
# Vectorized Intrinsics Can Be Replaced with Pure Java Code without Impairing Steady-State Performance

Júnior Löff
*Faculty of Informatics*
*Università della Svizzera italiana (USI)*
Lugano, Switzerland
loeffj@usi.ch

Filippo Schiavio
*Faculty of Informatics*
*Università della Svizzera italiana (USI)*
Lugano, Switzerland
filippo.schiavio@usi.ch

Andrea Rosà
*Faculty of Informatics*
*Università della Svizzera italiana (USI)*
Lugano, Switzerland
andrea.rosa@usi.ch

Matteo Basso
*Faculty of Informatics*
*Università della Svizzera italiana (USI)*
Lugano, Switzerland
matteo.basso@usi.ch

Walter Binder
*Faculty of Informatics*
*Università della Svizzera italiana (USI)*
Lugano, Switzerland
walter.binder@usi.ch

## ABSTRACT

Several methods of the Java Class Library (JCL) rely on *vectorized intrinsics*. While these intrinsics undoubtedly lead to better performance, implementing them is extremely challenging, tedious, error-prone, and significantly increases the effort in understanding and maintaining the code. Moreover, their implementation is platform-dependent. An unexplored, easier-to-implement alternative is to replace vectorized intrinsics with portable Java code using the Java Vector API. However, this is attractive only if the Java code achieves similar steady-state performance as the intrinsics.

This paper shows that this is the case. We focus on the `hashCode` and `equals` computations for byte arrays. We replace the platform-dependent vectorized intrinsics with pure-Java code employing the Java Vector API, resulting in similar steady-state performance. We show that our Java implementations are easy to fine-tune by exploiting characteristics of the input (i.e., the array length), while such tuning would be much more difficult and cumbersome in a vectorized intrinsic. Additionally, we propose a new vectorized `hashCode` computation for long arrays, for which a corresponding intrinsic is currently missing. We evaluate the performance of the tuned implementations on four popular benchmark suites, showing that the performance are in line with those of the original OpenJDK 21 with intrinsics.

Finally, we describe a general approach to integrate code using the Java Vector API into the core classes of the JCL, which is challenging because premature use of the Java Vector API would crash the JVM during its fragile initialization phase. Our approach can be adopted by developers to modify JCL classes without any changes to the native codebase.

## CCS CONCEPTS

• **General and reference** → **Empirical studies**; **Evaluation**; **Performance**; • **Software and its engineering** → *Just-in-time compilers*; *Dynamic compilers*; *Runtime environments*; • **Computing methodologies** → **Parallel programming languages**; • **Computer systems organization** → **Single instruction, multiple data**.

## KEYWORDS

Compiler Intrinsics, SIMD, Vector Instructions, Java Vector API, Portability, Java Virtual Machine.

**ACM Reference Format:**
Júnior Löff, Filippo Schiavio, Andrea Rosà, Matteo Basso, and Walter Binder. 2024. Vectorized Intrinsics Can Be Replaced with Pure Java Code without Impairing Steady-State Performance. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3645051



*ICPE '24, May 7–11, 2024, London, United Kingdom*

ACM ISBN 979-8-4007-0444-4/24/05
https://doi.org/10.1145/3629526.3645051

## 1 INTRODUCTION

In the context of the Java Virtual Machine (JVM), *intrinsics* [2, 21] are a complex runtime machinery introduced for implementing efficient low-level operations.[1] Intrinsics are often implemented in the form of template-generated, assembly-like code or as dedicated nodes in the intermediate representation (IR) used by the just-in-time (JIT) compiler. Several methods of the Java Class Library (JCL) are implemented as intrinsics to improve performance [19]: instead of relying on the JIT compiler to optimize a pure-Java implementation, the JIT compiler directly emits machine code as specified by the intrinsic. In this way, the compilation cost is reduced since the compiler does not need to perform optimization passes. Moreover, by leveraging intrinsics, JVM developers can express low-level computations which are not directly expressible in the Java language. With the increasing support of *vector instructions* [1] (i.e., special machine instructions leveraging SIMD registers to apply an operation to multiple data elements in parallel) in common processors, intrinsics can leverage such instructions to improve performance even on ordinary hardware. We refer to such intrinsics as *vectorized intrinsics*.

[1] Intrinsics are used for multiple reasons, such as for making low-level resources accessible or for improving performance. This paper focuses solely on the latter type of intrinsics, in particular on those making use of vector instructions.

While intrinsics undoubtedly lead to better performance, implementing them is extremely challenging [8, 50]. In addition to the very specific knowledge needed to write highly optimized, template-generated assembly code for a target architecture, implementing an intrinsic is hard, tedious, error-prone, requires a major development effort, and makes debugging, tuning, and testing very time-consuming. Moreover, as intrinsics are platform-dependent, the above effort should be repeated for every architecture that one wishes to support. This issue is aggravated in vectorized intrinsics, since even processors of the same architecture may support different vector extensions. As an example, x86 processors may support Streaming SIMD Extensions [24] (SSE), which have been released in four different versions (SSE1, SSE2, SSE3, SSE4) and Advanced Vector Extensions [25] (AVX), released in three different versions (AVX, AVX2, AVX512). To fully leverage vector instructions, developers need to implement different versions of the vectorized intrinsics, each making use of the vector instructions supported by the underlying processor. Due to this complexity and effort required, in practice, developers implement vectorized intrinsics only for very common architectures (e.g., x86) and only for selected methods of the JCL (e.g., commonly and frequently used primitives on byte arrays such as `hashCode` and `equals`), where the complexity and cost of implementing an intrinsic is justified by a significant performance gain.

These issues—huge effort in development, debugging, and testing, limited portability, substantial platform dependence of vector instructions—can be mitigated by replacing vectorized intrinsics with equivalent platform-independent Java code making use of the *Java Vector API* (JVA) [35], which allows developers to express explicit vector operations from Java code using an object-oriented API, without resorting to any native code. Using the API, appropriate vector instructions are emitted by the JIT compiler depending on the vector instructions supported by the architecture; hence, the same Java code can be executed on multiple platforms, possibly supporting different vector extensions.

The goal of this paper is to show that in addition to reducing the effort in understanding, extending, and maintaining the code, replacing vectorized intrinsics with equivalent Java code using the JVA results in similar steady-state performance, making the approach attractive even in production-level JVMs. We present a brief background on intrinsics in Section 2. As practical use cases, we apply this approach to the `hashCode` (Section 3) and `equals` (Section 5) computations for byte arrays. Moreover, as the code is now easier to extend, debug, and test, we show that it is also easy to fine-tune the code, exploiting characteristics of the input (i.e., the array length); fine-tuning would be very difficult to implement in an intrinsic. Furthermore, we propose a new vectorized `hashCode` computation for long arrays, for which a corresponding intrinsic is currently missing (Section 4). We evaluate the performance of the tuned implementations of `hashCode` and `equals` for byte arrays (but not the new vectorized methods) using four popular benchmark suites, showing that they provide similar performance than the vectorized intrinsics (Section 6). Our implementation is portable and can be run on any JVM and any architecture supported by the JVA, whereas the vectorized intrinsics in OpenJDK 21 [14] (the latest version of one of the most used JVMs worldwide) only work on selected architectures.

The JVA cannot be used during the fragile JVM initialization phase, because it would lead to the premature initialization of classes when the JVM is not yet ready to execute arbitrary Java code. This is a major obstacle, because many methods of the JCL (including `hashCode` and `equals`, on which we focus) are already exercised in the early phases of JVM initialization. This paper proposes a general approach to solve this issue. We describe our approach to modify the Java source code of the JCL, such that our Java implementation making use of the JVA can substitute the use of intrinsics in core classes of the JCL (Section 7). Our approach does not require any change in the native codebase.

