Software: Practice and Experience

WILEY

RESEARCH ARTICLE OPEN ACCESS

Register-Based and Stack-Based Virtual Machines: Which Perform Better in JIT Compilation Scenarios?

Bohuslav Šimek | Dalibor Fiala | Martin Dostal

University of West Bohemia, Plzeň, Czech Republic

Correspondence: Dalibor Fiala (dalfia@kiv.zcu.cz)

Received: 10 December 2024 | **Revised:** 2 July 2025 | **Accepted:** 10 August 2025

Funding: The authors received no specific funding for this work.

Keywords: just-in-time compilation | register-based VMs | stack-based VMs | virtual machines

ABSTRACT

Background: Just-In-Time (JIT) compilation plays a critical role in optimizing the performance of modern virtual machines (VMs). While the architecture of VMs – register-based or stack-based – has long been a subject of debate, empirical analysis focusing on JIT compilation performance is relatively sparse.

Objective: In this study, we aim to answer the question: "Register-based and stack-based virtual machines: which perform better in JIT compilation scenarios?"

Methods: We explore this through a comprehensive set of benchmarks measuring execution speed. To achieve this, we developed identical test cases in languages that support both types of VM architectures and ran these tests under controlled conditions. The performance metrics were captured and analyzed for JIT compilation, including initial interpretation, bytecode translation, and optimized code execution.

Results: Our findings suggest that register-based VMs generally outperform stack-based VMs in terms of execution speed. Moreover, the performance gap between the two architectures in mixed execution mode, which essentially copies characteristics of the underlying virtual machine, suggests that making the right choice of VM architecture is still important.

Conclusion: This study provides developers, researchers, and system architects with actionable insights into the performance trade-offs associated with each VM architecture in JIT-compiled environments. The findings can guide the design decisions in the development of new virtual machines and JIT compilation strategies.

1 | Introduction

Virtual Machines (VMs) have been integral to computing for decades, providing an abstraction layer that allows programs to operate in a manner that's independent of the underlying hardware. Historically, there have been two predominant architectures for these VMs: register-based [1] and stack-based [2]. Register-based VMs utilize a set of registers to hold data during operations. Instructions in such VMs typically specify the registers they operate on. This architecture, prevalent in physical hardware CPUs, naturally extended to virtual machines, mirroring the operational dynamics of physical computing environments. Programming languages such as *Lua* [3] and *PHP* [4] utilize register-based VMs.

Stack-based VMs, on the other hand, rely on a stack to hold data. Operations are performed at the top of the stack, and instructions push or pop data from it. The *Java Virtual Machine* (JVM), for

instance, is one of the most renowned instances of a stack-based VM [5].

Since the inception of this dichotomy, an important question has arisen: Which approach is more efficient [6–8]? It has been observed that register-based machines spend less time in overall execution, but this comes at the cost of less compact code size. On the other hand, stack machines might require more VM instructions for specific tasks, each often associated with an unpredictable indirect branch during VM instruction dispatch [6, 8].

Several researchers have conducted in-depth studies comparing architectures to support these claims, providing examples where one may outperform the other [6, 8]. However, except for a few speculations presented in *Virtual Machine Showdown: Stack Versus Registers* [8] and a severely flawed indirect comparison in *Evaluation of Android Dalvik Virtual Machine* [9] a role of Just-In-Time (JIT) compilation is mainly absent from these studies.

JIT compilation involves the compilation of computer code during a program's execution rather than before its execution. *JIT* start to be more prominent in the late 1990s, especially in efforts to optimise Java performance [10]. In modern programming language implementations, *JIT* compilation is widespread, as demonstrated by its adoption in languages such as *C Sharp*, *Java*, or *JavaScript*. The approach to *JIT* compilation is varied, but they typically focus on identifying sections of a program that are executed frequently [11].

Focusing on compiling only these specific sections makes the compilation process faster and more suitable for runtime execution. Furthermore, the real-time data obtained from actual execution enables *JIT* compilers to apply advanced optimization techniques, including inlining and type specialization or even deoptimization of some sections. As a result, the code produced can be as efficient as, or in some instances even outperform, the code produced by traditional static compilers [12].

To determine whether the inclusion of *JIT* significantly impacts the choice between stack-based and register-based virtual machines, we propose a benchmark. This benchmark will be grounded on the implementation of two virtual machines: one based on register architecture and the other on stack architecture. For both machines, a shared *JIT* engine with switchable base optimization will be implemented.

1.1 | Contributions and Organization

This article has the following contributions:

- We confirm a speed difference between register and stack-based virtual machines as it has been stated in studies *A Performance Survey on Stack-based and Register-based Virtual Machines* [6] and *Virtual Machine Showdown: Stack Versus Registers* [8].
- We design a framework that allows us to compare virtual machine architectures together with *JIT* compilation.
- We design benchmarks and confirm speculation [8] that compilation time will play only a marginal role and that the most important thing is still the cost of interpretation.

The rest of this article is organized as follows: Section 2 discusses related work. Section 3 provides an overview of stack and register virtual machines to help the reader better understand their architectural implications in *JIT* compilation. Section 4 presents possible approaches to *JIT* compilation and discusses their impact on virtual machines. Section 5 introduces our approach to the problem. Section 6 evaluates the basic performance of both implementations. Finally, we conclude the article in Section 7.

2 | Related Work

The effectiveness of both approaches has been a hot topic since the inception of virtual machine types. Initially, this topic was closely associated with the physical architecture of computers. One of the first studies in this area, *The Case Against Stack-Oriented Instruction Sets* [1], challenges the idea that stack-based code is more compact and requires decoding fewer instructions. The author presents a few small examples to support their claim, but these are not comprehensive enough to draw general conclusions. A similar critique applies to the direct response titled *Reply to The Case Against Stack-Oriented Instruction Sets* [13]. This article correctly addresses the issue of trivial examples in the original article but fails to provide any complex examples or statistics to support its arguments. The debate continues in the article *The Evaluation of Expression in a Storage-to-Storage Architecture* [14], where the author tries to support their position with data, but again, only using limited examples involving a few operations.

The abandonment of hardware stack machines in the 1980s and the subsequent increase in machine performance marked a shift in the debate toward efficient software implementation. During this time, languages like *BCPL* and *Pascal* emerged, primarily implemented within virtual machine [15]. Virtual machines that executed both of these languages were stack-based. Efforts to increase efficiency included initiatives to accelerate the dispatch loop [16], using super instructions, and implementing compile-time optimization [10]. In the 1990s, *Java* emerged. As previously mentioned, *Java* utilizes a stack-based virtual machine.

In contrast, the Inferno virtual machine represents a register-based approach, with its design choices elaborated in *The Design of the Inferno Virtual Machine* [17]. This article details the architecture of the *DIS virtual machine*, offering insights into the rationale behind its specific design elements. The *DIS virtual machine* is a register-based machine that was part of the Inferno operating system. They conclude that generating native code from register-based bytecode would be easier. However, this claim is not substantiated by any data or other studies.

Comprehensive studies comparing the two types of virtual stacks appeared after 2000, primarily in the papers *A Performance Survey on Stack-based and Register-based Virtual Machines* [6] and *Virtual Machine Showdown: Stack Versus Registers* [8], which will be discussed in more detail later. It is worth mentioning at

this point that neither of these studies delves into the role of *JIT* compilation in detail. The first mentioned [6] completely ignores *JIT* compilation, while the second [8] briefly touches upon it and offers some speculation, which we will explore in more detail later.

Outside the academic environment, a decision was made to use a new virtual machine, *Parrot VM*, for *Perl 6*, opting for a register-based architecture. This decision was not supported by any data, a situation that was practically mirrored in the case of a new virtual machine for *Lua* [3]. Another notable decision in this field was implementing the *Dalvik virtual machine* for the *Android* platform. This virtual machine was developed as an alternative to the Java virtual machine, with the key difference being its register-based design [9]. Consequently, the compilation process was extended by an additional step—conversion to the register-based variant. The performance of this machine in direct comparison with Java was evaluated in the paper *Evaluation of Android Dalvik Virtual Machine* [9]. The authors also discuss the influence of the *JIT* engine, albeit with some acknowledged limitations that may have biassed the results of the comparison [9].

One of the latest contributions to the debate is *WebAssembly*. Unlike a virtual machine or a programming language, *WebAssembly* is a standardized, portable binary-code format for executable programs [18]. In this case, a stack-based format was chosen. The choice of this format is not backed by any statistical data but is justified primarily by the claim that it facilitates efficient binary coding and verification [19].

2.1 | A Performance Survey on Stack-Based and Register-Based Virtual Machines

In the article [6] the authors built two versions of virtual machines: *Inertia*, a register-based virtual machine, and *Conceptum*, a stack-based virtual machine. During their benchmarking, they found that *Inertia* is considerably more efficient in terms of execution time and instruction dispatch compared to *Conceptum*. Specifically, *Inertia* spends 20.39% less overall execution time and 66.42% less time in instruction dispatch than *Conceptum*.

