Array Bytecode Support in MicroJIT

Shubham Verma
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
sverma1@unb.ca

Harpreet Kaur
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
harpreet.bamrah@unb.ca

Marius Pirvu
Java JIT Compiler Development
IBM Canada
Toronto, ON, Canada
mpirvu@ca.ibm.com

Kenneth B. Kent
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
ken@unb.ca

Abstract

Eclipse OpenJ9 is a Java virtual machine (JVM), which initially interprets Java programs. OpenJ9 uses Just-in-Time (JIT) compilers—like the default Testarossa JIT (TRJIT)—to translate the bytecodes of the Java program into native code, which executes faster than interpreting. TRJIT is an optimizing compiler, which boosts an application's long-term performance but can increase start-up time due to initial compiling. Despite this overhead, more often than not, start-up time is still improved when compared to an interpreter-only solution. MicroJIT aims to reduce this initial slowdown, making start-up time even quicker. MicroJIT is a non-optimizing, template-based compiler, which aims to reduce compilation overhead and start-up time. Array bytecodes were not supported in the initial implementation of MicroJIT, forcing them to either be interpreted or compiled using TRJIT. This work implements array bytecodes such as, *newarray, aaload, aastore*, in MicroJIT and measures their impact on execution of the programs. The implementation is tested with a regression test suite and the experiments are performed on the DaCapo benchmark suite. The results show that TRJIT along with MicroJIT including array bytecodes support is approximately 4.36x faster than the interpreter and 1.02x faster than the MicroJIT without array bytecodes support. These findings highlight the potential of MicroJIT in improving the performance of Java programs by efficiently handling array bytecodes.

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

ACM ISBN 979-8-4007-0401-7/23/10…$15.00
https://doi.org/10.1145/3623507.3623557

Keywords: bytecode, compiler, interpreter, Java virtual machine, Just-in-Time compiler, templated compiler, optimizing compiler, OpenJ9

ACM Reference Format:
Shubham Verma, Harpreet Kaur, Marius Pirvu, and Kenneth B. Kent. 2023. Array Bytecode Support in MicroJIT. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3623507.3623557

1 Introduction

Java is one of the most popular high-level programming languages because of its ability to run on different platforms. A Java program generates bytecode (.class files), that can be run by any Java Virtual Machine (JVM). The JVM interprets the bytecode converting it into machine-specific code, which runs directly on the CPU. While highly portable, this run-time conversion can be quite inefficient.

Optimization of JVM performance is critical in various real-world scenarios, where Java applications often serve many concurrent users. Improved performance can reduce the resource consumption and operational costs, and can directly enhance the user experience.

JIT compilers are used to improve the performance of Java programs. They perform dynamic compilation of the bytecode into native code at runtime, which can be directly executed by the hardware, avoiding the overhead of interpretation by the JVM [13]. JIT compilers can be either optimizing or non-optimizing, and each comes with its challenges. For example, TestarossaJIT (TRJIT) is the default compiler of the Eclipse OpenJ9 JVM. TRJIT is an optimizing JIT compiler that uses the profile information of the program to optimize the code. TRJIT can often create overhead during runtime and increases start-up time, because it needs to monitor the runtime behavior of the program, select the methods to compile, and allocate memory and CPU resources

for compilation [27]. MicroJIT is a non-optimizing, template-based compiler. By using a template-based approach, MicroJIT bypasses intermediate steps or optimizations that would complicate and slow down the code generation process. As a result, it reduces the compilation time in applications. MicroJIT is a work in progress and the initial focus is on implementing the most commonly used bytecodes. When it encounters an unsupported bytecode, it signals a compilation failure and makes the method fall back to TRJIT, or the interpreter [21]. MicroJIT can potentially be the first level of compilation in multi-level JIT compilation scenarios.

Previous studies have shown that increasing the bytecode support in MicroJIT leads to improved performance [21]. As arrays are very commonly used in Java programs, our aim was to investigate the performance improvement in applications by adding support for array bytecodes to MicroJIT. This will reduce the number of required operations by the interpreter while running methods containing array bytecodes. We implemented array bytecodes, such as newArray, arraylength, aaload, aastore, and successfully ran unit tests on each of the implemented bytecodes. We also ran benchmarks to evaluate the impact of the implemented bytecodes on the perormance of the JVM.

The rest of this paper reads as follows: Section 2 provides background information essential to understand the rest of the paper. Section 3 reviews the related work in this area. Section 4 discusses the design and implementation of our approach. Section 5 evaluates the performance through various benchmarks. Section 6 discusses the results. Section 7 suggests future work and Section 8 concludes the paper.

2 Background

The following background concepts are essential for this work. These concepts include Java, Java Virtual Machine (JVM), Just-In-Time (JIT) Compilation, Eclipse OpenJ9, and Arraylets.

2.1 Java

Java is an object-oriented language that makes software easily portable across operating systems [28], needing only a Java runtime environment (JRE) to run Java programs. The program's life cycle has three steps: write source code in a *.java* file, compile it into machine-independent bytecodes using *javac*, and then run it on the JVM, which converts bytecodes into CPU-executable instructions.

2.2 JVM

The Java Virtual Machine (JVM) is a core component of the Java runtime. A virtual machine behaves like a real machine, but it has no physical structure. To form a virtual machine, virtualizing software is combined with the real machine [28]. The virtual machine may or may not have the same amount or type of resources as the real machine. Processors of virtual machines can have different instruction sets as compared to the real machine. Often, the virtual machine is not able to provide similar performance as the physical machine. The JVM reads and executes the class file consisting of bytecodes, and generates machine code, which can then be executed by the CPU of the local machine [4].

