The Quest for the Best Thread-Safe Java List

MARK VAN WIJK, University of Twente, The Netherlands

With most modern processors having more than one core, more resources become available to programs that can use these extra cores. This research shows which thread-safe Java List implementations are the best to use in programs with different ratios of read to write operations, evaluated based on the performance, energy consumption and memory usage. We found that two implementations outperform the others on all fronts, unless there are hundreds of times more read than write operations.

Additional Key Words and Phrases: Java, List, Performance, Energy Consumption, Memory Usage, Concurrency, Thread-Safe.

1 INTRODUCTION

To help developers store grouped data, Java has an extensive suite of collections: the Java Collections Framework (JCF) [25]. The collections in this framework are implementations of several different Abstract Data Types (ADTs), of which the *list*, *set*, *map*, *queue* and *deque* are the most prevalent ones [21]. For all these ADTs, there are both non-thread-safe and thread-safe implementations available.

Each of these implementations has its own unique behaviour. For example, a list implementation might be faster at removing its first item, but slower at reading a value in its middle than another implementation. These differences have serious consequences for the footprint of a program. Just by changing the implementations for the used ADTs in a program, it is possible to get a 17% reduction in energy consumption [17]. While this is great, it does require developers to be aware of the differences between the implementations with regards to the metrics relevant to their use-case.

To unravel the properties of these implementations, there has been research on their performance [2, 40, 42], energy consumption [8, 12, 17, 32, 34, 35] and memory usage [2, 40]. However, these papers do either not focus on thread-safe implementation, or do so by considering each method individually. While isolating each method might seem fine, this is usually not what happens in applications and is also not necessarily what the implementations are optimized for. For instance, the documentation on the CopyOnWriteArrayList contains the sentence "This is ordinarily too costly, but may be more efficient than alternatives when traversal operations vastly outnumber mutations" [26]. This raises a few questions: it 'may' be more efficient, but is it? And what does 'vastly outnumber' mean? It also means that to determine whether the CopyOnWriteArrayList is the best fit in a use-case, requires more than focusing on the performance of the individual actions. This research will therefore focus on ratios between various actions on the implementations, to understand how these combinations influence the behaviour of the implementations.

Since the JCF is quite extensive, it is unfortunately not possible to focus on all ADTs. For this reason, we will focus on the most popular ADT: the list [2]. They account for more than half of all occurrences of Java data structures in GitHub repositories. The implementations will be evaluated based on the metrics that were most prevalent during our literature research:

RQ1. Which thread-safe list implementations from the JCF should be used to maximise performance?
RQ2. Which thread-safe list implementations from the JCF should be used to minimise power consumption?
RQ3. Which thread-safe list implementations from the JCF should be used to minimise memory usage?

To answer these questions, this research compares the behaviour of list implementations under different ratios between read and write operations for a variety of threads.

2 BACKGROUND

2.1 Implementations

There are several ways in which thread-safety can be achieved in Java. The first of which is locking, which can be achieved by using the *synchronized* keyword or an explicit lock. This ensures that only one thread at a time can access synchronized/locked code. More over, all memory gets updated for the thread entering the synchronized/locked code and it publishes its memory to all other threads once it leaves [22]. This approach is used in the *Vector* [31] and for the *Collections.synchronizedList(List)* method. This method can take as a parameter any list and make it thread-safe [21]. When this is done for the *ArrayList* [23] and *LinkedList* [29], we will refer to as *SynchronizedArrayList* and *SynchronizedLinkedList* from now on.

The next approach used to achieve thread-safety in the JCF is the *copy-on-write* pattern [26, 27]. In this case, a thread only acquires a lock when it wants to modify the state, so read operations do not require any kind of locking. This allows multiple threads to read at the same time, which might allow better scalability. Since the readers still need to be aware of any changes to the state, the internal state does need to be *volatile*, since this forces the thread to not see stale values. This brings a performance penalty, since the compiler cannot deviate from the program order for optimisations and it needs to update its cache on each access to the field [7, 24]. Writes to these data structures are typically expensive operations for two reasons. First, they require locks, meaning that there is no interleaving allowed. Second, an update requires a copy of the entire internal state. This internal state is usually an array that has to be copied in its entirety. This consumes time, memory and energy. An advantage is that even if a thread is updating the internal state, other threads can still safely read from it concurrently. An example of this pattern in the JCF is the CopyOnWriteArrayList [26].