Further analysis of the data supports the hypothesis about the performance differences between register-based and stack-based virtual machines. Register-based machines, like *Inertia*, typically execute far fewer dispatches than stack-based machines. However, stack-based machines occasionally outperform their register-based counterparts in benchmarks involving heavy arithmetic operations, owing to their fewer fetches per dispatch. In such scenarios, *Conceptum's* stack-based design allows for faster execution of arithmetic operations, as it requires fewer operands. Despite this, *Inertia* excels in recursion and memory operations. Unfortunately, the authors did not take into account any form of *JIT* compilation.

2.2 | Virtual Machine Showdown: Stack Versus Registers

In the article [8] the authors compare the performance of stack and register virtual machine implementations. Building upon previous work described in *The Case for Virtual Register Machines* [20], the authors quantified the number of instructions for both architectures using a basic translation scheme. The current article introduces a more sophisticated translation and optimization method for converting stack VM code to register VM code, aiming to provide a more precise assessment of the potential of virtual register machine architectures.

Experimental results are presented for a fully-featured register *Java Virtual Machine* (JVM) generated with the help of the *wmgen* [21] interpreter generator, ensuring that both versions of the interpreter reuse the same base codes. The benchmark also considers multiple types of dispatch methods, namely *inline-threaded* [22], *direct-threaded*, *token-threaded*, and *switch dispatches*. The authors mention that as the cost of dispatches decreases, any benefit from using a register VM instead of a stack VM diminishes.

Key findings include that a register architecture requires, on average, 46% fewer executed VM instructions than a stack architecture, though the register code is 26% larger. This increase in code size results in only a minor additional CPU load per eliminated VM instruction. Performance tests on an *x86-64* machine show that the register machine averages 1.48 times faster using a *C switch* statement for dispatch and 1.15 times faster than a stack JVM, even with more efficient inline-threaded dispatch.

At the end of the article, the authors offer some speculations, such as that translation from stack-based will be easier because stack variants are closer to the traditional *Intermediate Representation* (IR) used in compilers. However, this does not correspond to the current common *IR* in *LLVM* and *GCC* compilers, both of which use a register-based IR [23, 24]. The second speculation is more interesting from our perspective–it suggests that only a tiny part of the compilation time will be involved. The last speculation is that it will play a much more significant role in the interpretation's cost.

2.3 | Evaluation of Android Dalvik Virtual Machine

The primary motivation behind this article [9] is to compare the Android *Dalvik Virtual Machine* (DVM) with *Java Micro Edition* (JVME), as represented by the phoneMe reference implementation, on an experimental tablet board using an embedded Java benchmark. In stark difference to the previous articles, a *JIT* compilation is taken into account. The comparison itself has been made on the experimental tablet board and under separate Android and Linux installations.

The authors repeatedly acknowledge that their selected approach can be problematic; for example, they use different versions of the Linux kernel and the implementation of the *C standard library* (libc). This decision can lead to varying performance at the level of the operating system itself or system libraries and to distortion of the results. Another problematic area is the implementation of the virtual machine's core libraries, which varied in their approach across both virtual machines, even for the same functions. Some of them use native methods, some use the *Java Native*

Interface, and others are programmed directly in *Java*. This can also contribute to potential distortion [25].

Even the comparison of the *JIT* engine itself cannot be considered direct as *JVME* generates *A32* instruction set. In contrast, the *DVM* generates *T32* instruction set—a different variant of *ARM* assembly. The authors attempt to take this into account by considering the fixed penalty of 6% against *T32* instruction set. The last difference is the type of *JIT* compiler: trace in the case of the *DVM* and method in the case of *JVME*. The authors also acknowledge this.

The authors use two *DVM* implementations in assembly and *C* during the comparison. They find that the assembly version of the *DVM* interpreter is 60% faster than the *JVM* interpreter, yet the *C* version is only 6% faster. This can cast doubt on the findings of [8] because this difference is significantly lower. They also examine the size of the generated code, and it is interesting to find that the whole comparison is distorted by the existence of *super instructions* in the case of *DVM*. *Super instructions* represent a set of small operations in one single operation, which can lead to increased performance. If we exclude methods containing these super instructions, the *DVM* bytecode size becomes 33% larger than that of the *JVM*.

The performance measurement with *JIT* enabled shows a clear superiority of the *JVM*, as it exhibits an 11.7-fold speedup. In comparison, the *DVM* shows only a 4.0-fold speedup, achieving only one-third of the *JVM's* performance. According to the authors, this is a significant difference, even when considering the 6% performance difference between the *A32* and the *T32* code. The authors proceed to discuss the reasons why the difference is so significant. One reason is the small trace size, which limits efficient code generation, resulting in lower-quality code. In fact, they found that a trace includes only three bytecode instructions, on average.

Although [9] yields interesting results, it is, unfortunately, impossible to overlook several problems in the methodology that can lead to potential biases and invalidate the study.

3 | Stack and Register Virtual Machines

Stack-based architectures generally make implementation easier by automatically managing the locations of operands. All calculations are performed on the stack, requiring that every value be placed into the stack before any mathematical operations are conducted [2]. If we take into account a simple mathematical expression such as: $a = b \times c$, typical pseudo-bytecode for stack-based virtual machines, based on *JVM* bytecode might look like:

```
iload 2
iload 3
imul
istore 1
```

In this example, we load two variables `2` and `3`, representing values a and b, into the stack and do the multiplication operation. The result of this operation will be available in the stack, so we need another extra instruction to put this value into variable `1`. Both of the operands used for multiplication have been destroyed during the operation, and thus, we cannot reuse them.

On the other hand, register-based architectures expect explicit operand addressing. An illustrative pseudo-bytecode, similar to the *Lua* programing language bytecode format, is:

```
imul r1, r2, r3
```

In this case, just one instruction is needed as this instruction will directly fetch the data from the registers `r2` and `r3` and store them into target register `r1`. This operation does not destroy values in registers `r2` or `r3`.

The main difference between both of them is the number of instructions that need to be used to perform the multiplication operation. We can observe a need for more instruction in the case of stack-based machines. However, instructions themselves tend to be simpler—as they usually need just one operand. We can also easily calculate the *liveness* of both input operands. If we focus more on the execution cost, it can be divided into three distinct parts:

- Instruction Dispatch
- Operand Access
- Computation

The first component, Instruction Dispatch, involves retrieving the upcoming VM instruction from memory and then directing the flow to the appropriate interpreter code segment responsible for executing that VM instruction. As we have seen, many tasks can be described with fewer register machine instructions than stack-based machines.

Consequently, virtual register machines offer significant potential to reduce the number of instruction dispatches. In the *C* language, dispatch is primarily implemented using a comprehensive switch statement, with each case addressing a unique opcode within the VM instruction set. While this dispatch mechanism is straightforward, it can be inefficient, especially when branch prediction is suboptimal. For this reason, an alternative such as *threaded dispatch* has been proposed and used [16]. However, branch prediction has been significantly improved in the last three generations of CPU, and it should not pose any extra cost [26].

As the cost associated with dispatches decreases, the benefits of using a register VM over a stack VM become less evident. However, switch-based and basic *threaded* dispatch remain the dominant interpreter strategies. The switch remains the sole viable option when adherence to *ANSI C* is crucial.

In the context of operand access during VM instruction execution, it is important to note that a substantial part of the processing cost is attributed to operand retrieval. Unlike stack code, where the operand's location depends on the stack pointer's position, register code clearly specifies the operand location. Consequently, register instructions are often lengthier than their stack counterparts, leading to a more cumbersome register code and requiring increased memory accesses throughout execution [6].

Computation is the final component influencing VM performance. Regardless of the representation, basic computation is essential for operating on a virtual machine and cannot be entirely eliminated. However, it often constitutes the smallest portion of the overall cost [6, 8].

4 | *JIT* and It's Compilation Methods

The primary motivation for *JIT* stems from the fact that programming languages often possess features such as late binding, dynamic loading, and various forms of polymorphism that can hinder the generation of high-quality code at compile time. *Ahead-of-Time* (AOT) compilers traditionally base their decisions on the program's source code, producing optimized code efficiently for languages with strictly defined data types and control flow. However, features like dynamic typing, certain polymorphisms, and an open class structure can prevent compilers from having vital information until runtime [11].

JIT represents a form of runtime optimization, and a crucial aspect of these optimizations is the trade-off they present between compile time and code quality. The use of a *JIT* compiler hinges on finding a balance between pre-compilation efforts, *JIT* processing, and the optimization achieved through *JIT* compilation. Ideally, a runtime compiler should optimize more cycles than it consumes.

When considering a runtime system, deciding the frequency and focus of *JIT* invocation becomes important for performance. An implementation must choose between:

- converting Virtual Machine code to native code before execution (AOT) or
- invoking the *JIT* only for frequently executed segments.

Runtime optimizers operate at varying granularities. This design choice significantly influences overall effectiveness, dictating the optimizations the *JIT* can apply and the code fragment sizes it optimizes. As has been already mentioned, we can divide *JIT* approaches into three groups: *trace-based*, *method-based*, and combined approach [11].

