inline	Heuristic	Untouched
Execution time	0.925	0.932	0.957
Code size	1.313	1.059	1.015
Compile time	2.381	1.302	1.250

5.3.2 All benchmarks

For the whole set of benchmarks, the results are in Table 6. In the whole set of benchmarks overall performance gains drop compared to the string benchmarks.

This happens because of the nature of FastPath we are inlining: most of them are String related.

Table 6. All benchmarks results.

Characteristic\Approach	Always inline	Heuristic	Untouched
Execution time	0.980	0.999	0.998
Code size	1.220	1.044	1.037
Compile time	1.879	1.159	1.192

5.4 StringEquals benchmark

In this section we take benchmarks to extreme: usually a benchmark has a reasonable number of intrinsic calls, we are going to vary a number of calls from 50 to 300 and see results.

Mostly, this section motivates using the heuristic approach, because we see the evidence that leaving inlining decision entirely up to LLVM can increase code size significantly, and degrade performance. The *heuristic* approach solves this problem: it forces inlining of the first FastPath call met in a function, and disables inlining of all other FastPaths in the function. We have not tried sophisticated heuristics since in our opinion this research must be done against the production ready implementation of FastPath inlining, and as of now safepoint support is missing. Despite that our guess to inline the first FastPath call is quite good in all terms, which we are interested in.

We run benchmark, where we call `String::equals` ranging number of times between 50 and 300 for 4 and 1024 length strings. Strings have equal lengths, and differ only in the last character.

Fig. 8 visualizes the results for 4 length strings. For 1024 string length graph with performance is in Fig. 9.

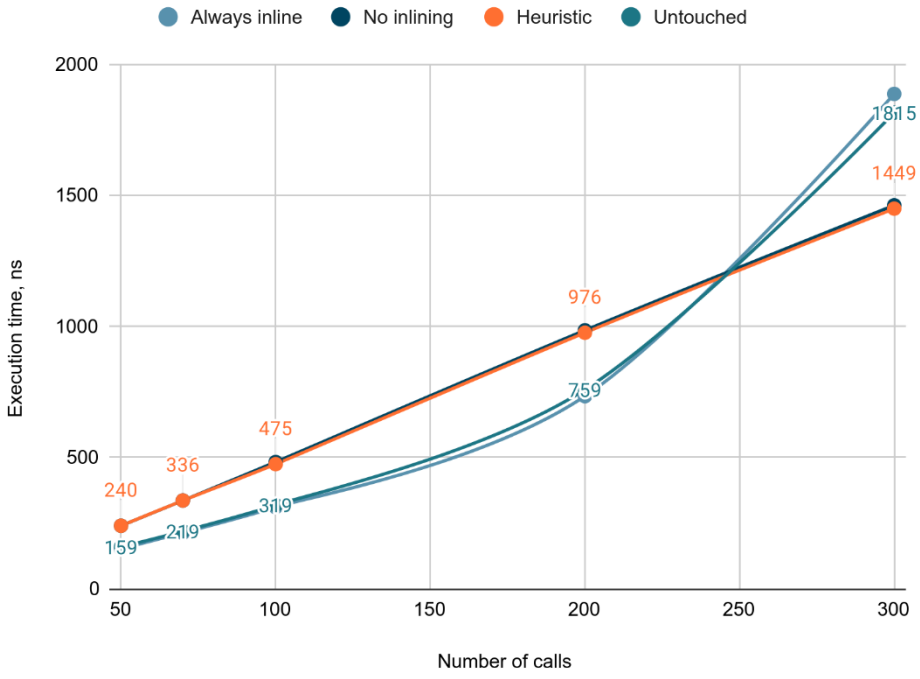

Fig. 8. Performance for 4 characters string.