To summarize, we show a new, practical way to implement primitives in the JCL using the JVA, offering an attractive, more portable, and easier-to-maintain alternative to the use of vectorized intrinsics. Our approach does not require any knowledge of the OpenJDK native codebase and can be reused by researchers and practitioners to modify JCL classes. In particular, our work makes the following contributions:

- We propose and evaluate fine-tuned JVA implementations of `hashCode` and `equals` for byte arrays (Sections 3 and 5, respectively).
- We propose and evaluate a new vectorized `hashCode` JVA implementation for long arrays (Section 4).
- We assess the performance of our new JVA implementations of `hashCode` and `equals` for byte arrays on four popular benchmark suites for Java (Section 6).
- We propose an approach to modify the Java source code of the core classes of the JCL (Section 7).

We complement the paper with an overview of related work (Section 8), a discussion on the limitations of our approach (Section 9), and our concluding remarks (Section 10).

## 2 BACKGROUND

Intrinsic functions (henceforth also called *intrinsics* for short and also known as *built-in functions*) are functions whose implementations are specially handled by the managed language runtime system. In particular, the managed language runtime system provides 1) a default implementation of the intrinsic function, written in the managed language, and 2) operating system- and architecture-specific semantically equivalent optimized implementations of the intrinsic function. If an optimized implementation is available for the underlying operating system and architecture, at runtime, the interpreter and/or the compiler replaces the default implementation with that optimized implementation. Optimized implementations often leverage operating system and hardware features that are not available in the managed language constructs and APIs, such as vector operations.

Since implementing intrinsics is challenging [8, 50], JVMs implement intrinsics only for some particular architectures and only for some frequently used methods provided by the JCL. Intrinsic implementations may vary not only depending on the operating system and the underlying architecture but also based on the JVM version, JVM vendor, and JIT compiler the JVM uses. For this reason, the platform-dependency of intrinsics aggravates the unpredictable performance of Java code on different architectures.

In this paper, we conduct our experiments on OpenJDK, a widely used open-source JVM implementation. OpenJDK specifies two types of intrinsics [21]: 1) *library intrinsics* that "may be replaced with hand-crafted assembly code, with hand-crafted compiler IR, or with a combination of the two" [22], and 2) *bytecode intrinsics* that are "not replaced by special code, but they are treated in some other special way by the compiler" [20]. In our work, we focus on intrinsics replaced by the JIT compiler and implemented as hand-crafted assembly code that leverages vector hardware instructions. We note that implementing this kind of instrinsics in OpenJDK not only requires knowledge of the low-level assembly programming model, but also knowledge of the metaprogramming techniques used to implement their code generation.

## 3 VECTORIZED HASHCODE FOR BYTE ARRAYS

In this section, we present the vectorized implementation and evaluation of the `hashCode` method. In Java, every object and array supports method `hashCode`, which returns an integer associated with the object or array. In this section, we focus on the `hashCode` computation for byte arrays, which is used also to compute the `hashCode` for Java Strings (as they are internally implemented as byte arrays). An efficient `hashCode` calculation for Java String is very important in multiple scenarios, such as pattern-matching algorithms [30, 31, 46, 47], data-processing systems [10, 23, 49], and data compression [18]. `Arrays.hashCode` is implemented as a variant of the *polynomial rolling hash* function [28], computed as reported in Equation 1.

$$hash(s) = \begin{cases} 0 & \text{if } N = 0 \\ \sum_{i=0}^{N-1} s[i] \cdot p^{N-i-1} \mod m & \text{otherwise} \end{cases} \quad (1)$$

where $s$ is an array of length $N$, $s[i]$ is the $i$-th element of $s$, while $p$ and $m$ are positive integers. In OpenJDK 21, $p = 31$, $m = 2^{32}$, and the computation is performed by a vectorized intrinsic requiring AVX2 vector instructions (which use 256-bit vector registers). We note that the mod $2^{32}$ operation is obtained implicitly due to overflows. The intrinsic has recently been introduced in OpenJDK 21 [13]. We highlight that our experiments show that the previous implementation of `hashCode` (i.e., as simple scalar loop) is not automatically optimized with the superword auto-vectorization optimization performed by the JIT compiler: as shown in Figure 1, the intrinsic largely outperforms the pure-Java implementation.

### 3.1 HashCode Implementation for Byte Arrays

The implementation of the vectorized intrinsic for `Arrays.hashCode` in OpenJDK 21 can be found in the link [17]. Due to its complexity, for the sake of clarity, we introduce the *recast* version of the intrinsic (i.e., an equivalent version using pure Java code relying on the JVA) and discuss this version. We note that all our versions presented in this paper have been thoroughly tested with generated test cases. The pseudo-code of the recast is shown in Figure 2. Variables with a bar on top denote vectors. Since the intrinsic relies on AVX2 instructions, the implementation (using four accumulators) efficiently processes 32 bytes in each iteration. The AVX2 instructions can only operate on 8 input bytes at a time, since each byte is converted to a 32-bit integer (zero extension), resulting in a fourfold size increase of the data.

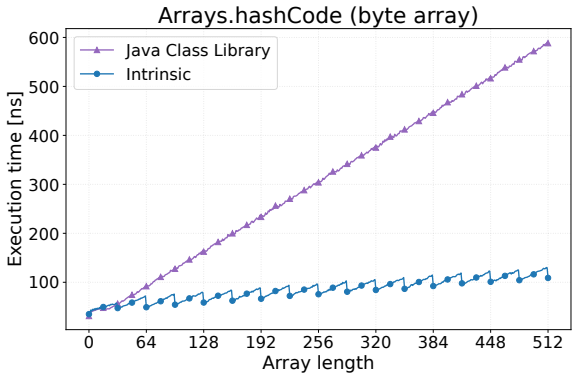


**Figure 1: Performance comparison between the OpenJDK 21 `hashCode` intrinsic and the byte-to-byte loop in pure-Java code implemented in the JCL up to OpenJDK 20.**

The implementation initializes the accumulators (each containing 8 integers) with zeroes (lines 5–8). Then, it loads the coefficient representing the constant $31^{32}$ (line 9). This constant depends on the number of bytes processed per iteration (32 in our example) and is loaded from the `POW31` array. Both our implementation and the intrinsic use pre-computed powers of 31. At position $i$, the `POW31` array stores the value $31^i$.

The main computation unfolds through the sequence of loading data (lines 12–15) and accumulating the computed intermediate results (lines 16–19). The accumulators are updated with the product of the prior accumulation and the coefficient, adding then the latest loaded data. Subsequently, the index advances by 32, reinstating the computation if at least 32 bytes remain. After the loop, each vector is multiplied with a reversed list storing the powers of 31 (lines 21–24). Ultimately, an aggregation is applied to all accumulators through an addition (line 25), yielding the hash value.

Subsequently to the execution of the unrolled vectorized loop, the code processes eventual residual bytes (up to 31). If such bytes exist (i.e., the array length is not a multiple of 32), they are aggregated using a scalar loop that processes 2 bytes in each iteration. This can be seen in lines 29–31. As in the case of the vectorized unrolled loop, the intrinsic uses this strategy to reduce the dependencies between iterations. After the loop, a single remaining byte may still need to be processed. If so, it is subsequently added to the hash value (line 33).

During the implementation of the recast version, we observed two improvements that could be applied to use more vector instructions. We describe the improvements in the following text and implement them in a *tuned* version of `hashCode`.