2.2.1 JVM Structure. A JVM has four important parts: the ClassLoader subsystem, the Runtime Data Area, the Execution Engine, and the Java Native Method Interface [4].

The Java compiler *javac* turns *.java* files into *bytecode*, stored in the JVM's Method Area [4]. The JVM memory, also called runtime data areas, includes the Method Area, Heap Area, Stack Area, Program Counter Registers, and Native Method Stack. Each area serves a specific function: the Method Area stores class structures; the Heap Area is for shared class objects; the Stack Area holds thread-specific stacks with frames for variables and partial results; Program Counter Registers track current JVM instructions; and the Native Method Stack holds native code for each thread.

The execution engine reads and executes bytecodes from the class file and generates machine-readable code [28]. At its core, it consists of an interpreter. It also consists of a Garbage Collector and/or compilers such as JIT or Ahead-Of-Time (AOT). The interpreter reads each line of bytecode from the class file and executes it. This bytecode-by-bytecode process affects the performance of the code and results in it running slower than JIT and AOT compiled versions. The Java Native Method Interface (JNI) allows Java applications to interact with other applications written in lower-level languages like Assembly, C, and C++ [4].

2.3 JIT Compilation

JIT compilation is a technique that compiles computer code during execution of a program, rather than before execution [14, 16, 23]. JIT compilation combines the advantages of both compilation and interpretation.

JIT compilation significantly improves the performance of the interpreted program running on the VM [16]. A JIT compiles and translates program code from the guest machine into native code at runtime, and executes it directly on the host machine, which is faster than interpreting it. The JIT analyzes the code as it runs and makes dynamic optimizations based on the execution patterns of the program.

2.3.1 Use of multiple JIT compilers. Although using a JIT compiler can improve some JVM performance, it also creates overhead during runtime and increases start-up time. The Jalapeño virtual machine [15] and OpenJ9 [2] [29] are some examples of virtual machines where a second JIT was used to reduce the start-up time and/or reduce overhead. Smalltalk-80 used dynamically translated code of the v-machine to native code, that was cached and paged when it was required [22]. In the OCaml data structure, bytecode

programs are translated into SSA-based intermediate code with some optimization, then translated into JavaScript to work on the web [30]. It may create overhead during the intermediate code phase but will optimize the code for faster performance. The V8 JavaScript engine is another example, which makes use of two JIT compilers, TurboFan and Liftoff, to optimize JavaScript and WebAssembly code based on execution patterns and complexity [8]. This approach enhances performance by tailoring optimizations to different code scenarios.

2.4 Eclipse OpenJ9

OpenJ9 was contributed by IBM to the Eclipse project [2], based on their J9 JVM. It was implemented independently, from other JVM code, and used the Java Virtual Machine Specification. It is built upon the open-source runtime component framework named Eclipse OMR [20]. It was designed to improve performance so, it can have a quick start-up, low overhead, and high throughput.

Coffin et al. showed the process of a method being compiled in OpenJ9 [20]. When OpenJ9 encounters a method, the execution engine checks if the method has already been JIT compiled. If the method has already been compiled then, it executes the compiled code directly, otherwise it decreases the method invocation counter. The method is sent to the Interpreter if the invocation count of the method has not reached a certain set threshold, which means the method can be compiled, but it is not ready to be compiled yet. Once the invocation counter reaches a predetermined threshold, the appropriate JIT compiler is used.

2.4.1 Testarossa JIT. Testarossa JIT (TRJIT) is the default JIT compiler used in OpenJ9 [2]. It has to balance between allocating more resources to compile many methods with the best optimization level in a short time, and saving resources for the application to run its tasks, as it compiles code at runtime [29]. It uses selective compilation, where only frequently used methods are compiled, which helps balance resource usage and optimize code performance. TRJIT is dedicated to providing code optimization on several levels ranging from cold to scorching. It also uses the shared class cache (SCC) and dynamic ahead-of-time (AOT) compilation technology to reduce overhead, which generally improves start-up time compared to interpretation [26]. In certain cases, TRJIT may not enhance start-up time. For instance, when dealing with complex or large application code, TRJIT's compilation process could be time-consuming and memory-intensive, leading to longer start-up times due to increased overhead. Also, if the application code is not frequently executed or reused, TRJIT may not be able to benefit from the SCC or AOT features, as the compiled code may not be cached or shared.

2.5 Arraylets

In a JVM, memory for objects (including arrays) is allocated on the heap. While it is desirable for objects to be stored in a contiguous region of memory [10], GC policies like *balanced* and *metronome* divide the heap space for running programs into regions [3]. If an array object is smaller than the region, then it will be allocated in a contiguous manner, but if an array is larger than the region, then the array needs to be allocated in different regions by using arraylets. An arraylet is a data structure, which has a spine containing some metadata and leaves. The spine is connected to leaves through arrayoids. Arrayoids are pointers to the respective arraylet leaves. Representation of arrayoids depends upon the implementation of the arraylets. Following are some techniques for structuring arraylets:

- Keep all the arrayoids of the respective arraylet leaves on the spine.
- Have a separate arraylet leaf from the spine, containing arrayoids of other respective leaves.
- Keep the arrayoid to the first leaf on the spine. The first leaf contains an arrayoid to the second leaf, the second leaf contains an arrayoid the to third leaf and so on, until the last leaf.

3 Related Work

In this section, we examine GraalVM Native Image, which involves initializing certain application components during the build process. Next, we explore the usage of two JIT compilers in a single JVM like the Jalapeño Virtual Machine, V8, and OpenJ9. They all use a combination of JIT compilers to optimize the performance of the application.