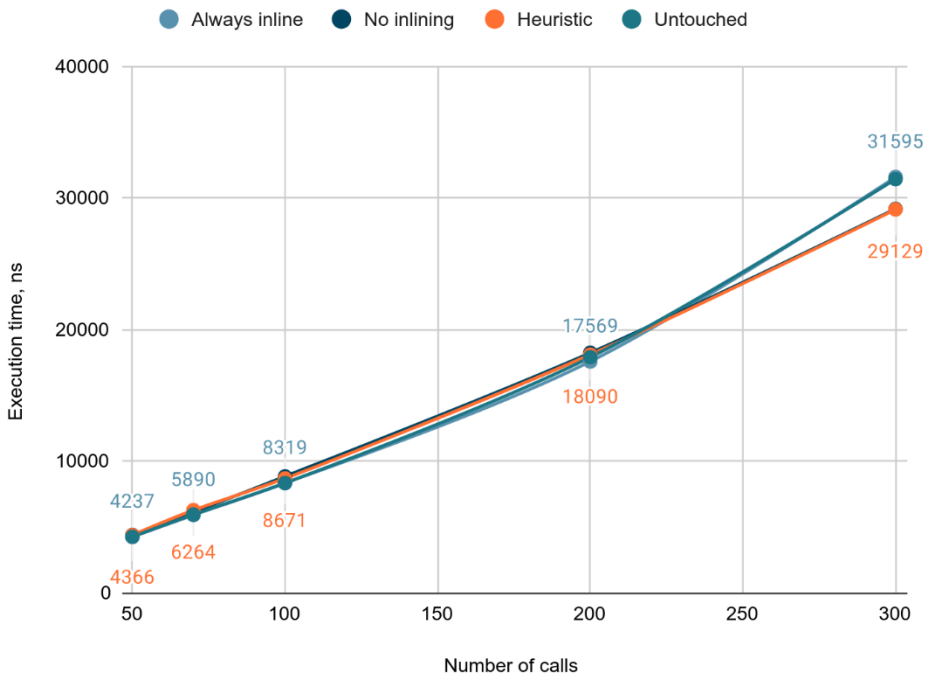

Fig. 9. Performance for 1024 characters string.

There are interesting points to note:

- *Untouched* and *always inline* approaches show similar performance
- *Untouched* and *always inline* approaches have lower performance for 300 number of calls compared to the *heuristic* and to the *no-inlining* approaches
- For 1024 length strings there is no significant difference between approaches since the strings are long, comparing 1024 characters outweighs eliminated call overhead.

The reason why *untouched* approach shows performance close to *always inline* is that when calls are left *untouched*, LLVM inlines almost all the `StringEquals` calls.

For 300 number of `StringEquals` calls the *no-inlining* and *heuristic* cases are more performant, independently of the string length. The reason we *think* of is that the machine code in *always inline* and *untouched* cases does not fit the instruction cache. A CPU we use has 64KiB of instruction cache per core, code size is 106.71KiB in *always inline* case, which is larger than the CPU cache.

As we see, leaving the decision about inlining up to LLVM can degrade performance and increase code size.

Because of increased code size, and possible performance degradation we have introduced the *hybrid* approach in favor of the *untouched* approach.

6. Future work

We see our future work as follows:

1. Safepoint support. We had to disable safepoint placement to make comparison fair but to make the *hybrid* approach ready for production, we should allow safepoint placement without sacrificing the performance of FastPaths.

2. More intrinsics. We based our work only on a part of intrinsics that are FastPaths. Other intrinsics can be migrated to FastPath so that we could inline them too.
3. x86_64 support. For now, hybrid approach supports only aarch64 architecture, though LLVM AoT compiler supports x86_64 and aarch64.
4. Heuristic improvement. Our heuristic to always inline the first FastPath met in the function already shows decent performance, compile time, and code size but still there can be more effective heuristics.

7. Conclusion

We proposed a hybrid approach to intrinsics encoding for VMs' compilers. We implemented the approach in LLVM-based static compiler for Ark VM. Our approach combines two approaches: *fast-calls* and *in-place expansion*.

We compared the hybrid approach to pure *fast-calls* and *in-place* expansion. We present a hybrid approach in two flavors: *untouched* where LLVM makes an inlining decision based on its own metrics, and *heuristic* which forces inlining of the first intrinsic met in a function and forbids inlining of other intrinsics.

The *heuristic* flavor shows better performance, code size management, and compile time compared to the *untouched* version. For string-related benchmarks, the hybrid heuristic approach is 6.8% faster compared to the no-inlining baseline while the pure in-place expansion achieves only 0.7% execution time improvement of the hybrid heuristic implementation.

Intermediate language for intrinsics like LLVM IR gives more control over intrinsics encoding: we can inline them, leave them as calls, or leave the decision up to compiler.