The last approach to achieve thread-safety we will cover, is the *compare-and-swap* operation. With this operation, a variable can be atomically updated if the thread is able to provide the current value of that variable. This prevents the need for locking, but in high contention environments, it might take a while before each

TScIT 37, July 8, 2022, Enschede, The Netherlands

operation is executed, since if a value gets updated by a thread, other threads which assume the previous value of the variable is still the most recent value, will fail [5, 16]. To ensure updates are correctly published to all threads, a field that is accessed through the *compare-and-swap* operation, does still need to be *volatile*. This means, just like for the reading of *copy-on-write* data structures, that the compiler cannot impose certain optimizations and needs to refresh its caches on each access. While the *compare-and-swap* patterns can be a convenient pattern in some cases, there are no list implementations within the JCF that use this approach.

2.2 Features to Analyze

Most programs are, if optimized, optimized for performance. In a lot of cases this might be fine, but other factors may be more important.

In a smartphone for example, it is important to keep the energy consumption low, since a battery has limited power. It might also be beneficial for a server to have a lower power consumption, since this might save a lot of money on the power bill or help a company get a step closer to achieving their goals to be carbon neutral. One might be tempted to assume that if the program is faster, it will consume less energy, but this is not necessarily the case [33].

On a Raspberry Pi, it might also be desired to keep the memory usage low, since some versions have limited memory. One could argue that in this case using Java might not be ideal, since its memory usage is already quite high compared to its performance and power usage [33], but there can be situations where the developer has no other option. It might also be the case that the memory is the limiting factor on a server, so reducing the memory usage of the program might improve the overall performance by reducing the need to garbage collections [9].

2.3 Methods to Analyze

Previously conducted research has primarily focused on the behaviour of individual methods. This research however, tries to do this differently by using both reads and writes in the same benchmark, since most applications do not use only one method. The methods that will be used for the benchmarks can be found in Table 1.

Table 1. Methods that will be benchmarked

Method	**Description**
add(Object)	Adds an element to the end of the list
contains(Object)	Returns whether an element is in the list
get(int)	Returns the element at a given position
iterator()	Allows an iteration of the list

3 BENCHMARKS

3.1 Selected Implementations

For the benchmarks, the following implementations will be used:

- *CopyOnWriteArrayList* (COW)
- *SynchronizedArrayList* (SAL)
- *SynchronizedLinkedList* (SLL)
- *Vector* (Vec)

3.2 System

The Java Virtual Machine (JVM) provides a lot of features, aimed at making the life of developers easier and ensuring that code is executed faster. Two of these features are the Just In Time (JIT) compiler and the Garbage Collector (GC). While usually a developer should not notice their effects, they can have a big influence on the benchmarks [15].

The GC ensures that developers do not need to worry about manual memory management. This does not come for free, they influence the performance of the benchmarks [9]. To reduce the effects imposed by the GC during benchmarking, a new GC has been released for Java 11: Epsilon. This GC does not free memory like a regular GC, but instead shuts the program down when it runs out of memory [36]. Unfortunately, the devices used in this research do not have enough memory to support benchmarking with this GC.

The JIT compiler on the other hand, attempts to improve the performance. Code in Java is not compiled directly to machine code, but to an intermediate representation: Java bytecode. When Java code runs, it will first interpret this bytecode. If the interpreted code runs a few times, the JIT can decide to optimize it and compile it to machine code. This means that in order to have consistent results, it is important to have a warm-up period [15].