The first improvement regards the computation of hashcode for arrays of $< 32$ bytes. In this case, the intrinsic processes all the input data with scalar operations (processing 2 bytes in each loop iteration). In our tuned version, we use scalar computation only for arrays of $< 8$ bytes, while for lengths from 8 to 31, we use vectorized instructions relying on a single accumulator.

```
int hashCode_recast(byte[] s) {
 int len = s.length;
 int h = 0;
 if (len >= 32) {
  acc0 = zero(8_INT);
  acc1 = zero(8_INT);
  acc2 = zero(8_INT);
  acc3 = zero(8_INT);
  m = load(8_INT, POW31[32]);
  int bound = len & ~31; // (length/32)*32
  for (int i = 0; i < bound; i += 32) {
   data0 = fromArray(8_BYTE, s, i);
   data1 = fromArray(8_BYTE, s, i+8);
   data2 = fromArray(8_BYTE, s, i+16);
   data3 = fromArray(8_BYTE, s, i+24);
   acc0 = acc0.mul(m).add(data0.convert(...));
   acc1 = acc1.mul(m).add(data1.convert(...));
   acc2 = acc2.mul(m).add(data2.convert(...));
   acc3 = acc3.mul(m).add(data3.convert(...));
  }
  acc0 = acc0.mul(POW31[31..24]);
  acc1 = acc1.mul(POW31[23..16]);
  acc2 = acc2.mul(POW31[15..8]);
  acc3 = acc3.mul(POW31[7..0]);
  h = acc0.add(acc1).add(acc2)
             .add(acc3).reduce(ADD);
 }
 int i = 1 + (len & ~31);
 for (; i < len ; i += 2) {
  h = 31 * 31 * h + 31 * (s[i-1] & 0xff) +
      (s[i] & 0xff);
 }
 if (i == len) {
  h = 31 * h + (s[i-1] & 0xff);
 }
 return h;
}
```

**Figure 2: Pseudo-code of the recast version of `hashCode` for byte arrays using the JVA.**

The second improvement applies for lengths > 32 that are not multiples of 32. The intrinsic processes the residual bytes (i.e., length mod 32) in a scalar loop. Instead, we use up to four additional vector instructions to process 8 bytes at a time. At the end, our tuned implementation loads the last 8 bytes of the array and applies a vector mask, setting to zero the vector positions that contain previously processed bytes to avoid processing them twice. The remaining (not masked) bytes are finally processed in the same way as in the loop using only vector instructions.

## 3.2 Evaluation Methodology and Experimental Setup

We designed a micro-benchmark to compare the execution time of the different `hashCode` implementations for different array lengths. Our goal is to understand whether our pure-Java implementations (recast and tuned) achieve similar steady-state performance as the intrinsic. In this section, we describe our evaluation methodology. We will use the same methodology in Sections 4.2 and 5.2. All the experiments in this paper are executed on a machine equipped with an Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz, featuring 16 physical cores supporting the AVX512 instruction set (supporting 512-bit vector registers). Hyper-threading and turbo boost are disabled. The machine has 256GB of RAM @ 3200MHz. The kernel is Linux 5.15.0-25-generic, and the OS is Ubuntu 22.04 LTS. We use OpenJDK build 21.0.1+12-29.

Our figures report the execution time for each array length from 0 to 512. In each run of the micro-benchmark (within one JVM process), we perform 40k series of measurements, each series with 513 measurements (on byte arrays of lengths 0–512). In each series, we execute the 513 measurements in a randomized order, to ensure that our dynamically compiled Java implementations do not gain any unfair advantage due to predictable execution paths. Among the 40k series of measurements, we consider the first 20k as warm-up, ignoring them. We run the micro-benchmark 5 times in different JVM processes. Overall, each data point (i.e., execution time) in the shown figures is the arithmetic mean of 100k measurements (the 20k steady-state measurements for each of the 5 runs). We note that there is no object allocation during these measurements, and we use the Epsilon no-op garbage collector [4] to prevent any interference of the measurements by the garbage collector. In all figures where the x-axis represents different array lengths, the y-axis shows the execution time in nanoseconds.

## 3.3 Evaluation of HashCode for Byte Arrays

Figure 3 presents a comparison of the steady-state performance of different `hashCode` implementations: the `hashCode` *intrinsic* implemented in OpenJDK 21 and the *recast* and *tuned* versions using the JVA. As the intrinsic only uses AVX2 instructions, the recast and tuned versions only use 256-bit vector instructions as well. The figures also show a tuned version that exploits AVX512 (i.e., 512-bit vector instructions), which was easily written with the JVA. For a fair comparison with the intrinsic, our focus remains exclusively on the 256-bit implementation. We show the AVX512 version only to highlight the potential for further performance enhancement on processors that support 512-bit vector instructions.

As shown in the figure, the curves of each version exhibit different characteristics. The performance trend of the intrinsic and the recast is characterized by its main vectorized loop using four accumulators and the residual computation employing a scalar loop. This can be seen in the figure, where the shortest execution times are measured for array lengths that are multiples of 32, for which the `hashCode` computation involves only vector instructions. Then, the subsequent 31 data points in the curve follow a pattern of increasing overhead due to the scalar loop.

The performance curve of our tuned implementation is also shaped by its main vectorized loop featuring four accumulators (for lengths ≥ 32). However, a notable difference lies in the final phase of the `hashCode` computation. Here, we substitute the scalar `for` loop iterations with vector instructions that process up to 8 bytes at a time. Since often there are less than 8 residual bytes, we apply a mask to disable unused lanes of the vector instruction. This approach leads to a discernible pattern marked by four steps within each 32-byte range. Each step in the range corresponds to the final vector instructions processing 8 bytes using a mask, accompanied by extra vector instructions (at most three) every 8 bytes. While

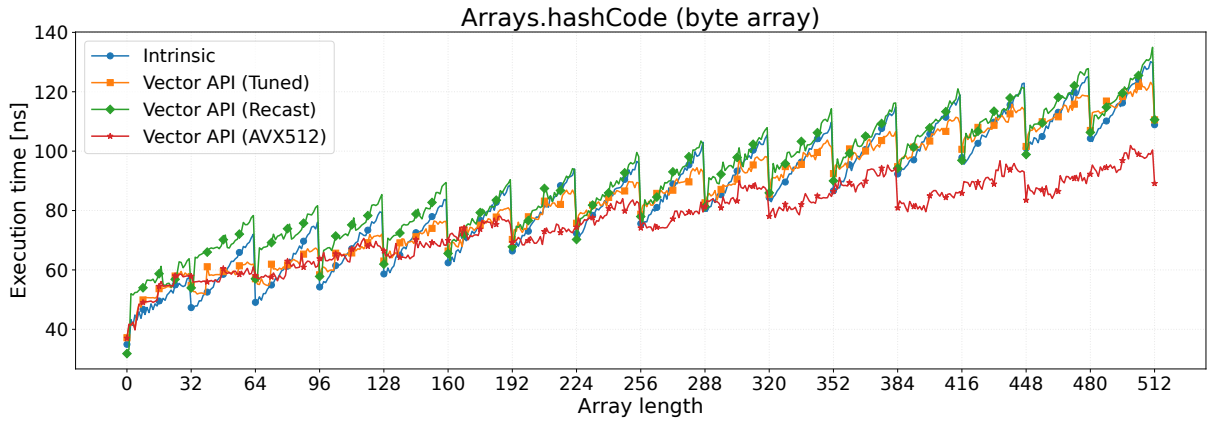


**Figure 3: Performance comparison between the OpenJDK 21 `hashCode` intrinsic and our pure-Java recast and tuned implementations.**

our optimization performs slightly worse than the intrinsic when there are only few residual bytes, it shows speedups when more residual bytes need to be processed.

In our experiments, we observe a different trend between the intrinsic and the recast version up to an array length of 160; beyond this threshold, the two implementations show comparable performance. This is because for longer arrays, the execution time is dominated by the loop using four accumulators, which is implemented in the same way in all versions. Compared with the intrinsic, our experimental results on this micro-benchmark show an overall speedup (geometric mean of the speedup factors for all measured lengths) of 0.94× for the recast version and of 1.10× for the tuned version.