4.1 | Trace-Based

A trace optimizer monitors runtime branches and jumps to identify frequently executed sequences, known as *hot traces*. When these traces are executed beyond a predefined threshold, the *JIT* compiler is activated to produce an optimized native code representation. This optimization not only encompasses local adjustments but also broader, regional enhancements, which can be viewed as interprocedural in the context of an *AOT* compiler. Such optimization is essential as a runtime trace can often include calls and returns [27, 28].

4.2 | Method-Based

In the case of method *JIT*, a method optimizer identifies procedures consuming substantial portions of the overall running time by examining specific counters. When a method is suitable for further optimization, the *JIT* compiler is invoked to generate its optimized native code. This comprehensive approach allows for optimizations, including code motion, regional instruction scheduling, dead-code removal, and strength reduction. Additionally, some optimizers facilitate inline substitution, enabling them to incorporate frequently executed code segments directly into the hot method or, in cases where most calls to a hot method arise from a singular call site, to inline the callee into the caller.

In essence, the method *JIT* compiler operates analogously to a regular compiler system. It processes the Virtual Machine (VM) *Intermediate Representation* (IR), translating it into its own variant of *IR*. After this transformation, the *JIT* applies multiple optimization passes on the *IR*. Finally, it produces native code, executing tasks such as instruction selection, scheduling, and register allocation.

4.3 | Combined Approach

It seems natural to combine these approaches—we will call the result approach combined. However, integrating these two approaches is not straightforward due to challenges like the need for multiple compilers, handling interactions between fragments compiled with entirely different methods, and choosing the right strategy. A combination of these approaches can be found in the *HipHop Virtual Machine* [4] in the form of region-based compilation or experimental meta-tracing *JIT* compiler framework *BacCaml* [11].

4.4 | Selecting a Proper Optimization Target

Regardless of the selected approach, a pivotal role of the system is to ascertain which methods necessitate compilation. Crafting an efficient strategy is essential, especially when it concerns gathering profile information to assist in these decisions. All previous approaches benefit from doing *JIT* for chunks of code that occupy a significant portion of execution time. Such methods can be identified by evaluating the frequency of their calls or the number of iterations in their loops. Data is collected through the instrumented Virtual Machine (VM) Code or the VM-code engine to better understand a method's efficiency. In the case of the instrumented VM Code, profile counters are both incremented and evaluated.

5 | Our Approach

To ensure a proper comparison without any potential interferences or differences in the implementation of both virtual machines, we will build both virtual machines from the ground up. The same applies to the *JIT* engine.

This section outlines the design and architectural decisions for the *Nozomi virtual machine* (Nozomi) and the *JIT* engine *Zero-kei*. Both will be used to benchmark the speed differences between stack and register architectures.

5.1 | Nozomi Virtual Machine Architecture

Nozomi operates as a virtual machine in two distinct modes: stack and register. Designed explicitly for benchmarking, its dual

modes and shared code eliminate potential discrepancies. The switch between these modes occurs during compilation, ensuring that neither mode incurs performance costs on behalf of the other. *Nozomi* is built in *Rust* edition 2021, and in both modes, the machine employs the Rust *match* construct, analogous to the *switch* statement in C. This choice stems from the fact that while *switch* dispatch might not be the fastest method, it remains widely adopted. Given its consistent use in both modes, it should not distort the results.

Both modes in *Nozomi* support fundamental mathematical operations such as addition, subtraction, multiplication, and division. Other supported functionalities include function calls and recursion. In terms of data types, the virtual machine supports primitive data types such as *integer* and *float*.

```
.function static int32 factorial(int32) {
    .code {
        RECV r1
        IS\_SMALLER r1, 2, r2
        JMPNC r2, label:RECURSION
        RET 1
        RECURSION: SUB r3, r1, 1
        SEND\_VAR r3
        FCALL "factorial", r4
        MUL r5, r1, r4
        RET r5
    }
}

.function static int32 main(int32) {
    .code {
        ASSIGN 5, r1
        SEND\_VAR r1
        FCALL "factorial", r2
        ECHO r2
        RET r2
    }
}
```

LISTING 1 | Factorial implementation in register version

Irrespective of its operational mode, the virtual machine interprets an assembler program defined in a text file. This file is parsed into bytecode, after which the virtual machine searches for a bytecode function labeled *main()*. If found, this bytecode function is executed, returning an execution result structure containing the value from the bytecode *main()* function. In an interpreter setup, the bytecode is perpetually interpreted in a continuous loop using the *match* construct in Rust. For this process, a variable symbolizing the instruction pointer is deployed. Depending on this variable's value, an instruction is chosen, and through a match, an appropriate handler for its execution is selected.

The method of passing parameters during execution can significantly impact the speed of the virtual machine when calling functions. When passing parameters during a function call, the register version of the virtual machine uses a stack. Parameters are placed on a dedicated stack with the opcode *SEND_VAR*. If the virtual machine encounters the opcode *FCALL*, which represents the calling of the function, this stack is passed into the function. Later, inside the function, the values of these parameters are obtained using the *RECV* opcode. In contrast, function calling in a stack-based machine revolves around removing a fixed number of parameters from the stack. Inside the called function, parameters are automatically placed into individual variable slots; they are not immediately put onto the stack.

```
.function static int32 factorial(int32) {
    .code {
        LOAD 0
        PUSH 2
        IS\_SMALLER
        JMPNC label:RECURSION
        PUSH 1
        RET
        RECURSION: LOAD 0
        PUSH 1
        SUB
        FCALL "factorial"
        LOAD 0
        MUL
        RET
    }
}

.function static int32 main(int32) {
    .code {
        PUSH 5
        FCALL "factorial"
        STORE 1
        LOAD 1
        ECHO
        LOAD 1
        RET
    }
}
```

LISTING 2 | Factorial implementation in stack version

5.2 | Zero-kei *JIT* Engine

The *Zero-kei JIT* engine is a method-based *JIT* engine, and, by design, it resembles a full-fledged compiler to a degree. It has its own register-based intermediate representation known as the *Low-Level Intermediate Representation* (LLIR). This representation is more low-level than a typical virtual machine, particularly when compared to *Nozomi*, but it still operates with useful abstractions, such as virtual registers. This allows for better abstraction of the target *Instruction Set Architecture* (ISA). Furthermore, *LLIR* is designed with optimization facilitation in mind. Currently, *Zero-kei* supports only the *x86-64 ISA*. However, this limitation is not a concern for our study, as the comparison will be made using the same *ISA*.

The *JIT* engine is integrated into both versions of the virtual machine. Virtual machines will count the number of executions of called functions, and upon reaching a specific threshold, the function will be compiled by the *JIT* engine and executed.

Additionally, we note that *Zero-kei JIT* compilation does not occur in a background thread. Compilation is performed synchronously at the point when a function reaches the invocation threshold. Consequently, during the execution phase, there is no ongoing background *JIT* activity that could interfere with benchmark timing or contribute to additional system noise.

Like regular compiler backends, method *JIT* compilers also have the same architectural components, including instruction selection, scheduling, and register allocation [29]. All of these elements interplay and significantly influence code quality. For this reason, we will now briefly discuss the implementation of these parts in *Zero-kei*.

5.2.1 | Instruction Selection

Instruction selection in compilers translates an *intermediate representation* (IR) into the target processor's *ISA* using techniques like peephole optimization or tree-pattern matching. The instruction selector, operating at compile time, transforms the *IR* into target machine code, yet challenges arise from the diverse ways *ISA*s can implement *IR* constructs. *Zero-kei* employs peephole optimization as its *IR* takes the form of a flat assembler, which stands to benefit from this approach. We believe that a set of common *peephole* optimizations will positively affect *Zero-kei*'s speed, especially in the stack-based machine mode.

Peephole optimization involves a *sliding window* or *peephole* that moves over the code. At each stage, the optimizer inspects the operations within this window, searching for recognizable patterns that can be enhanced. Upon identifying a pattern, the code is rewritten to feature a more efficient sequence of instructions. The efficiency of these optimizers is attributed to their restricted pattern sets and confined viewing window [30].

A typical example of such a pattern is a store operation followed immediately by a load operation from the same location. Such inefficiencies often arise from sequential translations or other optimization techniques like dead-code elimination or constant folding. A *peephole* instruction selector divides the selection process into three main tasks: expansion, simplification, and matching [31]. The expander converts the *IR* into a sequence of *Low-Level IR* (LLIR) operations that encapsulate all the instruction actions. *Zero-kei* features two expanders: one expecting stack-based code and another for register code. The functionality of the stack-based code expander is crucial, which we will discuss in detail in a separate part.

After the expander, the matcher assesses the refined *LLIR* against a library of patterns, pinpointing the pattern that most accurately represents all the effects within select *LLIR* instruction. All generated code relies on virtual registers rather than physical ones and must undergo register allocation.

5.2.2 | Stack to Register Expander

During the process of stack-to-register expansion, we will leverage the fact that the height and contents of the VM operand stack are always known. This allows for a straightforward mapping from stack locations to register numbers, as every value on the operand stack essentially functions as a temporary, short-lived variable. In contrast, local variables are long-lived and remain active for the duration of the method execution.