3.1 GraalVM Native Image

GraalVM Native Image is a tool that optimizes the performance of Java applications by using static analysis, heap snapshotting, and ahead-of-time compilation to create an optimized executable file [31]. This tool reduces the start-up time and memory footprint required by the application at runtime. Traditional JIT compilation compiles bytecode at runtime, whereas GraalVM initializes parts of the application at build time. GraalVM provides significant advantages for optimizing Java applications, but eliminates a whole category of Java applications by enforcing a "closed-world assumption" [11, 12]. The closed-world assumption means that all the code that is required by the application at run time is known at build time, and no new code can be loaded or generated at run time [9, 12]. The dynamic nature of Java gives a lot of power to applications at runtime, such as reflections, class loading, and class construction.

3.2 The Jalapeño Virtual Machine

In 2000, Alpern et al. talked about "The Jalapeño virtual machine", where they used multiple compilers to speed up the

starting time [15]. The idea of using two compilers in the same VM is that the newly added compiler can help reduce start-time while the older JIT continues to optimize code later in process. The Jalapeño virtual machine was a Java server VM developed at IBM. It supported multiple compilers that could generate code of different quality and speed. These compilers cooperated through adaptive optimization, which selected the best compiler based on factors like execution frequency, code size, and optimization level. It had a baseline compiler for quick code generation and an optimizing compiler for highly optimized code. It demonstrated improved start-up time and steady-state performance for Java applications through adaptive optimization.

3.3 V8: The JavaScript and WebAssembly Engine

V8 is an open-source high-performance JavaScript and WebAssembly engine developed by Google [1]. It is used in the Chrome web browser, and Node.js, a JavaScript runtime for server-side applications. V8 compiles JavaScript and WebAssembly code to native code for execution, using JIT compilation and various optimization techniques. It uses two compilers, TurboFan, which is an optimizing compiler, and Liftoff, which is a baseline compiler [8]. TurboFan excels at performing advanced optimizations like inlining, code motion, instruction combining, and sophisticated register allocation, for both JavaScript and WebAssembly [5]. On the other hand, Liftoff, a baseline compiler, swiftly generates highly efficient WebAssembly code by processing the bytecode in a single pass, without constructing an intermediate representation or performing complex optimizations [5, 8]. TurboFan produces faster, high-quality code but takes longer to compile and uses more memory. Liftoff produces decent-quality code that compiles quickly and uses less memory. To take advantage of the strengths of both compilers, V8 initially uses Liftoff to compile WebAssembly functions lazily to reduce start-up time and memory usage. Then, V8 recompiles frequently executed functions in the background with TurboFan, to improve performance.

3.4 MicroJIT in OpenJ9

Sogaro et al. presented an approach where one compiler, TRJIT, is responsible for improving optimization and the other, MicroJIT, primarily focuses on improving compilation time at the expense of lower optimization [29].

MicroJIT is a lightweight JIT compiler, that uses a single compilation pass [29]. It is built to improve applications' start-up time. As mentioned earlier, OpenJ9 uses TRJIT as its default JIT compiler to optimize code, at the expense of start-up time, so Sogaro et al. proposed to add MicroJIT along with TRJIT to improve start-up time. MicroJIT in Java Mobile Edition (J2ME) showed improvement in start-up time [17]. This structure allowed bytecodes to be translated using predefined machine code templates, which would eliminate the intermediary phase [20]. Adding extended bytecode support in MicroJIT can reduce the number of context switch cycles with the interpreter [21].

If MicroJIT is enabled, then the method is compiled with MicroJIT and generates non-optimizing code, otherwise the method is compiled with TRJIT and generates optimized code [20]. MicroJIT would normally be set to a lower invocation threshold count and TRJIT, a higher invocation threshold count.

The default JIT (TRJIT) can also support fast compilation, with non-optimized code but with the possible cost of overhead for constrained environments because it generates an intermediate language during its intermediary phase [20]. Adding MicroJIT can reduce this overhead in constrained environments because its template-based structure allows bytecodes to be translated using predefined machine code templates, which would eliminate the intermediary phase.

Coffin et al. showed that MicroJIT, is faster to execute methods as compared to the interpreter [20]. In another study, Coffin et al. showed that the performance of the JVM can be improved by increasing the number of supported bytecodes in MicroJIT [21]. They showed that, by adding support for bytecodes, the number of context switches reduces, and minimizes the required number of operations. Kent and Serra explained that switching between the interpreter and the two environments can be costly [25].

4 Design and Implementation

To improve start-up time of the OpenJ9 in resource-constrained environments, MicroJIT was introduced for Java 8 on the x86-64 Linux platform [20]. When a class is loaded, each method in the class will receive an initial invocation count that represents the number of times a method is invoked before being compiled. When MicroJIT is enabled, the default value of this initial invocation count is set to 20 as this value was shown to be optimal in many scenarios [24, 29].

Figure 1 shows the overall control flow of the compilation. It starts with setting up the method invocation counters to the compilation threshold [20]. The JVM interprets the method until the invocation count reaches zero. If MicroJIT is not enabled, then the method is compiled by TRJIT, and executed. If MicroJIT is enabled, and the method invocation count for MicroJIT is lower than the method invocation count for TRJIT, then the method is compiled by MicroJIT, and executed. Otherwise, the method is compiled by TRJIT (when its threshold is reached), and executed. The purpose of this approach is to address situations where the user sets low counts for TRJIT. In such cases, the MicroJIT counts might be higher, contradicting the user's intention. To mitigate this, we choose the minimum count between the two to ensure consistency [24].