## 4 VECTORIZED HASHCODE FOR LONG ARRAYS

In this section, we present and evaluate our implementation of the vectorized `hashCode` computation for long arrays. In OpenJDK 21, this computation has not (yet) been intrinsified. We describe a pure-Java vectorized implementation to show that using the JVA we can easily vectorize additional methods in the JCL that do not benefit from auto-vectorization.

Figure 4 illustrates the `hashCode` computation for long arrays. The computation makes use of the same polynomial rolling hash function we introduced in the previous section (Equation 1), with the difference that a pre-processing step is performed, consisting of an `XOR` operation on the most- and least significant part of each `long` element, yielding an integer (line 4).

### 4.1 HashCode Implementation for Long Arrays

We describe a new pure-Java implementation that exploits 256-bit vectors to efficiently vectorize the `hashCode` computation for long arrays. Figure 5 shows the core part of the proposed implementation.[2] As mentioned before, the interesting aspect is reading the data in a way to fully exploit the potential of vector instructions.

[2]For more details on vector operations in our pseudo-code, such as `withLane`, refer to the documentation of class `Vector` and its subclasses [26].

```
1  public static int hashCode(long[] ls) {
2    int h = 1;
3    for (long l : ls) {
4      int hash = (int)(l ^ (l >>> 32));
5      h = 31 * h + hash;
6    }
7    return h;
8  }
```

**Figure 4: Pseudo-code of `hashCode` for long arrays in OpenJDK 21.**

The main loop is implemented in lines 6–17. In lines 7–10, we load 4 array elements into a long vector and reinterpret it as an integer vectors of 8 elements. We perform this operation 4 times, reading 16 long elements (reinterpreted as 32 integers) in total. In lines 11–14, we 1) perform the `XOR` operation between each pair of integers originated from the same long via vector shifts, and 2) merge the results from the 4 *part* vectors into 2 *data* vectors, interleaving the results in even and odd positions. Finally, in lines 15–16, we perform the `hashCode` computation.

### 4.2 Evaluation of HashCode for Long Arrays

Figure 6 compares the steady-state performance of the default `hashCode` implementation in OpenJDK 21 with ours. The methodology and experimental setup are the same ones described in Section 3.2. As can be seen from the figure, our vectorized implementation significantly improves performance w.r.t. the default implementation. The performance improvement is more evident as the array length increases. Overall, experimental results on this micro-benchmark show that our JVA implementation outperforms the JCL implementation by a speedup factor (geometric mean of the speedup factors for all measured lengths) of 1.92×.

## 5 VECTORIZED EQUALS FOR BYTE ARRAYS

In this section, we present a vectorized implementation and evaluation of the `equals` method, which determines the equality of two arrays. It is extensively employed in applications and libraries, for

```
1  int hashCode_long(long[] ls) {
2   int len = ls.length;
3   ... // omitted code for len < 16
4   acc0 = zero(8_INT);
5   acc1 = zero(8_INT).withLane(7, 1);
6   for (int i = 0; i <= len-16; i += 16) {
7      part0 = fromArray(4_LONG,ls,i).asInt();
8      part1 = fromArray(4_LONG,ls,i+4).asInt();
9      part2 = fromArray(4_LONG,ls,i+8).asInt();
10     part3 = fromArray(4_LONG,ls,i+12).asInt();
11     data0 = part0.shiftLeft().XOR(part0)
12     .blend(part1.shiftRight().XOR(part1), mask);
13     data1 = part2.shiftLeft().XOR(part2)
14     .blend(part3.shiftRight().XOR(part3), mask);
15     acc0 = acc0.mul(POW31[16]).add(data0);
16     acc1 = acc1.mul(POW31[16]).add(data1);
17  }
18  ... // omitted code for residual (len-i) longs
19 }
```

**Figure 5: Pseudo-code of the vectorized implementation of `hashCode` for long arrays using the JVA.**

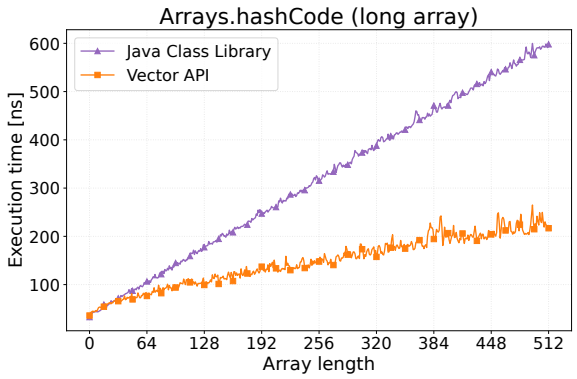


**Figure 6: Performance comparison between the OpenJDK 21 `hashCode` implementation and our version using the JVA.**

example, string equality is internally implemented as arrays equality. This operation is implemented as a simple loop comparing one byte in each iteration. The loop could in principle be automatically vectorized with the superword auto-vectorization of the JIT compiler. However, we highlight that our experiments show that such an optimization is not applied on `equals`: as shown in Figure 7, the intrinsic largely outperforms the pure-Java implementation. In this section, we show that an implementation based on the JVA shows performance in line with the intrinsic.

## 5.1 Equals Implementation for Byte Arrays

We focus on the `equals` implementation for byte arrays, where equality is determined by performing byte-to-byte comparisons. The `equals` implementation starts testing trivial cases, such as the arrays' references and their lengths. Subsequently, the implementation resorts to a byte-to-byte comparison to check whether the elements are equal, which is implemented as an intrinsic [16].

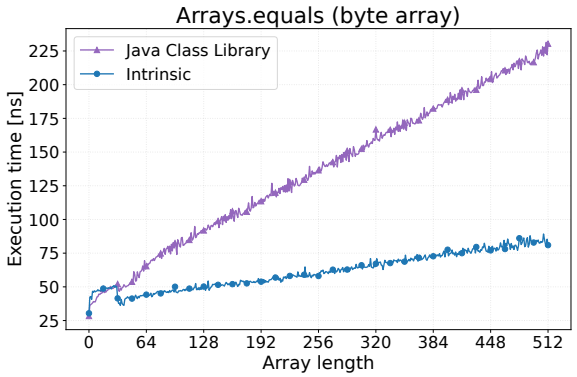


**Figure 7: Performance comparison between the OpenJDK 21 `equals` intrinsic and the byte-to-byte loop in pure-Java code implemented in the Java Class Library.**

```
1  int equals_recast(byte[] s1, byte[] s2) {
2   ... // check if s1 and s2 have same length
3   int len = s1.length;
4   if (len >= 64) {
5      int pos = 0;
6      do {
7        if (!(fromArray(64_BYTE, s1, pos)
8         .cmpEQ(fromArray(64_BYTE, s2, pos))
9         .allTrue()) return false;
10       pos += 64;
11     } while (pos < len-64);
12     return fromArray(64_BYTE, s1, len-64)
13       .cmpEQ(fromArray(64_BYTE, s2, len-64))
14       .allTrue();
15  } else if (len >= 32) {
16     return fromArray(32_BYTE, s1, 0)
17       .cmpEQ(fromArray(32_BYTE, s2, 0))
18       .allTrue() &&
19       fromArray(32_BYTE, s1, len-32)
20       .cmpEQ(fromArray(32_BYTE, s2, len-32))
21       .allTrue();
22  } else {
23     for (int j = 0; j < len; j++) {
24       if (s1[j] != s2[j]) {
25         return false;
26       }
27     }
28     return true;
29  }
30 }
```

**Figure 8: Pseudo-code of the recast version of `equals` for byte arrays using the JVA.**

Based on the original implementation of the intrinsic, we implement a recast version using the JVA (see Figure 8). As the intrinsic employs 512-bit vector instructions if available, our recast version also uses them. For arrays with lengths from 64, the implementation iterates over the arrays, comparing them in batches of 64 bytes at a time (lines 6–11). Subsequently, a final vector instruction is utilized to compare all residual bytes (lines 12–14). This vectorized comparison loads the final vector starting at position `len-64` (line 12), possibly comparing some array elements for a second time.