References

[1]. Dehghani A. HotSpot Intrinsics. https://alidg.me/blog/2020/12/10/hotspot-intrinsics (accessed May 14, 2025).
[2]. Damásio T., Pacheco V., Goes F., Pereira F., and Rocha R. Inlining for Code Size Reduction. In Proceedings of the 25th Brazilian Symposium on Programming Languages (SBLP '21), Association for Computing Machinery, New York, NY, USA, 2021. doi: 10.1145/3475061.3475081
[3]. The LLVM Project. https://llvm.org/ (accessed May 20, 2025).
[4]. ArkCompiler Runtime Core: Static Core. https://gitee.com/openharmony/arkcompiler_runtime_core/tree/master/static_core (accessed May 14, 2025).
[5]. Oracle. Java Virtual Machine Technology Overview. https://docs.oracle.com/en/java/javase/21/vm/java-virtual-machine-technology-overview.html (accessed May 20, 2025).
[6]. Android Open Source Project. Android runtime and Dalvik. https://source.android.com/docs/core/runtime (accessed May 20, 2025).
[7]. HotSpot Runtime Overview: Interpreter. https://openjdk.org/groups/hotspot/docs/RuntimeOverview.html#Interpreter|outline (accessed May 14, 2025).
[8]. ARM Software. Procedure Call Standard for the Arm® 64-bit Architecture (AArch64): The Base Procedure Call Standard. https://github.com/ARM-software/abi-aa/blob/main/aapcs64/aapcs64.rst#the-base-procedure-call-standard (accessed May 14, 2025).
[9]. Android RunTime (ART) ARM64 sin Entrypoint Initialization. https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/arch/arm64/entrypoints_init_arm64.cc;l=187;drc=0e8091312485670e84ee17daf25256e5836112b0 (accessed May 14, 2025).
[10]. HotSpot Stub Generator for x86_64 sin Function. https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/cpu/x86/stubGenerator_x86_64_sin.cpp (accessed May 14, 2025).
[11]. Android RunTime (ART) Compiler. HArrayLength instruction https://cs.android.com/android/platform/superproject/main/+/main:art/compiler/optimizing/nodes.h;l=6463;drc=621d1350d431ed0cc3d4a5a43a079adc1d86a31f (accessed May 14, 2025).

[12]. HotSpot C2 MacroAssembler for x86: Arrays Equals. https://github.com/openjdk/jdk/blob/60a4594b9f9acd82ef3ff22fc6a2df238dd981b9/src/hotspot/cpu/x86/c2_MacroAssembler_x86.cpp#L4377 (accessed May 14, 2025).

[13]. Android RunTime (ART) ARM64 Intrinsics. StringEquals encoding. https://cs.android.com/android/platform/superproject/main/+/main:art/compiler/optimizing/intrinsics_arm64.cc;l=2288;drc=9c2a0a8bfc5369a110956ac26cf9cf145a6a4bb7 (accessed May 14, 2025).

[14]. Löff J., Schiavio F., Rosà A., Basso M., and Binder W. Vectorized Intrinsics Can Be Replaced with Pure Java Code without Impairing Steady-State Performance. In Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), Association for Computing Machinery, New York, NY, USA, 2024, pp. 14–24. doi: 10.1145/3629526.3645051

[15]. ArkCompiler Runtime Core: VM Benchmarks. https://gitee.com/openharmony/arkcompiler_runtime_core/tree/OpenHarmony_feature_20241108/static_core/tests/vm-benchmarks (accessed May 14, 2025).

Information about authors

Денис Владиславович ЗАВЕДЕЕВ – аспирант Института системного программирования им. В.П. Иванникова Российской академии наук. Сфера научных интересов: компиляторы, языковые виртуальные машины.

Denis Vladislavovich ZAVEDEEV – postgraduate student at Ivannikov Institute for System Programming of the Russian Academy of Sciences. Research interests: compilers, language virtual machines.

Роман Александрович ЖУЙКОВ – старший научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статическая и динамическая оптимизация программ, компиляторные технологии.

Roman Aleksandrovich ZHUYKOV – Senior researcher in Compiler Technology department at ISP RAS. Research interests: static and dynamic program optimization, compiler technologies.

Леонид Владленович СКВОРЦОВ – стажёр-исследователь отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Leonid Vladlenovich SKVORTSOV – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

Михаил Вячеславович ПАНТИЛИМОНОВ – научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ, компиляторные технологии, СУБД.

Mikhail Vyacheslavovich PANTILIMONOV – researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, DBMS.