TABLE 1 | Straightforward expansion.

Stack-based	Register-based
load 2	mov r2, r5
load 3	mov r3, r6
mul	mul r7, r5, r6
store 1	mov r7, r1

TABLE 2 | Bytecode expansion in Zero-kei.

Stack-based	Register-based
load 2	(ignored)
load 3	(ignored)
mul	mul r4, r2, r3
store 1	mov r4, r1

Because of this, many stack-based instructions can be converted into equivalent register-based virtual machine instructions, turning implicit operands into explicit operand registers. However, there are certain exceptions associated with operations on the stack itself:

- Instructions that load a local variable to the operand stack or store data from it become `move` instructions or `assign` in the case of *Zero-kei IR*.
- Instructions that pop the operand stack, namely `pop` and `pop2`, are completely ignored.
- Instructions like `dup` and `dup2` that handle the stack turns into particular "move" sequences based on the operand stack's current state.

Table 1 provides an illustrative example of the bytecode translation process. Consider a simple program that wants to multiply two values, which can be described by the equation $a = b \times c$. Typically, the first operand within an instruction serves as the destination register. In this case, the bytecode's specific task is to multiply two integers from two separate local variables and then save the result into a different local variable.

As shown in Table 1, the instruction expansion, by design, will produce many extra instructions. These can be viewed as leftovers from the expansion process. Such unnecessary instructions can adversely affect execution performance, especially when the most efficient version of the code would simply be: `mul r1, r2, r3`.

This straightforward approach has been slightly modified in the case of *Zero-kei* stack to *IR* expansion, incorporating an automatic collapse of load instructions during the expansion process if the next operation actively uses both of these values. Thanks to this and additional peephole optimization, these superfluous instructions will be effectively eliminated (Table 2).

5.2.3 | Impact of Expander

As discussed in the previous section, stack-based bytecode translation requires an explicit expansion step that converts operand-stack operations into temporary virtual registers. This introduces small, but measurable translation-time overhead, approximately 1.58%. Conversely, register-based bytecode inherently provides explicit operands, lowering translation overhead, typically below 0.73%, during *LLIR* construction. The absolute difference of 0.85% indicates that the expander's cost is negligible in practice, as this is a one-time operation.

We explicitly distinguish between the costs incurred during *IR* construction, translation overhead, and those occurring during optimization. Although both bytecode variants undergo identical local optimization phases once in *LLIR* form, the initial stack-to-register expansion adds instructions that increase the *IR* construction overhead. While downstream optimizations often remove redundant instructions, patterns specific to stack operations, such as duplicate, may produce *IR* sequences less conducive to immediate simplification, resulting in slight performance inefficiencies that persist after optimization.

5.2.4 | LLIR Design Considerations and SSA Form

Zero-kei internally uses *LLIR* as its intermediate representation. Notably, *LLIR* is not in *static single assignment* (*SSA*) form and our compilation pipeline does not include an *SSA* conversion at any stage. The absence of an *SSA-based* representation in *LLIR* limits the applicability and effectiveness of optimizations such as aggressive dead code elimination and advanced value numbering. Consequently, compared to an *SSA-based IR*, our *LLIR* potentially sacrifices some optimization effectiveness.

Considering these trade-offs, designing bytecode closer to *SSA* form could potentially reduce translation overhead and improve optimization, allowing some optimization and transformation passes to be combined [32]. However, *SSA* form is inherently less suitable for direct interpretation, as it can slow down interpreters due to the runtime overhead of variable versioning, the complexity of handling ϕ-functions, the interpreter's preference for simpler control flow, and the increased instruction count typically associated with *SSA*. A practical evaluation [33] supports this claim. This finding is particularly significant, given that JIT-compiled code often runs alongside interpreted code in mixed-execution environments. *SSA-based* bytecode may add complexity or inefficiencies during interpretation, which can hurt overall performance. While currently beyond the scope of our study, exploring a balanced approach that uses *SSA-based* bytecode primarily for *JIT* compilation, while maintaining efficient interpretation, appears to be a promising direction for future work.

More advanced instruction selection, such as *SSA-based* lowering or global pattern matching, could reduce stack-to-register overhead. Early value numbering and better register coalescing could also minimize redundant moves. These optimizations would likely narrow the performance gap between stack-based and register-based virtual machines under *JIT* and *AOT* modes. However, the core advantage of register-based architectures, due to lower dispatch frequency, explicit operand handling, and alignment with modern CPUs, would remain. Full validation of this hypothesis with an SSA-enabled *JIT* pipeline and stronger backend optimization is an important goal for future work.

5.2.5 | Instruction Scheduling

Instruction scheduling is a process that reorders operations in a procedure to optimize execution time, assuming the code is already optimized. This scheduling is crucial for processors with pipelined execution, as the sequence of operations can directly influence performance [34]. *Zero-kei* does not utilize any form of instruction scheduling. However, this does not impact the benchmarks, as instruction scheduling is skipped for both virtual machine execution modes.

5.2.6 | Register Allocation

Effective utilization of processor resources, particularly registers, is vital for the performance of compiled code, as registers provide quicker access than memory. The challenges associated with register allocation and assignment, in their most expansive form, are *NP-complete* [35]. Due to this complexity, strategies are required to minimize unnecessary data transfers between registers and memory. Among the various strategies for data spilling, *graph-colouring* [35] and *linear scan* [36] are commonly used. The latter is especially efficient for *JIT* compilers [37]. As previously highlighted, the speed of *JIT* compilation is dependent on the efficiency of its components; hence, *Zero-kei* employs the *linear scan* algorithm.

6 | Evaluation

In this section, we evaluate the basic performance of the *Nozomi* virtual machine in conjunction with the *Zero-kei JIT* engine. First, we introduce our setup and describe how we gathered data from microbenchmark programs. Next, we present the evaluation results of both the VM and *JIT* across various settings using microbenchmark programs.

6.1 | Setup

6.1.1 | Implementation

We implemented a set of benchmark programs directly in both versions of *IR* code. Each program has two identical versions, one stack-based variant and the second register-based. We selected the following set of problems: factorial (recursive), factorial (iterative), fibonacci (recursive), fibonacci (iterative), pi-approximation, collatz conjecture, Euler's totient function, factorize, integer square root and perfect number.

Every problem is chosen for benchmarking due to its computational characteristics, which can help in evaluating the performance differences between *register-based* and *stack-based* virtual machines, along with JIT engine capabilities. The *factorial (recursive)* involves calculating the product of all positive integers up to a given number using a recursive method, testing the VM's

ability to handle recursive function calls and stack management efficiently. In contrast, the *factorial (iterative)* calculates the same factorial value but utilizes an iterative approach, highlighting the VM's efficiency in loop and arithmetic operation management.

Similarly, the *fibonacci (recursive)* computes a fibonacci number using recursion, which is crucial for assessing the VM's performance in managing recursive calls and the associated overhead. The *fibonacci (iterative)* emphasizes loop efficiency and arithmetic operations, offering a point of comparison to its recursive counterpart.

Pi-approximation involves a series of arithmetic operations, testing the VM's floating-point computation efficiency and repetitive arithmetic tasks. The *Collatz conjecture*, with its conditional branching and arithmetic operations, examines the VM's ability to efficiently manage dynamic and unpredictable execution paths. Calculating *euler's totient function* requires iterating over a range of integers and evaluating the greatest common divisor, assessing the VM's capability in handling mathematical functions and iteration.

Factorize implements the factorization of integers into their prime factors. It evaluates the VM's efficiency in executing algorithms that involve division operations and conditional checks. Problem *integer square root* focuses on arithmetic operations, especially how the VM optimizes calculations involving square roots and integer arithmetic. Finally, in the case of a *perfect number*—determining whether a number is perfect involves summing its divisors and comparing the sums, which evaluates the VM's performance in arithmetic operations, loops, and conditionals.

All of these problems have their own driver program, which executes the problem in a loop with a fixed count of 1000 executions efficiently, allowing it to kick in *JIT* compilation after reaching a particular compilation threshold.

```
.function static int32 main() {
    .code {
        ASSIGN 1, r1
        BENCH:SEND\_VAR 15
        FCALL "fibonacci", r2
        INC r1
        NEQ r1, 1000, r2
        JMPC r2, label:BENCH
        RET r1
    }
}
```

LISTING 3 | Benchmarking loop for fibonacci function

While each problem in this set targets specific computational traits relevant to virtual machine performance, they collectively serve to isolate the costs of interpretation, operand access, and *JIT* overhead. This isolation is essential for clearly attributing performance differences to specific implementation choices and avoiding extraneous interference. Nonetheless, these microbenchmarks do not necessarily mirror the complexity found in large-scale or real-world applications. For instance, production workloads often have complex control flow, dynamic data structures, extensive library interactions, concurrent execution contexts, asynchronous event handling, and external system calls. Such complexities introduce additional layers of performance variability that these more straightforward benchmarks do not adequately capture.