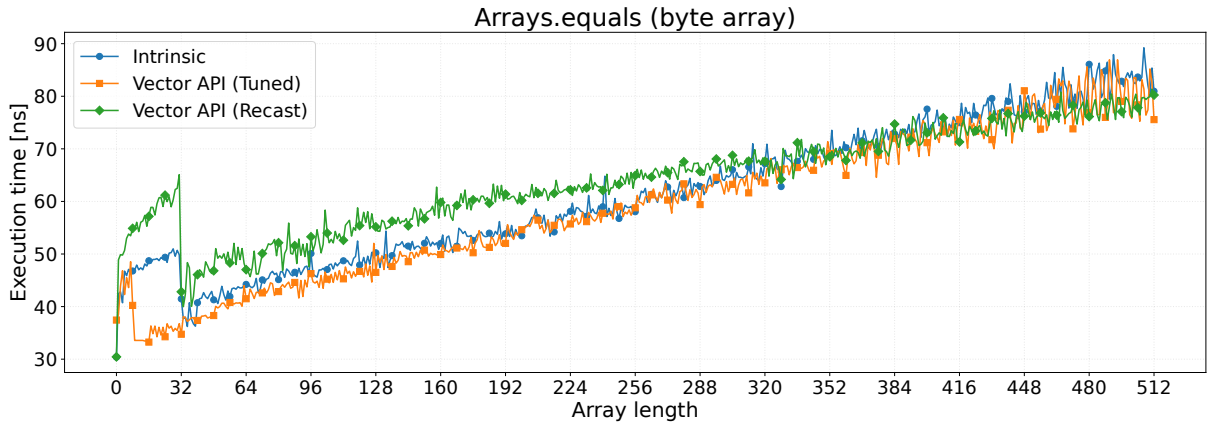


**Figure 9: Performance comparison between the OpenJDK 21 `equals` intrinsic and our pure-Java recast and tuned implementations.**

For arrays with lengths from 32 to 63, the implementation compares the arrays with two 256-bit vector comparisons, i.e., from position 0 to 31, and from position `len-32` to `len-1` (lines 16–21). Arrays with length < 32 are compared using scalar instructions.

We now suggest a few code improvements that could lead to better performance by exploiting vector instructions also on arrays shorter than 32 bytes. The core loop described in Figure 8, applicable for lengths ≥ 64 bytes (lines 6–11), remains unchanged. The code for lengths from 32 to 63 also remains unchanged, i.e., the two vector comparisons (32 bytes each). However, we use the scalar loop for byte-to-byte comparison only for arrays with less than 8 bytes. For arrays with lengths from 8 to 16, our implementation executes two 64-bit vector comparisons. Finally, for arrays with lengths from 17 to 31, our tuned implementation executes two 128-bit vector comparisons.

## 5.2 Evaluation of Equals for Byte Arrays

For evaluating `equals`, we use again the same methodology described in Section 3.2. However, since we have to compare two arrays, we initially duplicate all the byte arrays. We always compare two identical arrays, meaning that `equals` has to process all the array content to find that the arrays are equal.

In Figure 9, we compare the overall steady-state performance of our implementations (recast and tuned) with the intrinsic. As can be seen in the graph, the range of lengths where the performance difference is most visible is between 0 and 32. In this range, both the intrinsic and the recast versions use scalar computations to process short strings. The intrinsic compares multiple bytes reading integers and shorts out of the byte array, while the recast performs a byte-wise comparison, explaining their performance difference in this range. The tuned version uses scalar computation only for arrays with less than 8 bytes, and only vector instructions otherwise. Compared with the intrinsic, our experimental results on this micro-benchmark show an overall speedup (geometric mean of the speedup factors for all measured lengths) of 0.95x for the recast version, and of 1.04x for the tuned version.

# 6 EVALUATION ON BENCHMARK SUITES

In this section, we evaluate steady-state performance of the tuned `hashCode` and `equals` implementations using the JVA that we described in Sections 3 and 5, respectively. To modify the Java source code of the JCL, we employ the approach described later in Section 7. We conduct a performance evaluation on popular realistic benchmark suites for the JVM. We note that our goal is not to significantly outperform the (already optimized) vectorized intrinsics, but to demonstrate that the easy-to-tune pure-Java alternative, in addition to being easier to implement, test, debug, and maintain, can replace the intrinsic implementation without impairing steady-state performance.

Our evaluation considers the Renaissance [36], DaCapo [7], and ScalaBench [42] benchmark suites. For Renaissance and ScalaBench, we use the latest versions of the suites at the time of writing (Renaissance GPL v0.15.0 and ScalaBench v0.1.0). For DaCapo, we use both v9.12-Bach (released in December 2009) and v23.11-Chopin (released in November 2023). We note that the two DaCapo versions are substantially different in their workloads. We exclude benchmarks whose execution on Java 21 is not supported or with known bugs [40]. We use the *default* input size. Benchmarks can execute multiple iterations, which can either be considered as warm-up or steady-state. We run warm-up iterations until dynamic compilation and GC ergonomics are stabilized, as follows. For DaCapo and ScalaBench, we follow the approach described by Lengauer et al. [29], executing 40 warm-up iterations for each benchmark. For Renaissance, we use the same number of warm-up iterations as specified in the documentation [37]. All other iterations after warm-up are classified as steady-state. Our measurements consider only steady-state iterations, and we execute 10 such iterations for each benchmark. Finally, we run every benchmark 10 times in different JVM processes, collecting a total of 100 steady-state iterations (from 10 different JVM processes). For the measurements, we use the same experimental setup described in Section 3.2, except we use the default G1 garbage collector [11].

Table 1 reports the execution time of each benchmark when run 1) with the original vectorized intrinsic as implemented in OpenJDK 21, and 2) with our tuned implementation using the JVA.

**Table 1: Performance comparison on popular benchmark suites: original OpenJDK 21 with `hashCode` and `equals` intrinsics, vs. modified OpenJDK 21 using our tuned JVA implementation.**

| Benchmark | Intrinsic | | Vector API | | |
|---|---|---|---|---|---|
| **Renaissance GPL 0.15.0** | **Time (ms)** | **Std. dev.** | **Time (ms)** | **Std. dev.** | **Rel. diff.** |
| akka-uct | 6604.54 | 140.23 | 6610.62 | 160.85 | -0.09% |
| als | 1415.16 | 11.77 | 1416.98 | 13.85 | -0.13% |
| chi-square | 621.24 | 84.75 | 610.75 | 86.91 | 1.69% |
| dec-tree | 740.63 | 23.93 | 747.20 | 24.21 | -0.89% |
| dotty | 767.41 | 15.16 | 766.05 | 16.61 | 0.18% |
| finagle-chirper | 1793.15 | 25.29 | 1779.18 | 27.60 | 0.78% |
| finagle-http | 2068.98 | 24.63 | 2112.00 | 15.25 | -2.08% |
| fj-kmeans | 1084.49 | 4.16 | 1136.04 | 9.38 | -4.75% |
| future-genetic | 1851.89 | 34.66 | 1866.27 | 30.41 | -0.78% |
| gauss-mix | 683.33 | 51.74 | 679.78 | 53.18 | 0.52% |
| log-regression | 711.14 | 44.19 | 735.34 | 58.01 | -3.40% |
| mnemonics | 2667.95 | 6.47 | 2766.90 | 263.81 | -3.71% |
| movie-lens | 6478.20 | 50.63 | 6492.78 | 38.31 | -0.23% |
| naive-bayes | 336.51 | 29.15 | 340.15 | 32.34 | -1.08% |
| neo4j-analytics | 1526.52 | 16.99 | 1539.31 | 29.80 | -0.84% |
| page-rank | 2843.99 | 56.93 | 2866.12 | 52.79 | -0.78% |
| par-mnemonics | 2048.16 | 25.53 | 2080.03 | 70.24 | -1.56% |
| philosophers | 2793.10 | 99.25 | 2761.93 | 170.18 | 1.12% |
| reactors | 9097.17 | 543.53 | 9117.86 | 497.19 | -0.23% |
| rx-scrabble | 113.89 | 11.09 | 113.73 | 12.20 | 0.14% |
| scala-stm-bench7 | 810.70 | 37.92 | 811.33 | 50.56 | -0.08% |
| scrabble | 87.12 | 9.08 | 86.89 | 8.49 | 0.27% |
| **Geo. mean** | **1246.00** | | **1254.88** | | **-0.71%** |