MicroJIT has multiple phases of compilation: metadata generation, pre-prologue generation, prologue generation,

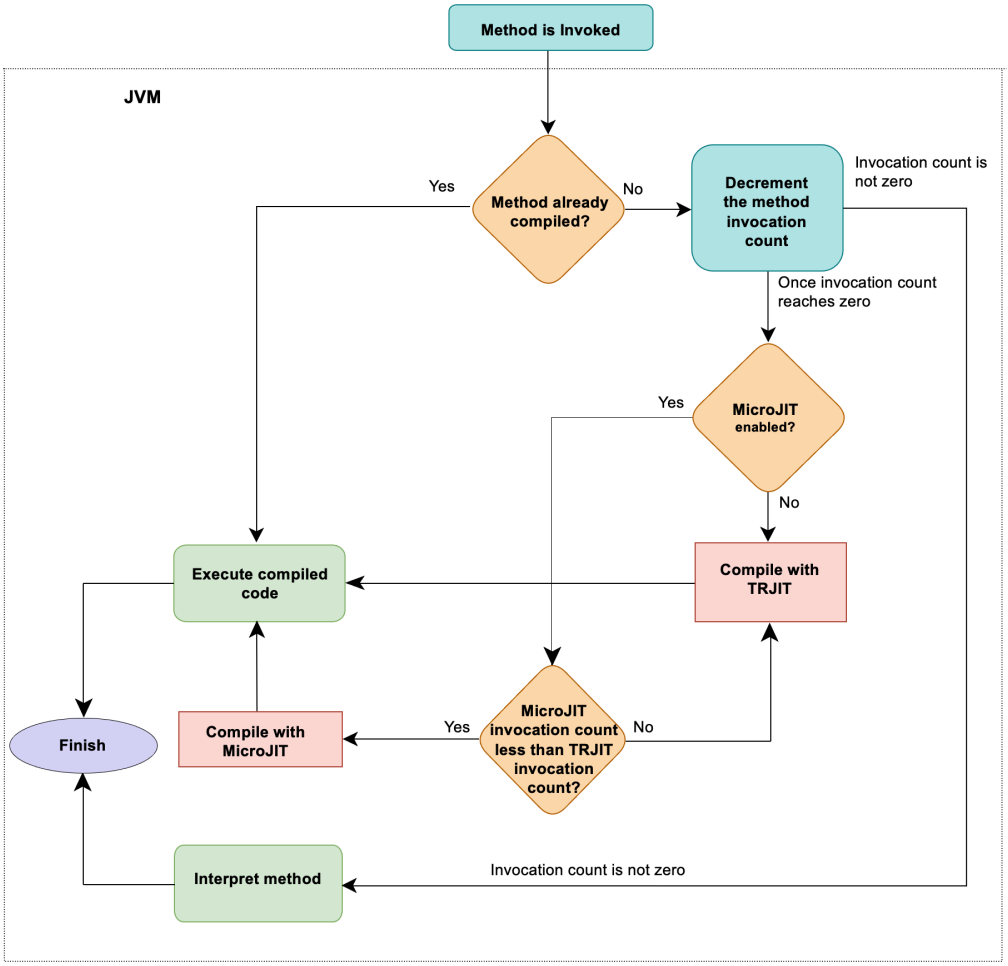


Figure 1. Compilation control flow in OpenJ9 (with MicroJIT)

and body generation [20]. Inside MicroJIT, the code is generated using pre-defined templates, and the template is a sequence of assembly instructions that are used to generate the code. To add support for new bytecodes, such as newArray, iaload, lastore, we need to add new templates for the bytecodes, and patch the template with buffer addresses. A buffer address is location where the code is generated. The buffer address is passed to the template as an argument.

To test the MicroJIT, we used the microjit-tests regression framework [20]. We added unit tests in the regression framework to test the functionality of the implemented array bytecodes, ensuring that the modifications made to the OpenJ9 code base did not introduce any regressions or issues in the existing code.

The following bytecodes are necessary for the complete implementation of Java array support in MicroJIT:

`arraylength`: Loads the length of the array.
`iaload`: Loads the integer value from array.
`laload`: Loads the long value from array.
`aaload`: Loads the reference from array.
`caload`: Loads the char value from array.
`saload`: Loads the short value from array.
`baload`: Loads the byte or boolean value from array.
`daload`: Loads the double from array.
`faload`: Loads the float value from array.
`iastore`: Stores the integer value into the integer array.
`lastore`: Stores the long value into the long array.
`aastore`: Stores the reference into the reference array.
`castore`: Stores the char value into the char array.
`sastore`: Stores the short value into the short array.
`dastore`: Stores the double value into the double array.
`fastore`: Stores the float value into the float array.
`bastore`: Stores the byte or boolean value into the respective byte or boolean array.
`newarray`: Creates a new array of primitive type.

Templates for each of these bytcodes were written in Netwide Assembler (NASM)—an assembler and disassembler for the Intel x86 architecture [7]. It is capable of generating code

for different operating systems and running on various platforms. The syntax of NASM is similar to the Intel syntax, and it supports various features such as macros, multiple output formats, and optimization.

Listing 1 shows examples of macros for the bytecodes. Macros in NASM enable us to define reusable pieces of code or data sections, which then can be invoked by a name [6]. Programmers can avoid repetitive code, easily maintain and invoke these predefined sections with different parameters as needed.

Listing 2 shows the `iastore` bytecode template written in NASM. Whenever a method is compiled in MicroJIT that requires storing an integer value into an integer array, MicroJIT will invoke the predefined `iastore` template.

```
%macro _32bit_slot_stack_to_rXX 2
mov %1d, dword [r10 + %2]
%endmacro

%macro _64bit_slot_stack_to_rXX 2
mov %{1}, qword [r10 + %2]
%endmacro

%macro pop_single_slot 0
add r10, 8
%endmacro
```

Listing 1. Macros for the bytecodes

```
template_start iastoreTemplate
   _32bit_slot_stack_to_rXX r11,0
   pop_single_slot
   _32bit_slot_stack_to_rXX r12,0
   pop_single_slot
   _64bit_slot_stack_to_rXX rax,0
   pop_single_slot
   mov dword [rax+4*r12+0x10], r11d
template_end iastoreTemplate
```

Listing 2. Example iastore Template

Writing the NASM code required good understanding of the x86 architecture and the NASM syntax. The NASM template for the newarray bytecode was one of the most challenging array templates to write, because the template needed to handle the allocation of both, contiguous and non-contiguous array objects. Along with allocation, the template also needed to handle different data types of the arrays. Additionally, NASM code is challenging to debug due to its low-level nature, lack of high-level abstractions, limited debugging features, and the need for manual handling of registers and memory addresses.