Thus, microbenchmark results should be seen as indicative, not definitive, of real-world performance. Larger benchmarks or full applications introduce factors like inlining, garbage collection, threading, memory patterns, and caching. These workloads tend to have more varied instruction mixes and longer runtimes, which can amplify the benefits of advanced *JIT* optimizations and diminish the overhead of instruction dispatch.

6.1.2 | Methodology

The executed benchmark is based on the Rust benchmarking tool Divan [38], as designing our own benchmarking tools would be out of scope for this study. One of the fundamental challenges of benchmarking is that the operation could potentially be too fast for the timer to measure accurately [39]. This is further complicated by the fact that timer precision varies depending on the platform and operating system and can differ across individual setups. To account for this, the timer used for measuring will calculate its precision by conducting multiple timing measurements.

Timing measurements will be conducted by utilizing the following algorithm: The Precision algorithm seeks the smallest non-zero duration that the timer can reliably measure. It updates this minimum duration based on the results of successive timing attempts. The process involves either rapid successive measurements or measurements with increasing delays. A count is maintained to determine when a consistent minimum duration is observed, which is then considered the timer's precision.

In improving the precision of time measurement, the benchmarking tool bundles a series of iterations into a single entity, denoted as a *sample*. The equation $v(s)$ is utilized to compute the required number of iterations, or *sample size*, to effectively address the issues related to the accuracy of the timer:

$$v(s) = \begin{cases} v(2 \times s) & \text{if } t(s) < 100 \times \tau_{\text{precision}} \\ s & \text{if } t(s) \geq 100 \times \tau_{\text{precision}} \end{cases} \quad (1)$$

The benchmarking tool determines the final sample size by progressively doubling the number of iterations until the duration of a sample is at least 100 times the τ precision. This adjustment is achieved by re-timing each iteration at $t(2 \times s)$, ensuring that the final outcome is not just based on the initial duration measurement [38].

If one disregards the re-timing aspect of $t(s)$ and assumes that $t(s)$ consistently yields a predictable value, then the calculation of $v(s)$ can be understood as follows:

$$v(s) \approx 2^{\frac{100 \times \tau_{\text{precision}}}{t(s)}} \quad (2)$$

The inspiration behind the approach of scaling sample size based on timer precision from study *Robust Benchmarking in Noisy Environments* [40]. However, the approach of the benchmarking tool is more straightforward as we are not taking into account timer accuracy [38].

The benchmarking tool does not depend on timer accuracy since determining such accuracy without a more precise reference timer is challenging. This is particularly true given that the *Instant*, a primary time-measuring component in Rust, is implemented using the most accurate timer available on the target platform [41]. As our benchmark was executed on Linux, this boils down to using *clock_gettime* (*Monotonic Clock*). Therefore, the existing approach can be considered sufficiently effective for our current needs.

The benchmarking process also takes into account overhead from the benchmarking loop. This is done by measuring the additional time consumed by the benchmarking loop itself, separate from the timer's precision. It involves repeatedly timing a set number of operations and calculating the smallest average duration per operation. This average represents the loop's overhead, which is crucial for ensuring the accuracy of performance measurements, as it accounts for the extra time introduced by the benchmarking process.

6.1.3 | Execution

To ensure the most accurate interpretation of the results, we began by measuring both virtual machines without enabling the *JIT* engine, evaluating both the register and stack-based variants. We conducted a sequence of tests, initially conducting 1000 *warm-up* executions. This was followed by a more extensive phase where we executed 1000 samples, with each sample comprising 50 executions, cumulatively amounting to 50,000 iterations.

In the second measurement, we accounted for the performance with the *AOT* compilation done by the *JIT* engine. In this mode, the code is first compiled and then executed. Again, we carried out 1000 samples, with each sample comprising 50 executions, cumulatively amounting to 50,000 iterations. For the third measurement, we enabled the *JIT* engine itself. Once more, we completed 1000 runs, dismissing the first 100 as *warm-up* trials.

We selected a relatively low *JIT* compilation threshold of 50 for purely experimental reasons, ensuring that the *JIT* engine starts promptly even with a moderate number of executions. In contrast, production systems often use higher thresholds to amortize the overhead of compiling hot code paths. For example, in the *Oracle JVM* with *tiered compilation* enabled, the default threshold for compilation at tier three is 2000 executions [42]. When tiered compilation is disabled, the threshold varies: 1000 executions in client mode and 10,000 in server mode [43]. While our chosen threshold may emphasize *JIT* startup costs more than typical in real-world applications, it enables rapid experimentation and reliably surfaces differences between *register-based* and *stack-based* virtual machines.

Similarly, we discarded the first 100 runs of each benchmark as a *warm-up* phase. This practice allows the system, including caches, CPU frequency, and the *JIT* engine itself, to reach a steady state before measurements begin. This approach is supported by prior work [39] and is commonly adopted in benchmarking frameworks, though the specific warm-up length may vary.

We would like to explicitly clarify that after the warm-up phase where the first 100 iterations were discarded, the *JIT* compilation threshold had been reached for all hot functions relevant to the microbenchmarks. However, to maintain a clean benchmarking setup, all previously compiled *JIT* code was discarded before actual measurements began. Consequently, during the measured phase, execution resumed in interpretation mode, and functions had to re-trigger *JIT* compilation dynamically. Therefore, not all functions were precompiled at the time of measurement start, and a small portion of execution time accounts for re-compilation overhead.

Our choice of 1000 samples was intended to balance statistical confidence with practical runtime constraints. This number of runs helps reduce measurement noise without excessively prolonging the benchmarking process. Importantly, variations in experimental parameters should not meaningfully affect the core outcome. These parameters include increasing the *JIT* threshold, adjusting the warm-up duration, or modifying the total number of iterations. While our parameters were pragmatically chosen to ease experimentation and enhance clarity, the key performance trends hold across a broad range of reasonable configurations. We validated this by varying key parameters, such as the JIT threshold and warm-up duration, and observing that relative performance trends remained stable, indicating that our conclusions are not sensitive to specific experimental settings.

We ran all the microbenchmarks on *Ubuntu Linux* with Linux kernel version 5.10.16.3 and dedicated hardware with the following parameters: CPU: AMD Ryzen 5 5600X 6-Core Processor; Memory: 64GB DDR4 3200Mhz with calculated precision of 10ns.

6.1.4 | Threats to Validity

Our evaluation has several threats to validity. First, our *JIT* engine does not implement some of the more advanced and intricate optimizations typically found in the instruction scheduling part of the *JIT* engine. The absence of these optimizations might not provide a full spectrum of performance metrics, as such optimizations can play a crucial role in determining the execution efficiency of *JIT* engines. Specifically, these omitted optimizations could further narrow the observed performance difference between stack and register execution.

This suggests that our results could exhibit a more significant difference between the two than what might be observed in real-world scenarios where all optimizations are employed. Furthermore, the decision to disregard certain optimizations might overlook potential interaction effects between them, leading to an incomplete representation of their collective impact. Hence, while our findings provide valuable insights into the *JIT* engine's

performance under specific conditions, they may not comprehensively capture its potential in a fully optimized environment.

Another source of potential variability lies in the underlying hardware architecture. All our benchmarks were conducted on an *x86-64* machine, which features sophisticated *out-of-order execution* and *branch prediction*. Performance might differ on other platforms, especially those with in-order pipelines or different instruction sets, such as *ARM* or *RISC-V*. Although the core trade-off between stack-based and register-based Virtual Machines should remain conceptually consistent–since each of the mentioned *ISAs* implements a similar set of optimizations [44, 45], disparities in areas such as cache hierarchies, branch predictors, or microarchitectural tweaks can still result in performance differences. Moreover, studies involving more complex software [46] show that this is not merely theoretical.

To evaluate whether our findings hold across *non-x86-64* architectures, an optional experiment on alternative architectures could be conducted. For instance, rerunning the same microbenchmarks on modern *ARM* or *RISC-V* systems would help assess whether register-based virtual machines retain their advantage. Such an experiment is outside the scope of the current study, but we view it as an important direction for future work to broaden the applicability of our conclusions.

6.2 | Results

The results of the comparison between individual modes: *AOT*, *JIT* and interpretation–for the register-based virtual machine are shown in Figure 1. To summarise the data: *AOT* compilation is, on average, faster than *JIT* compilation. The specific speed increase varies across different tasks, but overall, *AOT* consistently shows a performance advantage. On average, *AOT* is approximately 2.61 times faster than *JIT*.

JIT compilation outperforms interpretation in terms of speed. The extent of this advantage also varies with different tasks, but *JIT* consistently demonstrates more efficient execution compared to interpretation. On average, *JIT* is approximately 72.7 times faster than interpretation. In conclusion, *AOT* compilation emerges as the fastest method among the three, offering a significant speed advantage over both *JIT* compilation and interpretation. The efficiency gains are task-dependent but are evident across different scenarios.