| Benchmark | Intrinsic | | Vector API | | |
|---|---|---|---|---|---|
| **ScalaBench 0.1.0** | **Time (ms)** | **Std. dev.** | **Time (ms)** | **Std. dev.** | **Rel. diff.** |
| apparat | 5122.33 | 169.34 | 5129.38 | 145.43 | -0.14% |
| factorie | 10716.55 | 217.71 | 10852.47 | 426.89 | -1.27% |
| kiama | 222.29 | 17.26 | 222.62 | 18.93 | -0.15% |
| scalac | 761.20 | 36.63 | 762.31 | 35.45 | -0.15% |
| scaladoc | 1281.05 | 7.72 | 1269.87 | 27.39 | 0.87% |
| scalap | 90.10 | 4.76 | 90.33 | 4.62 | -0.26% |
| scalariform | 323.94 | 18.69 | 327.00 | 19.29 | -0.94% |
| scalaxb | 180.26 | 6.64 | 180.10 | 6.71 | 0.09% |
| tmt | 3217.71 | 45.83 | 3279.01 | 50.93 | -1.91% |
| **Geo. mean** | **836.92** | | **840.47** | | **-0.42%** |

| Benchmark | Intrinsic | | Vector API | | |
|---|---|---|---|---|---|
| **DaCapo 23.11-Chopin** | **Time (ms)** | **Std. dev.** | **Time (ms)** | **Std. dev.** | **Rel. diff.** |
| avrora | 3383.00 | 15.69 | 3369.85 | 17.59 | 0.39% |
| biojava | 8109.00 | 124.73 | 8642.29 | 89.29 | -6.58% |
| eclipse | 12693.16 | 53.74 | 12683.34 | 45.49 | 0.08% |
| fop | 583.33 | 3.00 | 585.46 | 3.41 | -0.37% |
| graphchi | 5034.84 | 70.58 | 4994.41 | 35.61 | 0.80% |
| h2 | 2679.13 | 38.83 | 2686.87 | 42.92 | -0.29% |
| jme | 6920.72 | 2.75 | 6924.44 | 2.80 | -0.05% |
| jython | 4279.78 | 88.23 | 4296.74 | 85.05 | -0.40% |
| kafka | 5046.25 | 28.54 | 5008.61 | 30.20 | 0.75% |
| luindex | 4900.54 | 58.55 | 4899.83 | 57.11 | 0.01% |
| lusearch | 3137.30 | 39.01 | 3149.17 | 41.59 | -0.38% |
| pmd | 1539.85 | 13.91 | 1615.78 | 18.57 | -4.93% |
| spring | 2382.96 | 183.22 | 2463.59 | 91.42 | -3.38% |
| sunflow | 3690.26 | 357.97 | 3691.84 | 328.14 | -0.04% |
| xalan | 589.07 | 5.76 | 601.11 | 6.10 | -2.04% |
| zxing | 1231.35 | 7.89 | 1246.63 | 9.20 | -1.24% |
| **Geo. mean** | **3068.70** | | **3101.97** | | **-1.08%** |

| Benchmark | Intrinsic | | Vector API | | |
|---|---|---|---|---|---|
| **DaCapo 9.12-Bach** | **Time (ms)** | **Std. dev.** | **Time (ms)** | **Std. dev.** | **Rel. diff.** |
| fop | 175.91 | 5.38 | 177.77 | 4.01 | -1.06% |
| h2 | 2275.57 | 116.82 | 2282.13 | 120.95 | -0.29% |
| jython | 1286.18 | 81.78 | 1297.37 | 50.33 | -0.87% |
| luindex | 367.44 | 4.85 | 369.65 | 5.06 | -0.60% |
| lusearch | 109.90 | 7.95 | 110.39 | 7.80 | -0.45% |
| lusearch-fix | 111.75 | 7.12 | 109.86 | 7.90 | 1.69% |
| pmd | 482.10 | 30.15 | 480.73 | 29.96 | 0.28% |
| sunflow | 501.29 | 34.37 | 493.01 | 29.16 | 1.65% |
| xalan | 119.72 | 1.63 | 120.96 | 1.41 | -1.04% |
| **Geo. mean** | **343.87** | | **344.11** | | **-0.07%** |

For fairness, since OpenJDK 21 does not use any intrinsics for `hashCode` on long arrays, we do not use the version described in Section 4 in this evaluation. The execution times reported in the table represent the arithmetic mean of 100 steady-state iterations for each benchmark. We also report the standard deviation, the relative difference of the execution time of the JVA implementation w.r.t. the intrinsic (i.e., the difference between the execution time of the intrinsic and the one of the JVA implementation, divided by the one of the intrinsic) and the overall per-suite geometric-mean execution times and relative difference.

As one can see from the table, our tuned implementation using the JVA does not impair steady-state performance w.r.t. the intrinsic in the evaluated benchmarks. Overall, our version results in very similar steady-state performance to the vectorized intrinsic. Considering the average (geometric mean) of all benchmarks in a suite, our version results in a relative difference of -0.71% (Renaissance), -0.42% (ScalaBench), -1.08% (DaCapo-Chopin) and -0.07% (DaCapo-Bach). Considering all benchmarks, the average execution time (geometric mean) relative difference is 0.67%.

Overall, our implementations provide similar steady-state performance than the vectorized intrinsics, but with the benefit of being written purely in Java, being much more portable, easier to understand, maintain, and fine-tune. In addition, our approach avoids writing platform-specific code to exploit particular vector extensions a processor may support—the JVA automatically uses the vector instructions supported by the underlying architecture.

## 7 MODIFYING CORE CLASSES IN THE JCL

In this section, we detail our approach to modify the Java source code of the JCL, such that our Java implementations making use of

the JVA can substitute the use of intrinsics in core classes of the JCL (particularly in class `Arrays`). Even though the JCL is implemented as part of the JVM, our approach does not require one to modify the native code of the JVM and hence can be exploited by developers without the need to recompile the JVM. We note that our approach is fully compliant with the JVM specification. We describe our approach in the context of OpenJDK 21 and we remark that we evaluated our approach on state-of-the-art benchmark suites in Section 6.

Modifying core classes in the JCL (such as `Arrays`) is challenging [6, 33, 39, 41] since during the early phases of JVM initialization, many Java features (including the JVA, as well as e.g. the Java Reflection API [34]) cannot be used. Modifying JCL methods (such as `Arrays.hashCode` and `Arrays.equals`) to use such features will cause JVM crashes. The reason is that modified JCL classes may alter the order in which classes are initialized, leading to premature class initializations. Moreover, when modifying methods in the JCL classes, it is crucial to avoid cyclic dependencies that would lead to an infinite recursion in class initialization. This behaviour is particularly subtle because a modified method may trigger the initialization of some classes, and the initializers of these classes may use (directly or indirectly) the modified method.