5 Evaluation

This section provides an overview of the experimental setup, the tools employed for benchmarking, the implementation utilized, and the comparison baselines employed in the performance evaluation.

5.1 Experimental Setup

The experimental setup involved conducting the experiments on a machine equipped with an *Intel(R) Core(TM) i7-8700 CPU* running at *3.20GHz*, kernel version of *5.4.0-125-generic*, system architecture of *x86_64 (64-bit)* and 32GB of installed RAM and had *6 cores* and *12 CPU* threads for handling the computational workloads. The execution times were recorded in log files and analyzed using bash scripts.

5.2 Regression Test Framework

The MicroJIT design underwent unit testing using the microjit-tests regression framework [20]. Tests are written in Java, compiled by the selected JIT compiler, and executed. These tests cover a comprehensive set of edge cases for Java bytecodes with MicroJIT. Support was added to the test suite for testing array bytecodes by making changes to the test runner script and adding argument converter classes.

5.3 Benchmarking

The DaCapo Benchmark suite is designed to evaluate the performance of JVMs, memory, and compilers. It comprises a range of workloads that consists of non-trivial memory allocations and garbage collection behaviors [18].

The experiments were conducted using the DaCapo 9.12 MR1 maintenance release version. The study employed a diverse range of workloads:

- **fop** - Parses an XSL-FO file and creates an encrypted PDF.
- **luindex** - Uses Lucene to index documents like Shakespeare's works.
- **sunflow** - Uses raytracing to create realistic images.
- **lusearch-fix** - Uses Lucene for text searches on classic texts.
- **avrora** - Provides simulation tools for AVR micro-controllers.
- **h2** - Executes transactions on a simulated banking app.
- **jython** - Executes Python scripts.
- **xalan** - Parses an XSLT file to produce an XML file.
- **eclipse** - Uses an integrated development environment (IDE).
- **pmd** - Parses Java source code files.

6 Results and Discussion

Figure 2 and Table 1 collectively summarize two key dimensions: the average execution time and the normalized performance metrics. To analyze the early iterations in the start-up phase, the average execution times are measured over the first 10 iterations of each of the DaCapo benchmarks.

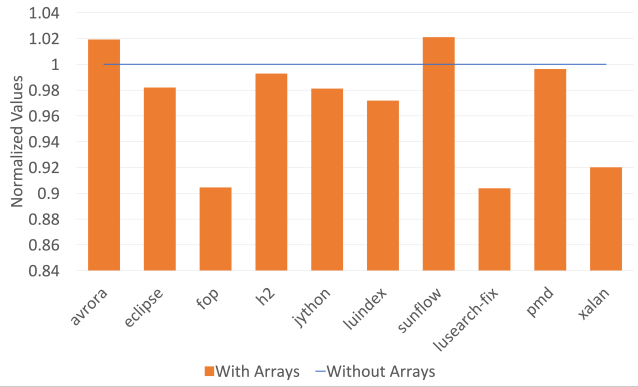


Figure 2. Average execution of TRJIT along with MicroJIT; with and without array support

Table 1. Average Execution Times of TRJIT with MicroJIT enabled—the lesser, the better

Workload	Without arrays (msec)	With arrays (msec)	Gain/Loss (normalized)
avrora	2010.6	2049.3	1.019
eclipse	45 341.4	44 527.0	0.982
fop	292.3	264.4	0.905
h2	2899.6	2878.8	0.993
jython	1841.3	1806.7	0.981
luindex	930.5	904.3	0.972
sunflow	1970.5	2012.1	1.021
lusearch-fix	452.4	408.9	0.904
pmd	551.5	549.5	0.996
xalan	609.9	561.2	0.920
Avg	5690.0	5596.2	0.969

The normalized performance metrics are calculated by dividing the execution time of the MicroJIT with array support by the execution time of the MicroJIT without array support. The column labeled "Without arrays (msec)" provides the baseline execution times without array support, while "With arrays (msec)" indicates the execution times after array implementation has been added. The "Gain/Loss" column shows the normalized performance metric—where a value greater than 1 suggests performance degradation and a value less than 1 indicates a performance improvement. For instance, the 'avrora' benchmark shows a slight increase in execution time by a normalized value of 1.019 after adding array support. Conversely, benchmarks like 'fop' and 'lusearch-fix' demonstrate significant performance gains with improvements shown by 0.905 and 0.904, respectively. On average, the normalized execution time across all benchmarks is approximately 0.969, suggesting a modest performance improvement of about 3.1% with the added array support.

The frequency of array bytecodes in each workload is shown in Table 2. We can see that the frequency of array bytecodes varies from workload to workload. For example, eclipse has the highest frequency of array bytecodes, which is not surprising because eclipse is a large application. Table 3 shows that MicroJIT is compiling more methods of each workload after array support was added to it. It is evident from Table 3 that eclipse was able to compile more methods with the MicroJIT compiler with array support than without array support. Given that, whenever the JVM encounters an unsupported bytecode, it abandons the entire method to the very slow interpreter. Therefore, it is not surprising to see that the performance of eclipse improved after adding support for array bytecodes in the MicroJIT compiler. Also notice, that workloads such as avrora, fop, sunflow, lusearch-fix, pmd, and xalan have very low frequency of array bytecodes, which is also reflected in the number of methods compiled with MicroJIT.