The reasons behind these differences in performance between AOT and *JIT* modes can be mainly attributed to their compilation strategies. *JIT* compiler engages in partial and dynamic compilation, compiling only the most frequently executed sections of code at runtime. This approach incurs startup overhead, particularly if the *JIT* engine has to monitor and instrument bytecode to identify hot methods, but can produce code highly tailored to the observed execution patterns. By contrast, *AOT* compilation translates all bytecode segments ahead of time, which can lead to longer initial compilation phases but avoids on-the-fly instrumentation and background *JIT* overhead.

Figure 2 represents the same comparison but for the stack-based version of the virtual machine. The trends are similar: *AOT* compilation is, on average, faster than *JIT* compilation. On average, *AOT* is approximately 2.91 times faster than *JIT*. The same is true for *JIT* compilation; in this case, *JIT* is approximately 60.37 times faster than interpretation.

Figures 3, 4, and 5 present the same comparisons in interpretation, *JIT*, and *AOT* modes respectively, but with all execution times normalized to the register-based virtual machine (i.e., register-based = 1.0). This normalization enables clearer relative performance comparisons across the two virtual machine designs.

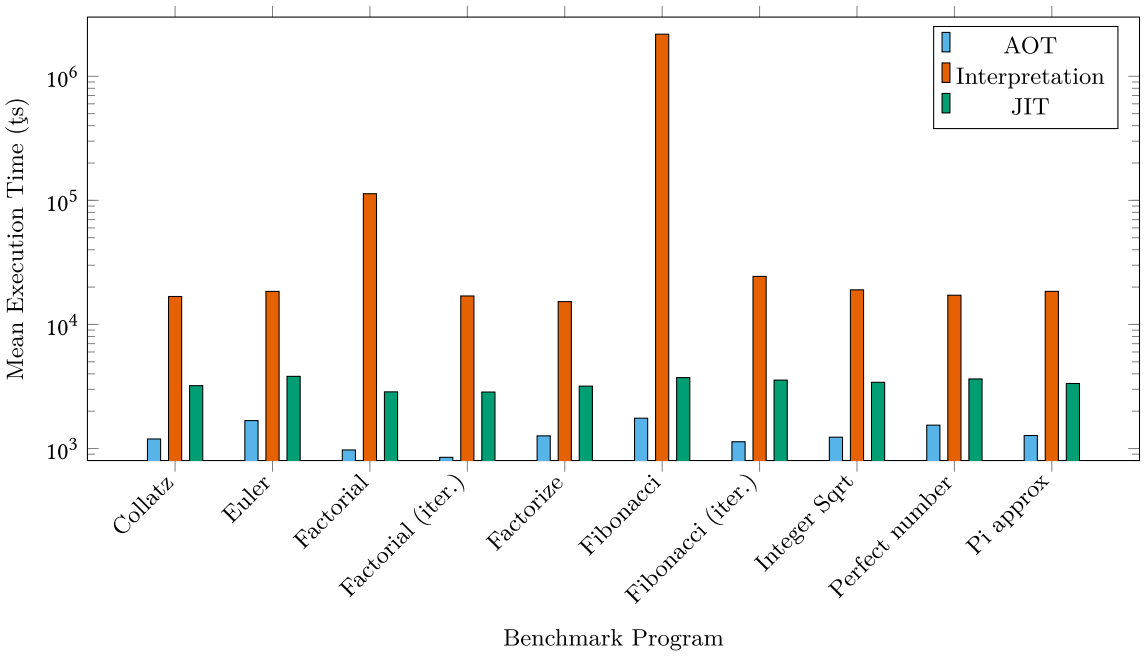

FIGURE 1 | Comparison of mean execution times for register-based version.

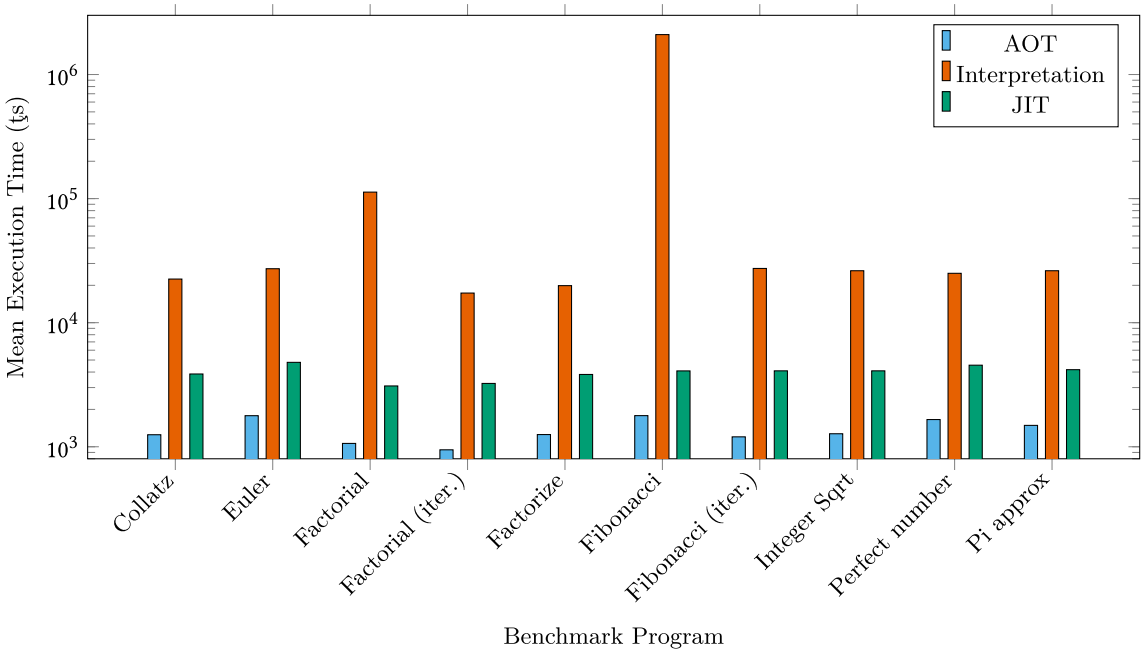

FIGURE 2 | Comparison of mean execution times for stack-based version.

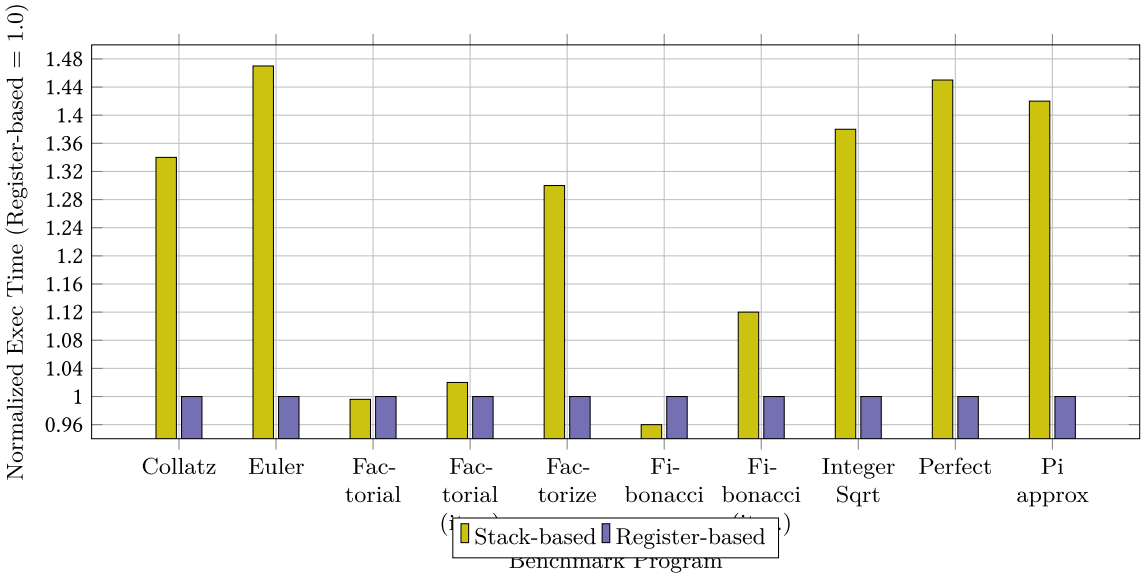

FIGURE 3 | Interpretation mode: stack versus register (normalized to register-based).

A comparison of both virtual machines in interpretation mode is illustrated in Figure 3. The stack-based virtual machine is slightly faster than the register-based machine for recursion-based problems, such as factorial or Fibonacci calculations. In these cases, the stack-based machine is 1.04 times faster. This difference is attributed to the way parameters are passed in the register-based implementation, which uses a dedicated stack, creating a small overhead. However, this overhead is less significant for tasks that do not involve numerous or recursive function calls, such as calculations of pi. In these examples, the register-based machine is 1.31 times faster than the stack-based version.

Figure 4 shows a similar comparison, but both virtual machines are now utilizing a *JIT* engine. The differences are now narrower for recursion-based problems, as they are only 1.09 times faster. For the rest of the examples, the register-based version is 1.21 times faster. Overall, it is 1.18 times faster. We can also state that the *JIT* compilation mitigates a sub-optimal implementation of parameters passing in the register version of the virtual machine.