During the optimizations, the JIT compiler can also detect whether a block of code is redundant. If for example 1 + 1 is calculated, but the result is never used, the JIT compiler can decide to erase the code. Usually this is convenient since it improves performance, but in this case, it also means that it can remove (parts of) the benchmarks if they are not written with enough care. To prevent this, the benchmarking software of the OpenJDK team, called JMH, is used. JMH is a microbenchmarking framework that gives access to *Blackholes*. When a value is fed to such a *Blackhole*, it is guaranteed to not be removed by the JIT compiler [19].

For the assessment of energy consumption, we will used a modified version of JoularJX [13]. While JoularJX is able to monitor Linux machines with Intel processors and any Windows machine, it is still lacking a way to estimate the energy consumption for other platforms. We implement this in our version, along with a lot of refactoring to make the code easier to read and make it possible to write the results to a CSV file will help during the analysis [14].

To assess the memory usage for the different implementations, another tool from the OpenJDK team is used: JOL. JOL can accurately determine how much memory is used in total by an object in Java [20]. This is used instead of the *Instrumentation#getObjectSize(Object)* method provided by Java, since according to its specification, it "is useful for comparison within an implementation but not between implementations" [28].

To run the benchmarks, the latest version of Java at the time of writing this will be used, which is Java 18, in the form of OpenJDK 18.0.1. It will run on three different devices:

- A laptop (Lenovo) running Windows 10 Home 21H1, an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz and 16GB of RAM
- A laptop (MSI) running Ubuntu 20.04.4 LTS, an Intel(R) Core (TM) i5-4210H CPU @ 2.90GHz and 8GB of RAM

- A Raspberry Pi 4 B running Ubuntu 21.04, Broadcom 2711, Cortex-A-72, 64-bit SoC @ 1.5 GHz and 2GB of RAM

The Raspberry Pi is used since it has an ARM architecture instead of an x86-based architecture like the laptops. This is more common in mobile devices and is starting to gain more ground in computers [3, 6]. On ARM based processors, re-orderings of reads and writes are more likely to happen. Therefore the effect of the happens-before relation imposed by for example *volatile*s might be different [4, 11].

4 PERFORMANCE

4.1 Hypothesis

The hypothesis for the performance is that when functions are combined, their time will add up.

To express this hypothesis as a formula, we use the following definitions: $a^{\lambda}_{i,t}$ the time in seconds for an operation where i is the ith action, t threads, λ is the list implementation and w_i how many times the ith action is repeated. With this, we can express the expected time A in Equation 1.

$$A(t, \lambda, w_1, w_2) = w_1 a^{\lambda}_{1,t} + w_2 a^{\lambda}_{2,t} \tag{1}$$

Since this research focuses on the ratio between w_1 and w_2 and instead of their absolute values, we will for simplicity take w_2 as 1. This means that A is now a linear equation of one variable. This can be used to find the w_1 for which the throughput of a composite action is equal for the implementations λ and σ. This is shown in Equation 2.

$$w_1 = \frac{a^{\sigma}_{2,t} - a^{\lambda}_{2,t}}{a^{\lambda}_{1,t} - a^{\sigma}_{1,t}} \tag{2}$$

With Equation 2 it is possible to find the domains in which each list is expected to achieve the highest throughput. To do this, we will use the property that the equation is linear. First we find the implementation with the lowest A if $w_1 = 0$. This equation is the fastest at a ratio of 0. After that, we can find the closest intersection with another implementation, which is the fastest starting at that ratio until its next intersection. We repeat this until no line crosses the fastest line.

4.2 Results

To test the hypothesis, we will first look at the results of the Add and Contains performance. Based on this, we can get the measured performance of the ContainsAdd operation, with 30 Contains operations for each Add operation and compare it to the expected performance. First we will look at the Contains operation in Figure 1. Note that the SynchronizedArrayList and the Vector are stacked.