Figure 3 shows the comparison of the average execution time for each of the JVM options, based on an initial 10 iterations of each workload, without discarding any iteration. The results show that the OpenJ9 JVM performs better with MicroJIT enabled, when tested with DaCapo workloads. We ran our experiments with Interpreter only, TRJIT with default warm optimization, TRJIT with cold optimization, TRJIT with no-optimization, and TRJIT with MicroJIT enabled. TRJIT is a dynamic JIT compiler and uses warm optimization by default, but it adjusts the level of optimization automatically based on the execution behavior of the application [29]. By setting the *-Xjit:optLevel=cold* option, the TRJIT compiler performs minimal optimizations, which helps to reduce start-up time of short-lived applications over peak performance [20]. By setting the *-Xjit:optLevel=noOpt* option, despite its name, the TRJIT compiler still performs very limited optimization, allowing reduction of the start-up time of long-lived applications over peak performance [20, 24].

By using the MicroJIT compiler along with the TRJIT compiler, we can reduce the start-up time of the application [29]. MicroJIT compiles the code for less frequently run methods, while TRJIT compiles the more frequently run methods. This allows the JVM to optimize the code only for the frequently executed methods, which helps to reduce the start-up time of the application.

As we can see in Figure 3, interpreted code took a very long time to execute the workloads, while the default TRJIT, cold optimization, no-optimization, and MicroJIT enabled required less time to execute the code as compared to interpreted code. We also observed that the MicroJIT enabled TRJIT required the least amount of time to execute the code, except sunflow. The sunflow could contain a large number of unsupported non-array bytecodes, which are causing methods to fall back to the interpreter. TRJIT with cold optimization and no-optimization required more time to execute the

Table 2. Number of array bytecodes in each workload

Bytecode Name	avrora	eclipse	fop	h2	jython	luindex	sunflow	lusearch-fix	pmd	xalan
newarray	89	2167	68	218	558	153	63	54	95	46
arraylength	184	4029	159	358	623	309	92	89	153	83
iaload	38	1169	39	112	160	148	71	61	32	7
laload	10	381	16	25	17	15	31	9	10	7
faload	0	7	0	0	1	0	16	0	0	0
daload	1	20	0	0	13	0	0	0	0	0
aaload	139	4521	86	270	585	334	104	51	107	57
baload	41	215	32	54	81	91	29	31	38	17
caload	105	1596	102	170	148	141	24	28	109	28
saload	1	4	0	0	13	0	0	0	0	0
iastore	17	526	14	80	90	102	20	62	18	2
lastore	3	118	3	5	5	6	7	3	3	3
fastore	0	2	0	0	0	0	24	0	0	0
dastore	0	2	0	0	0	0	0	0	0	0
aastore	44	1948	37	339	944	105	33	22	80	21
bastore	37	1304	30	46	126	101	43	28	32	28
castore	51	275	46	135	86	80	30	34	40	30
sastore	0	1	0	0	2	0	0	0	0	0
Total	760	18285	632	1812	3452	1585	587	472	717	329

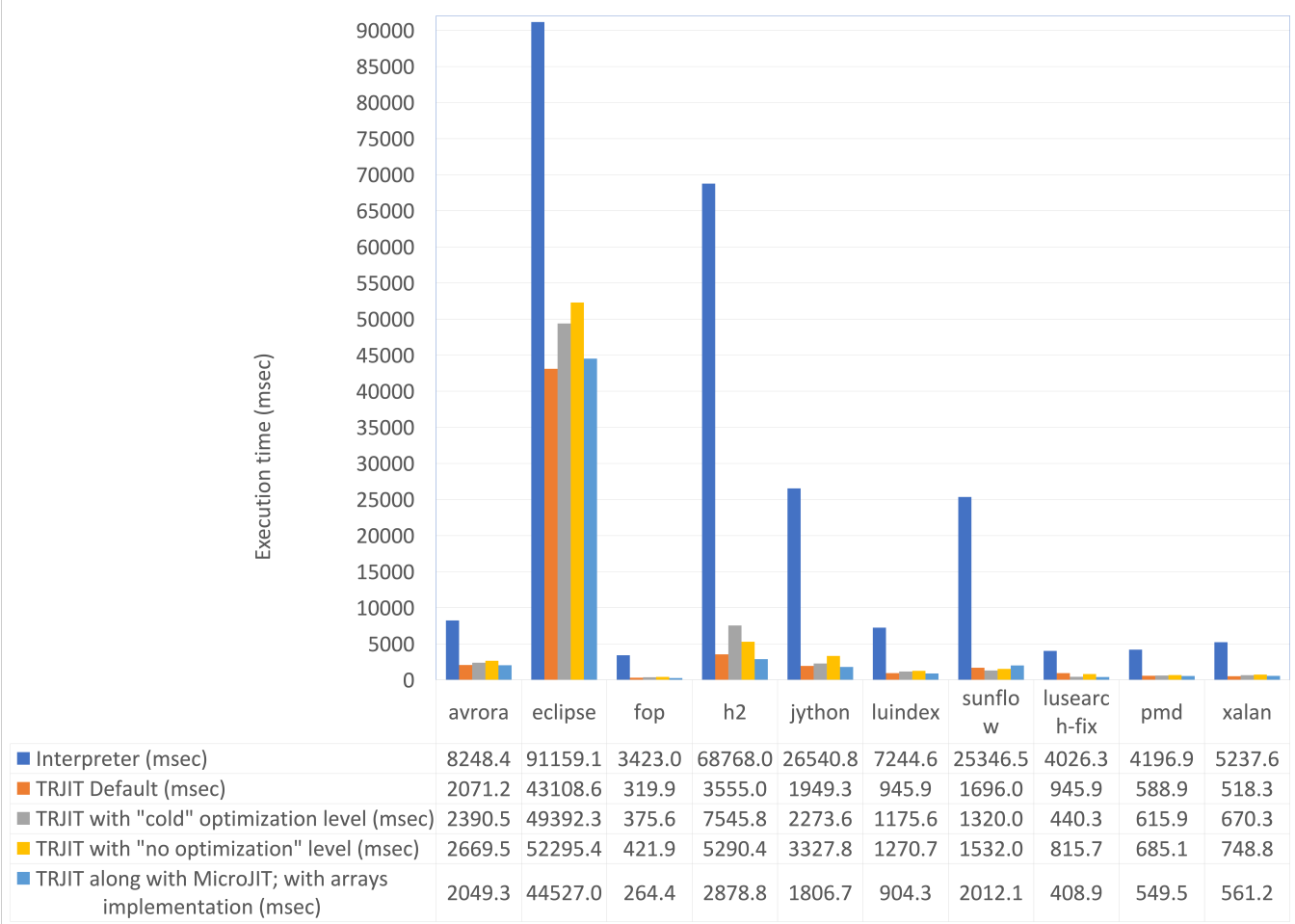
Table 3. Number of methods compiled with MicroJIT