To ensure proper JVM initialization, our approach introduces *initialization guards* and leverages class redefinition. In particular, our modified JCL methods make use of a custom `JVMInitialization` class that allows checking whether JVM initialization has completed via its `isInitialized` static method, which returns a boolean flag. Below, we describe how our approach enables the use the JVA in JCL core classes.

## 7.1 Modified JCL Methods and Initialization Guards

We modify methods in the JCL by inserting our implementation into the body of the original method implementation, guarded by a conditional invoking the `JVMInitialization.isInitialized` method. If this method returns `true`, our implementation (using the JVA) is executed. Otherwise, the original JCL implementation is used. Thanks to our guard (that checks whether JVM initialization has completed) and because the JVM ensures lazy class initialization upon the first use of a class [32], the classes used by our implementation (i.e., the classes of the JVA) will not be initialized during JVM initialization.

## 7.2 Class JVMInitialization

Since Java does not expose an interface to check whether JVM initialization has completed, we implement the static `JVMInitialization .isInitialized` method, which returns a boolean flag that is toggled right after JVM initialization. This method could be trivially implemented by storing the boolean flag in a volatile static field and by setting this field to `true` in the beginning of an application's `main` method. However, this incurs a serious performance issue: the cost of a volatile read upon each invocation of the `isInitialized` method can be significant and jeopardize the optimizations introduced by our implementations.

For this reason, we implement the flag as a static `isInit- ialized` method that initially returns the boolean constant `false`, and we use a Java agent [15] to redefine the `isInit- ialized` method to return `true` upon the execution of the agent's `premain` method (i.e., when the JVM is ready to execute arbitrary Java code). This strategy allows exploiting the branch-elimination optimization of the JIT compiler, improving performance even further. From the JIT-compiler perspective, the static `isInitialized` method returns a constant and hence only one of either our implementation or the original JCL implementation will be executed. The compiler will therefore perform branch elimination and remove the code of the implementation that will never be executed, increasing the code size budget for other optimizations, such as method inlining. In the (unlikely) case that the JIT compiler already optimized the modified method during JVM initialization, before our Java agent redefines the `isInitialized` method, we rely on OpenJDK's deoptimization feature to ensure the correctness of our solution. Moreover, the JIT compiler in OpenJDK will re-compile and optimize hot code after our class redefinition, ensuring that our initialization guards incur no overhead in steady state.

Finally, to avoid cyclic dependencies that would lead to infinite recursions in class initialization, we make the Java agent (which is allowed to execute arbitrary Java code) initialize all the classes used by our implementations before redefining the `isInitialized` method.

# 8 RELATED WORK

In this section, we discuss related work. We first discuss techniques to improve performance of high-level and managed languages (Section 8.1). Then, we detail benchmarks for vector computations in high-level languages (Section 8.2).

## 8.1 Portability without Sacrificing Performance

Substantial research effort has been made with the goal of proposing portable and high-level programming languages, domain-specific languages (DSL), and libraries without sacrificing performance [3, 48]. As an example, in the "abstraction without overhead" line of work, [9, 44, 45] show that by leveraging staged compilation [38], the cost of abstractions can be removed at compilation time, allowing developers to implement high-performance applications in high-level languages. Within this line of work, an alternative vector API for Scala and Java has been proposed by Stojanov et al. [43] to express vector computations with a high-level DSL, showing that an implementation leveraging the proposed DSL for expressing vector operations can outperform pure Java code that relies on auto-vectorization. In contrast, we evaluate our implementations based on the JVA against vectorized intrinsics, which are not the result of automatic program transformation, but fine-tuned implementations carefully written by expert JVM developers.

Generally, the need for expressing low-level operations within high-level programming languages has been discussed and motivated by Frampton et al. [12]. This work proposes techniques to safely integrate low-level components within high-level languages, as done in the JVA. Similarly, we demonstrate that the JVA allows optimizing the JCL without the burden of writing intrinsics.

### 8.2 Benchmarking Vector Computations on Managed Runtimes

Basso et al. propose JVBench [5], a benchmark suite for the JVA, showing that by leveraging the API, developers can write applications that result in higher performance with respect to auto-vectorization. The Swan benchmark suite [27] has been proposed for benchmarking vectorized operations in the context of mobile applications. In contrast to these benchmark suites, we analyze the performance of the JVA against fine-tuned vectorized intrinsics. To the best of our knowledge, our work is the first to demonstrate that the JVA can be used to easily optimize core JCL classes without the burden of writing vectorized intrinsics.

## 9 LIMITATIONS

A limitation of our work is that, while the most popular architectures and JVMs support the JVA, not all of them fully support the API. Nonetheless, this API exhibits continuous performance improvements in subsequent versions and enables higher performance than relying on the JVM's limited auto-vectorization capabilities [5]. Another limitation is that our evaluation was conducted on a single machine and architecture (x86). We plan to experiment more extensively on a wide range of architectures.

By delegating the generation of vector instructions to the JVA, the code (if hot) will be processed by the JIT compiler, which may apply decisions resulting in suboptimal performance. In contrast, in OpenJDK 21 the code specified in the intrinsic is emitted without undergoing further optimizations, giving more control to the developers.

Finally, using the JVA incurs extra JIT-compilation costs, while machine-code generation for an intrinsic may be faster. This may impair the performance of short-running applications, as steady-state performance may be reached later in the application's execution. As part of our ongoing research, we are investigating whether our approach negatively affects startup performance or not.

## 10 CONCLUSIONS

In this paper, we support the claim that replacing platform-dependent template-generated assembly code implemented via vectorized intrinsics with equivalent, portable, pure-Java code using the JVA does not impair steady-state performance, making the approach attractive even in production-level JVMs. We show this by replacing the vectorized intrinsics of `hashCode` and `equals` for byte arrays with equivalent Java code. Moreover, in addition to reducing the effort in understanding, extending, debugging, testing, and maintaining the code, the resulting code is much easier to fine-tune to further improve performance. We propose code improvements for `hashCode` and `equals`, where tuning is based on the array length; such tuning would be difficult to implement in vectorized intrinsics. Furthermore, we proposed a new vectorized `hashCode` computation for long arrays, for which a corresponding intrinsic is missing in OpenJDK 21.

Our evaluation shows that our tuned implementations provide similar performance than the vectorized intrinsics on four popular and realistic benchmark suites. Finally, we present a technique that can be adopted by developers to modify core classes of the JCL without disrupting JVM initialization; our technique does not require any changes to the native codebase of OpenJDK.

The findings in this work highlight the potential of the JVA as a viable alternative for vectorizing other compute-intensive methods in the JCL using only platform-independent Java code, avoiding writing complex platform-dependent native code that is hard to understand, extend, maintain, debug, and test.

As part of our future work, in addition to tackling the limitations discussed in Section 9, we plan to expand our use cases, replacing vectorized intrinsics in other commonly executed methods in the JCL with pure-Java code using the JVA, evaluating the performance improvements. We also plan to identify methods that could benefit from vectorization via the JVA and for which a vectorized intrinsic is currently missing. Finally, we will evaluate more metrics (e.g., energy consumption) and hardware architectures (e.g., ARM).

## ACKNOWLEDGEMENTS

The research presented in this paper was supported by Oracle (ERO project 1332) and the Swiss National Science Foundation (project 200020_188688).