The *AOT* compilation in Figure 5 mode presents a comparison of the speed of code compiled ahead of time, thereby demonstrating the performance impact of compiled code without interpretation. The register version is marginally faster in every measured problem, including the recursive ones. Generally, it is 1.06 times faster. Suppose we focus only on recursive problems. It is 1.04 times faster; for other problems, it is 1.07 times faster. This can be considered a compilation overhead for the stack version, as there

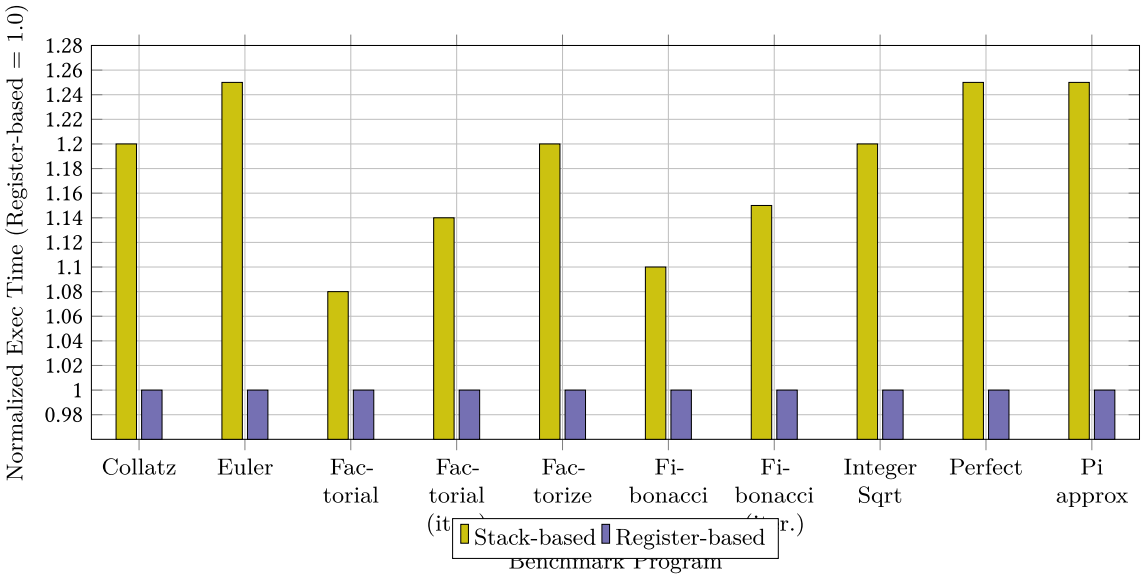

FIGURE 4 | JIT mode: stack versus register (normalized to register-based).

is a need for an extra step to transform it into a proper *IR* for the *JIT* compiler. This suggests plausibility for the idea proposed in [17]—that transforming register-based code into native code will be faster.

7 | Conclusions and Future Work

7.1 | Conclusions

We have implemented prototypes of two virtual machines: one with a register architecture and the other with a stack architecture. Additionally, we developed a *JIT* engine that enables the compilation of bytecode from both virtual machines into native code in both *JIT* and *AOT* modes. The performance of both virtual machines was evaluated through synthetic experiments on selected problems. Each problem was executed on both virtual machines in three modes: interpretation, *AOT*, and *JIT*.

In the case of interpretation, we can confirm that register-based virtual machines are faster than their stack-based variants, as has been stated in [8] and [6]. In our case, the way virtual machines implement function calling plays a role to a certain degree and can slightly distort the results. However, even with that, we can confirm that the register-based machines were, in our study, 1.31 times faster than their stack-based counterparts.

From the results, we can report that *JIT* compilation consistently outperforms interpretation in both stack-based and register-based virtual machines, indicating its efficiency in executing tasks. In the context of *JIT*, the performance differences between stack-based and register-based implementations are more pronounced in recursion-based problems. While the stack-based virtual machine shows a slight edge in recursion, the register-based machine excels in tasks with fewer function calls, such as π calculations. However, this can be attributed to the design choices in the register version of the virtual machine. Specifically, the register-based machine is approximately 1.18 times faster than its stack-based counterpart in general tasks when using a *JIT* engine. However, this advantage narrows in recursion-based problems, with the stack-based machine being only 1.09 times faster. Overall, the *JIT* engine enhances performance in both types of virtual machines. However, the extent of this improvement varies depending on the nature of the tasks and the machine architecture. We can generalize these findings to the conclusion that the performance curve consistently mirrors the performance of the underlying virtual machine as determined by its architecture.

7.2 | Future Work

7.2.1 | Implement a More Advanced JIT Engine Optimization Techniques

Implementing a *JIT* engine with additional optimizations can be more efficient compared to the naive approach we selected. For instance, *SSA* enables more effective optimizations by ensuring that each variable is assigned exactly once and every variable is defined before it is used [47]. This can lead to significant performance improvements during code execution. Additionally, incorporating further optimization techniques, such as dead code elimination or loop unrolling, can further enhance the efficiency of the *JIT* engine.

7.2.2 | Use Different JIT Method

Employ a different *JIT* method than the currently implemented method-based *JIT* engine. A method-based *JIT* engine compiles methods or functions at runtime, which can be effective but may not optimize across method boundaries. Exploring alternative strategies, such as trace-based compilation, could offer performance benefits, as this approach focuses on compiling frequently

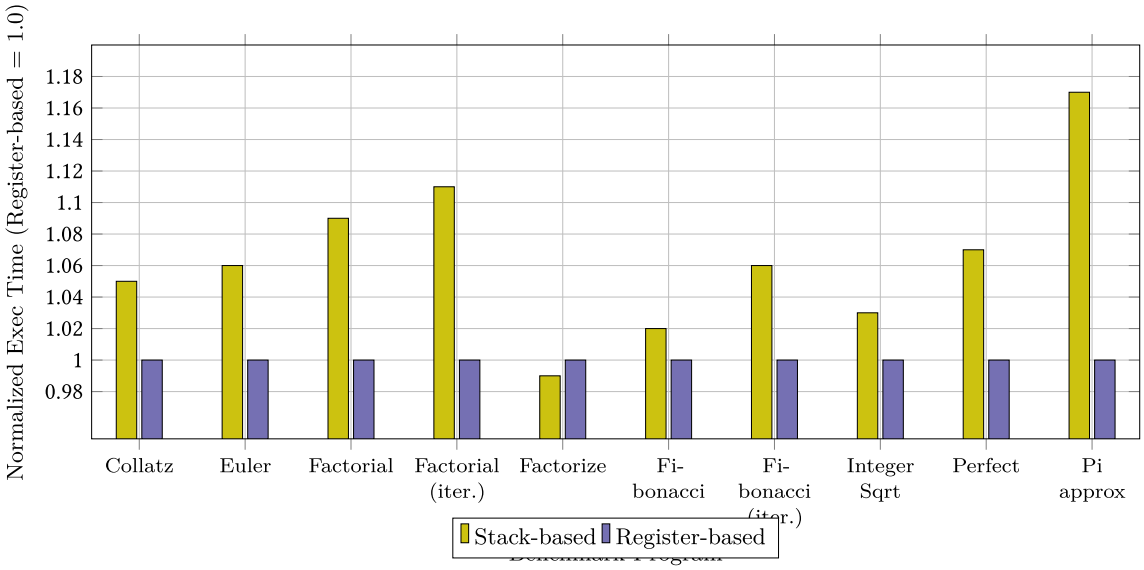

FIGURE 5 | AOT mode: stack versus register (normalized to register-based).

executed paths. This can potentially lead to more efficient execution under certain conditions [27]. The same goes for a combined approach, represented, for example, by a region-based JIT engine.

7.2.3 | Use of Real Programs for Benchmarking

The evaluation of our *JIT* implementation's performance can be further enhanced by incorporating real-world programs instead of relying solely on synthetic benchmarks [39]. Real-world applications offer a more accurate representation of typical usage patterns and can expose performance bottlenecks that synthetic benchmarks might overlook.

Author Contributions

Bohuslav Šimek: conceptualization, investigation, methodology, resources, software, validation, visualization, writing – original draft, writing – review and editing. **Dalibor Fiala:** project administration, supervision, writing – review and editing. **Martin Dostal:** supervision, writing – review and editing.

Acknowledgments

Open access publishing facilitated by Zapadoceska univerzita v Plzni, as part of the Wiley - CzechELib agreement.

Data Availability Statement

The data that support the findings of this study are available from the authors upon request.

References

1. G. J. Myers, "The Case Against Stack-Oriented Instruction Sets," *ACM SIGARCH Computer Architecture News* 6, no. 3 (1977): 7–10.

2. D. M. Bulman, "Stack Computers: An Introduction," *Computer* 10, no. 5 (1977): 18–28.

3. R. Ierusalimschy, L. H. De Figueiredo, and W. C. Filho, "The Implementation of Lua 5.0," *Journal of Universal Computer Science* 11, no. 7 (2005): 1159–1176.