Workload	Without arrays implementation	With arrays implementation	Additional methods
avrora	48	49	1
eclipse	555	580	25
fop	27	29	2
h2	101	107	6
jython	100	103	3
luindex	52	59	7
sunflow	19	20	1
lusearch-fix	19	21	2
pmd	29	31	2
xalan	19	20	1

code as compared to the default TRJIT. One possible reason could be that the list of benchmark workloads is more suitable for the default TRJIT compiler and does not provide significant benefit to the TRJIT with cold optimization and no-optimization. Another possible reason could be that benchmarks were not short enough for the TRJIT overhead to show up.

The significance of adding array bytecodes support in MicroJIT compiler extends beyond just execution performance gains. Many of the resource constrained devices such as IoT devices and embedded systems use Java as their primary programming language, where efficient execution of array operations is critical [19]. Array operations are fundamentally very common in Java application development, and widely used for data manipulation and storage, so adding support for array bytecodes can have substantial impact on overall application speed and resource utilization. Furthermore, by combining MicroJIT with the TRJIT compiler, this study showcases the potential of using a specialized JIT compiler like MicroJIT in combination to achieve a finer level of optimization based on code execution behavior [20].

However, this study also has some limitations. We ran our experiments with only 10 warm-up iterations, which might not be representative of real-world scenarios, where the execution pattern of applications may be more dynamic, including start-up timing. Our study used a diverse set of benchmark workloads, but still they may not cover all the



	avrora	eclipse	fop	h2	jython	luindex	sunflow	lusearch-fix	pmd	xalan
Interpreter (msec)	8248.4	91159.1	3423.0	68768.0	26540.8	7244.6	25346.5	4026.3	4196.9	5237.6
TRJIT Default (msec)	2071.2	43108.6	319.9	3555.0	1949.3	945.9	1696.0	945.9	588.9	518.3
TRJIT with "cold" optimization level (msec)	2390.5	49392.3	375.6	7545.8	2273.6	1175.6	1320.0	440.3	615.9	670.3
TRJIT with "no optimization" level (msec)	2669.5	52295.4	421.9	5290.4	3327.8	1270.7	1532.0	815.7	685.1	748.8
TRJIT along with MicroJIT; with arrays implementation (msec)	2049.3	44527.0	264.4	2878.8	1806.7	904.3	2012.1	408.9	549.5	561.2

Figure 3. Average execution time of workloads

possible scenarios of array operations. Furthermore, our research is based on MicroJIT and TRJIT, and may not generalize well to other JIT compilers [20]. In the context of resource constrained environments, it is crucial to consider the trade-offs between memory consumption and overall performance gains.

7 Future Work

Future work could explore the adaptive strategies for determining the number of warm-up iterations required based on the runtime-behavior, which might help in understanding some of the unexpected results we observed in our study when using TRJIT options. Future work could also explore the impact of adding support for other array operations such as *anewarray* and *multianewarray* in the MicroJIT compiler. We can also explore using array bound check elimination optimization [32]. There is potential to make start-up time faster, if bound checking overhead can be removed during the initialization routines, where some arrays or such data structures may be involved.

8 Conclusion

In this study, we investigated the impact of adding bytecode support for array operations in the MicroJIT compiler on the performance of Java applications. The results of our study showed that adding support for array bytecodes can reduce the execution time when MicroJIT was enabled. By handling the array bytecodes, MicroJIT can make Java programs faster, particularly in resource-constrained environments. Increasing support of array bytecodes in the MicroJIT compiler is a good step towards making Java more efficient for resource-constrained environments, and laying the foundation for further optimization and research in this area.

Acknowledgments

The authors express their appreciation for the support and resources provided by the Centre for Advanced Studies—Atlantic at the University of New Brunswick. The researchers are grateful for the collaboration and facilities offered by

CAS Atlantic, which have been instrumental in carrying out this study.

Special gratitude is extended to the Atlantic Canada Opportunities Agency (ACOA) for their generous funding support through the Atlantic Innovation Fund (AIF) program. Additionally, the authors would like to acknowledge the New Brunswick Innovation Foundation for their valuable contribution to this endeavor.