## REFERENCES

[1] 2013. Intel Xeon Phi Coprocessor High Performance Programming. Morgan Kaufmann.
[2] 2015. Software Analysis and Optimization. In *Power and Performance*, Jim Kukunas (Ed.). Morgan Kaufmann.
[3] Bowen Alpern, Steve Augart, Stephen M Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, et al. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–417.
[4] Andrew Binstock. 2019. Epsilon: The JDK's Do-Nothing Garbage Collector. https://blogs.oracle.com/javamagazine/post/epsilon-the-jdks-do-nothing-garbage-collector.
[5] Matteo Basso, Andrea Rosà, Luca Omini, and Walter Binder. 2023. Java Vector API: Benchmarking and Performance Analysis. In *CC*. 1–12.
[6] Walter Binder, Philippe Moret, Éric Tanter, and Danilo Ansaloni. 2016. Polymorphic Bytecode Instrumentation. *Softw. Pract. Exper.* 46, 10 (2016), 1351–1380.
[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*. 169–190.
[8] Brian Goetz. 2019. dont intrinsify Objects::hash. https://mail.openjdk.org/pipermail/amber-dev/2019-April/004264.html.
[9] Kevin J Brown, Arvind K Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 89–100.
[10] Maximilian Böther, Lawrence Benson, Ana Klimovic, and Tilmann Rabl. 2023. Analyzing Vectorized Hash Tables Across CPU Architectures. *Proceedings of the VLDB Endowment* 16 (8 2023), 2755–2768. Issue 11.
[11] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management*. 37–48.
[12] Daniel Frampton, Stephen M Blackburn, Perry Cheng, Robin J Garner, David Grove, J Eliot B Moss, and Sergey I Salishev. 2009. Demystifying Magic: High-Level Low-Level Programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 81–90.
[13] GitHub. 2023. 8282664: Unroll by hand StringUTF16 and StringLatin1 polynomial hash ... · openjdk/jdk21u@e37078f<. https://github.com/openjdk/jdk21u/commit/e37078f5bb626c7ce0348a38bb86ca2ca62ba915.
[14] GitHub. 2023. jdk21. https://github.com/openjdk/jdk21.
[15] GitHub. 2023. Package java.lang.instrument. https://docs.oracle.com/en/java/javase/21/docs/api/java.instrument/java/lang/instrument/package-summary.html.
[16] GitHub. 2023. Source code of arrays_equals in c2_MacroAssembler_x86.cpp. https://github.com/openjdk/jdk21/blob/

890adb6410dab4606a4f26a942aed02fb2f55387/src/hotspot/cpu/x86/c2_MacroAssembler_x86.cpp#L4086.
[17] GitHub. 2023. Source code of arrays_hashcode in c2_MacroAssembler_x86.cpp. https://github.com/openjdk/jdk21/blob/890adb6410dab4606a4f26a942aed02fb2f55387/src/hotspot/cpu/x86/c2_MacroAssembler_x86.cpp#L3285.
[18] GitHub. 2023. Source code of java.util.zip.ZipCoder. https://github.com/openjdk/jdk21u/blob/89aea0dace4afd2b1bbc5d636322969afcf0072c/src/java.base/share/classes/java/util/zip/ZipCoder.java#L294.
[19] GitHub. 2023. Source code of vmIntrinsics.hpp. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/classfile/vmIntrinsics.hpp.
[20] GitHub. 2023. Source code of vmIntrinsics.hpp, comment on bytecode instrinsics. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/classfile/vmIntrinsics.hpp#L78.
[21] GitHub. 2023. Source code of vmIntrinsics.hpp, comment on instrinsics. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/classfile/vmIntrinsics.hpp#L72.
[22] GitHub. 2023. Source code of vmIntrinsics.hpp, comment on library instrinsics. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/classfile/vmIntrinsics.hpp#L74.
[23] Tobias Groth, Sven Groppe, Thilo Pionteck, Franz Valdiek, and Martin Koppehel. 2022. Accelerated Parallel Hybrid GPU/CPU Hash Table Queries with String Keys. In *Database and Expert Systems Applications*. 191–203.
[24] Intel. 2007. Intel SSE4 Programming Reference. https://www.intel.com/content/dam/develop/external/us/en/documents/\d9156103-138479.pdf.
[25] Intel. 2015. Intel Advanced Vector Extensions 512. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.
[26] Java Platform, Standard Edition & Java Development Kit – Version 21 API Specification. 2023. Class Vector. https://docs.oracle.com/en/java/javase/21/docs/api/jdk.incubator.vector/jdk/incubator/vector/Vector.html.
[27] Alireza Khadem, Daichi Fujiki, Nishil Talati, Scott Mahlke, and Reetuparna Das. 2023. Vector-Processing for Mobile Devices: Benchmark and Analysis. In *2023 IEEE International Symposium on Workload Characterization*. 15–27.
[28] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc.
[29] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 3–14.
[30] Chung Yu Liao and Cheng Hung Lin. 2017. A Novel Parallel Dual-character String Matching Algorithm on Graphical Processing Units. In *ICA3PP*. 197–210.
[31] Cheng-Hung Lin, Jin-Cheng Li, Chen-Hsiung Liu, and Shih-Chieh Chang. 2017. Perfect Hashing Based Parallel Algorithms for Multiple String Matching on Graphic Processing Units. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2639–2650.
[32] Tim Lindholm, Frank Tellin, Gilad Bracha, Alex Buckley, and Daniel Smith. 2023. The Java Virtual Machine Specification – Java SE 20 Edition – Chapter 5. Loading, Linking, and Initializing. https://docs.oracle.com/javase/specs/jvms/se20/html/jvms-5.html#jvms-5.5.
[33] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-Specific Language for Bytecode Instrumentation. In *AOSD*. 239–250.
[34] Oracle. 2023. Java Reflection API. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html.
[35] Oracle Corporation. 2023. JEP 448: Vector API (Sixth Incubator). https://openjdk.org/jeps/448.
[36] Prokopec, A. and Rosà, A. and Leopoldseder, D. and Duboscq, G. and Tůma, P. and Studener, M. and Bulej, L. and Zheng, Y. and Villazón, A. and Simon, D. and Würthinger, T. and Binder, W. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *PLDI*. 31–47.
[37] Renaissance Suite. 2019. Renaissance Suite - Documentation. https://renaissance.dev/docs.
[38] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*. 127–136.
[39] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2017. Accurate Reification of Complete Supertype Information for Dynamic Analysis on the JVM. In *GPCE*. 104–116.
[40] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2019. Analysis and Optimization of Task Granularity on the Java Virtual Machine. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 19 (jul 2019), 47 pages.
[41] Andrea Rosà and Walter Binder. 2018. Optimizing Type-specific Instrumentation on the JVM with Reflective Supertype Information. *Journal of Visual Languages & Computing* 49 (2018), 29–45.
[42] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 657–676.
[43] Alen Stojanov, Ivaylo Toskov, Tiark Rompf, and Markus Püschel. 2018. SIMD intrinsics on managed language runtimes. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 2–15.
[44] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 609–616.
[45] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014), 1–25.
[46] Tran Ngoc Thinh, Surin Kittitornkun, and Shigenori Tomiyama. 2007. Applying Cuckoo Hashing for FPGA-based Pattern Matching in NIDS/NIPS. In *FPT*. 121–128.
[47] Kuo-Kun Tseng, Ying-Dar Lin, Tsern-Huei Lee, and Yuan-Cheng Lai. 2005. A Parallel Automaton String Matching with Pre-hashing and Root-indexing Techniques for Content Filtering Coprocessor. In *ASAP*. 113–118.
[48] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (jan 2013), 24 pages.
[49] Tianqi Zheng, Zhibin Zhang, and Xueqi Cheng. 2020. SAHA: A String Adaptive Hash Table for Analytical Databases. *Applied Sciences* 10, 6 (2020), 1–18.
[50] Zoltán Majó. 2016. C1 arraycopy intrinsic type checks missing. https://mail.openjdk.org/pipermail/hotspot-compiler-dev/2016-June/023527.html.