4. G. Ottoni, "HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack," *SIGPLAN Notice* 53, no. 4 (2018): 151–165, https://doi.org/10.1145/3296979.3192374.

5. T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification* (Addison-Wesley, 2013).

6. R. Fang and S. Liu, "A Performance Survey on Stack-Based and Register-Based Virtual Machines," arXiv preprint, arXiv:1611.00467 (2016), https://doi.org/10.48550/arXiv.1611.00467.

7. D. Gregg, A. Beatty, K. Casey, B. Davis, and A. Nisbet, "The Case for Virtual Register Machines," *Science of Computer Programming* 57, no. 3 (2005): 319–338.

8. Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual Machine Showdown: Stack Versus Registers," *ACM Transactions on Architecture and Code Optimization* 4, no. 4 (2008): 1–36.

9. H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon, "Evaluation of Android Dalvik Virtual Machine," in *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (Copenhagen, Denmark) (JTRES'12)* (Association for Computing Machinery, 2012), 115–124, https://doi.org/10.1145/2388936.2388956.

10. J. Aycock, "A Brief History of Just-In-Time," *ACM Computing Surveys (CSUR)* 35, no. 2 (2003): 97–113.

11. Y. Izawa and H. Masuhara, "Amalgamating Different JIT Compilations in a Meta-Tracing JIT Compiler Framework," in *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (Virtual, USA) (DLS 2020)* (Association for Computing Machinery, 2020), 1–15, https://doi.org/10.1145/3426422.3426977.

12. K. Adams, J. Evans, B. Maher, et al., "The Hiphop Virtual Machine," *SIGPLAN Notices* 49, no. 10 (2014): 777–790, https://doi.org/10.1145/2714064.2660199.

13. P. U. Schulthess and E. P. Mumprecht, "Reply to the Case Against Stack-Oriented Instruction Sets," *ACM SIGARCH Computer Architecture News* 6, no. 5 (1977): 24–27.

14. G. J. Myers, "The Evaluation of Expressions in a Storage-to-Storage Architecture," *ACM SIGARCH Computer Architecture News* 6, no. 9 (1978): 20–23.

15. P. Kornerup, B. B. Kristensen, and O. L. Madsen, "Interpretation and Code Generation Based on Intermediate Languages," *Software: Practice and Experience* 10, no. 8 (1980): 635–658.

16. J. R. Bell, "Threaded Code," *Communications of the ACM* 16, no. 6 (1973): 370–372.

17. P. Winterbottom and R. Pike, "The Design of the Inferno Virtual Machine," in *Proceedings of IEEE Compcon'97*, vol. 97 (IEEE Computer Society, 1997), 241–244.

18. A. Rossberg, B. L. Titzer, A. Haas, et al., "Bringing the Web up to Speed With WebAssembly," *Communications of the ACM* 61, no. 12 (2018): 107–115, https://doi.org/10.1145/3282510.

19. W3C Community Group, "Design Rationale," 2020, https://github.com/WebAssembly/design/blob/main/Rationale.md.

20. B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron, "The Case for Virtual Register Machines," in *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators (San Diego, California) (IVME '03)* (Association for Computing Machinery, 2003), 41–49, https://doi.org/10.1145/858570.858575.

21. M. A. Ertl, D. Gregg, A. Krall, and B. Paysan, "Vmgen–A Generator of Efficient Virtual Machine Interpreters," *Software: Practice and Experience* 32, no. 3 (2002): 265–294.

22. I. Piumarta and F. Riccardi, "Optimizing Direct Threaded Code by Selective Inlining," *SIGPLAN Notices* 33, no. 5 may 1998 (1998): 291–300, https://doi.org/10.1145/277652.277743.

23. L. Li and E. L. Gunter, "K-LLVM: A Relatively Complete Semantics of LLVM IR," in *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, ed. R. Hirschfeld and T. Pape (Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020), 7:1–7:29, https://doi.org/10.4230/LIPIcs.ECOOP.2020.7.

24. D. Novillo, "GCC an Architectural Overview, Current Status, and Future Directions," in *Proceedings of the Linux Symposium (Ottawa, Ontario, Canada)*, vol. 2 (Red Hat, Inc., 2006), 185.

25. M. Grimmer, M. Rigger, L. Stadler, R. Schatz, and H. Mössenböck, "An Efficient Native Function Interface for Java," in *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (Stuttgart, Germany) (PPPJ'13)* (Association for Computing Machinery, 2013), 35–44, https://doi.org/10.1145/2500828.2500832.

26. E. Rohou, B. Narasimha Swamy, and A. Seznec, "Branch Prediction and the Performance of Interpreters: Don't Trust Folklore," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (San Francisco, California) (CGO'15)* (IEEE Computer Society, 2015), 103–114.

27. M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz, "Efficient Just-in-Time Execution of Dynamically Typed Languages via Code Specialization Using Precise Runtime Type Inference," Technical Report ICS-TR-07-10. Donald Bren School of Information and Computer Science, University of California, Irvine (2007).

28. A. Gal, C. W. Probst, and M. Franz, "HotpathVM: An Effective JIT Compiler for Resource-Constrained Devices," in *Proceedings of the 2nd International Conference on Virtual Execution Environments (Ottawa, Ontario, Canada) (VEE'06)* (Association for Computing Machinery, 2006), 144–153, https://doi.org/10.1145/1134760.1134780.

29. J. R. Goodman and W.-C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," in *Proceedings of the 2nd International Conference on Supercomputing (St. Malo, France) (ICS'88)* (Association for Computing Machinery, 1988), 442–452, https://doi.org/10.1145/55364.55407.

30. A. S. Tanenbaum, H. Van Staveren, and J. W. Stevenson, "Using Peephole Optimization on Intermediate Code," *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, no. 1 (1982): 21–36.

31. J. W. Davidson and C. W. Fraser, "The Design and Application of a Retargetable Peephole Optimizer," *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2, no. 2 (1980): 191–202.

32. M. Lemerre, "SSA Translation Is an Abstract Interpretation," *Proceedings of the ACM on Programming Languages* 7, no. POPL (2023): 65, https://doi.org/10.1145/3571258.

33. J. von Ronne, N. Wang, and M. Franz, "Interpreting Programs in Static Single Assignment Form," in *Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators (Washington, D.C.) (IVME'04)* (Association for Computing Machinery, 2004), 23–30, https://doi.org/10.1145/1059579.1059585.

34. P. B. Gibbons and S. S. Muchnick, "Efficient Instruction Scheduling for a Pipelined Architecture," *SIGPLAN Notices* 21, no. 7 (1986): 11–16, https://doi.org/10.1145/13310.13312.

35. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register Allocation via Coloring," *Computer Languages* 6, no. 1 (1981): 47–57.

36. M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, no. 5 1999 (1999): 895–913.

37. K. Sagonas and E. Stenman, "Experimental Evaluation and Improvements to Linear Scan Register Allocation," *Software: Practice and Experience* 33, no. 11 (2003): 1003–1034.

38. N. Vazquez, "Announcing Divan!," 2024, https://nikolaivazquez.com/blog/divan/.

39. T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing Wrong Data Without Doing Anything Obviously Wrong!," *ACM SIGPLAN Notices* 44, no. 3 (2009): 265–276.

40. J. Chen and J. Revels, "Robust Benchmarking in Noisy Environments," in *In 2016 IEEE High Performance Extreme Computing Conference (HPEC'16), Twentieth Annual HPEC Conference* (IEEE, 2016), https://doi.org/10.48550/arXiv.1608.04295.

41. The Rust Documentation Team, "Instant in std::time–Rust," 2024, https://doc.rust-lang.org/std/time/struct.Instant.html#underlying-system-calls.

42. OpenJDK Project, "compiler_globals.hpp in OpenJDK," 2025, https://github.com/openjdk/jdk/blob/master/src/hotspot/share/compiler/compiler_globals.hpp.

43. Oracle Corporation, "Java–Java SE 8 Tools Reference," 2015, https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html#BABDDFII.

44. E. Blem, J. Menon, and K. Sankaralingam, "Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures," in *Proceedings–International Symposium on High-Performance Computer Architecture* (IEEE, 2013), 1–12, https://doi.org/10.1109/HPCA.2013.6522302.

45. D. Weaver and S. McIntosh-Smith, "An Empirical Comparison of the RISC-V and AArch64 Instruction Sets," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis (Denver, CO, USA) (SC-W'23)* (Association for Computing Machinery, 2023), 1557–1565, https://doi.org/10.1145/3624062.3624233.

46. V. Dakić, L. Mršić, Z. Kunić, and G. ðambić, "Evaluating ARM and RISC-V Architectures for High-Performance Computing With Docker and Kubernetes," *Electronics* 13, no. 17 (2024): 3494, https://doi.org/10.3390/electronics13173494.

47. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An Efficient Method of Computing Static Single Assignment Form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Austin, Texas, USA) (POPL'89)* (Association for Computing Machinery, 1989), 25–35.