The authors wish to express their sincere thanks to all colleagues and individuals who have contributed to this research effort, both directly and indirectly. Special thanks to Stephen MacKay and DeVerne Jones for their support and guidance throughout the project. Their expertise, insights, and collaboration have greatly enriched the outcomes of this study.

References

[1] [n. d.]. Documentation · V8. https://v8.dev/docs Accessed on July 18, 2023.

[2] [n. d.]. Eclipse OpenJ9. https://www.eclipse.org/openj9/. Accessed on June 27, 2022.

[3] [n. d.]. GC policies -. https://eclipse.dev/openj9/docs/gc/ Accessed on September 6, 2023.

[4] [n. d.]. The Java® Virtual Machine Specification. https://docs.oracle.com/javase/specs/jvms/se8/html/index.html Accessed on July 29, 2022.

[5] [n. d.]. Liftoff: a new baseline compiler for WebAssembly in V8 · V8. https://v8.dev/blog/liftoff Accessed on July 18, 2023.

[6] [n. d.]. NASM Macros. https://www.cs.uaf.edu/courses/cs301/2014-fall/notes/asm-macros/ Accessed on June 28, 2023.

[7] [n. d.]. NASM The Netwide Assembler. https://www.nasm.us/doc/ Accessed on June 28, 2023.

[8] [n. d.]. WebAssembly compilation pipeline · V8. https://v8.dev/docs/wasm-compilation-pipeline Accessed on July 18, 2023.

[9] [n. d.]. Why GraalVM? https://www.graalvm.org/why-graalvm/ Accessed on July 17, 2023.

[10] 2019. A Guide to Double Map Arraylets. https://blog.openj9.org/2019/05/01/double-map-arraylets/

[11] 2022. Project Leyden Delays OpenJDK AOT Compiler, Optimizes JIT Compiler Instead. https://www.infoq.com/news/2022/06/project-leyden-delays-aot/ Accessed on March 7, 2023.

[12] 2022. Standardizing Native Java: Aligning GraalVM and OpenJDK. https://www.infoq.com/articles/native-java-aligning/ Accessed: March 7, 2023.

[13] 2023. The JIT compiler - IBM Documentation. https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler

[14] Gills Alexander S. [n. d.]. just-in-time compiler (JIT). https://www.theserverside.com/definition/just-in-time-compiler-JIT Accessed on March 21, 2023.

[15] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The Jalapeño Virtual Machine. *IBM Syst. J.* 39, 1 (jan 2000), 211–238. https://doi.org/10.1147/sj.391.0211

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

[17] David F. Bacon, Perry Cheng, David P. Grove, and Martin T. Vechev. 2007. System and method for concurrent garbage collection. https://www.freepatentsonline.com/y2007/0022149.html

[18] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[19] Lars Ræder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. 2000. Java Bytecode Compression for Low-End Embedded Systems. *ACM Trans. Program. Lang. Syst.* 22, 3 (may 2000), 471–489. https://doi.org/10.1145/353926.353933

[20] Eric Coffin, Scott Young, Harpreet Kaur, Julie Brown, Marius Pirvu, and Kenneth B. Kent. 2020. MicroJIT: A Case for Templated Just-in-Time Compilation in Constrained Environments. In *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '20)*. IBM Corp., USA, 179–188.

[21] Eric Coffin, Scott Young, Kenneth B. Kent, and Marius Pirvu. 2019. A Roadmap for Extending MicroJIT: A Lightweight Just-in-Time Compiler for Decreasing Startup Time. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '19)*. IBM Corp., USA, 293–298.

[22] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient implementation of the smalltalk-80 system. In *POPL '84*.

[23] Anthony Heddings. 2020. What Is Just-In-Time (JIT) Compilation? https://www.howtogeek.com/devops/what-is-just-in-time-jit-compilation/

[24] Harpreet Kaur, Marius Pirvu, and Kenneth B. Kent. 2023. Performance Evaluation of Template-based JIT Compilation in OpenJ9. In *To appear in: CASCON, the 33rd Annual International Conference* (Las Vegas, USA) *(CASCON '23)*. IBM Corp., USA, 10 pages.

[25] Kenneth Kent and Micaela Serra. 2000. Hardware/Software Co-Design of a Java Virtual Machine. *Proceedings of the International Workshop on Rapid System Prototyping* (01 2000), 66–71. https://doi.org/10.1109/IWRSP.2000.855196

[26] Marius Pirvu. 2018. Optimize JVM Startup with Eclipse OpenJ9. https://developer.ibm.com/articles/optimize-jvm-startup-with-eclipse-openjj9/developer.ibm.com/articles/optimize-jvm-startup-with-eclipse-openjj9

[27] Ricardo Nabinger Sanchez, Jose Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. 2011. Using Machines to Learn Method-Specific Compilation Strategies. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 257–266.

[28] Jim Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[29] Federico Sogaro, Eric Aubanel, Kenneth B. Kent, Vijay Sundaresan, Marius Pirvu, and Peter Shipton. 2017. MicroJIT: A Lightweight, Just-in-Time Compiler to Improve Startup Times. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering* (Markham, Ontario, Canada) *(CASCON '17)*. IBM Corp., USA, 140–150.

[30] Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience* 44 (2014).

[31] Christian Wimmer. 2021. GraalVM Native Image: Large-Scale Static Analysis for Java (Keynote). In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Chicago, IL, USA) *(VMIL 2021)*. Association for Computing Machinery, New York, NY, USA, 3. https://doi.org/10.1145/3486606.3488075

[32] Thomas Wuerthinger, Christian Wimmer, and Hanspeter Mossenbpock. 2007. Array bounds check elimination for the Java HotSpot™ client compiler. *ACM International Conference Proceeding Series* 272, 125–133. https://doi.org/10.1145/1294325.1294343

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