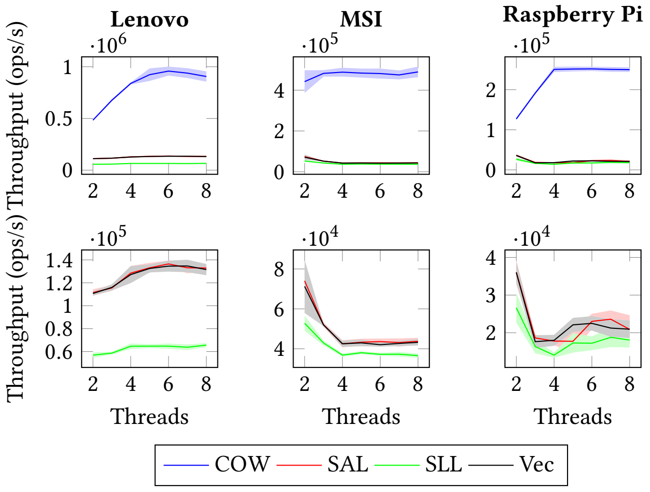

Fig. 1. Contains Performance (second row without COW)

Figure 1 clearly shows that the CopyOnWriteArrayList outperforms the other implementations on the Contains operation by quite a big margin. The throughput of the SynchronizedArrayList and Vector are almost equal to each other and the SynchronizedLinkedList is performing by far the worst. We will now look at how this compares to the Add performance in Figure 2.

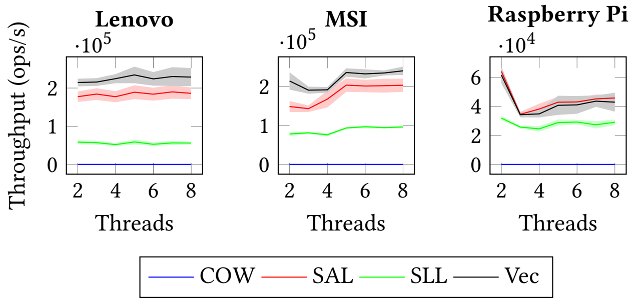

Fig. 2. Add Performance

From Figure 1 and 2 it is clear that even though the CopyOnWriteArrayList's Contains performance is head and shoulders above its competition, its Add performance is so bad that the combination of the two actions is expected to result in the worst throughput. The SynchronizedArrayList and Vector perform nearly equal in both operations and by coming in second for the Contains operation and being the fastest for the Add operation, they are expected to be the fastest on the combined action. Lastly the SynchronizedLinkedList does not really impress, with a last place for the Contains operation and second to last place for the Add operation. However neither operations are as bad as the Add performance for the CopyOnWriteArrayList, it should not come in last on the composite action. In Figure 3 we show that this indeed the case. The first row is the measured performance and the second row the predicted performance based on Equation 1.

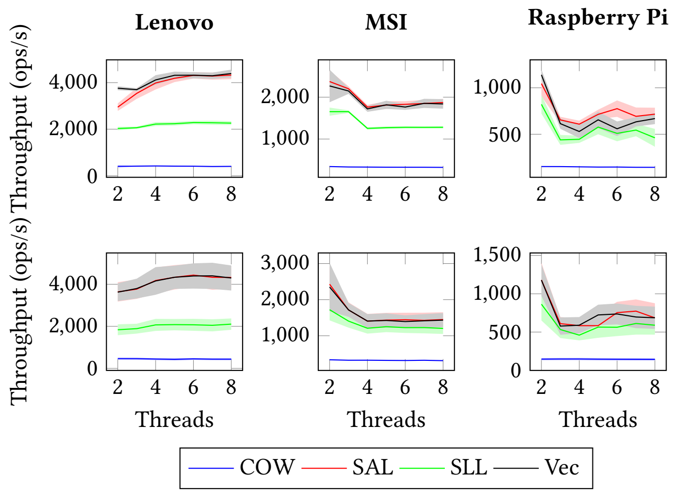

Fig. 3. ContainsAdd Performance (first row measured, second row predictions, SAL and Vec are stacked)

While the shape of the graphs on the first row (the measured values) and the second row (the predicted values based on the hypothesis) seem to match, they do still not overlap for all graphs. This happens for most most actions. This leads to the percentages of correct predictions on the platforms shown in Table 2.

Table 2. Percentage correctly throughputs estimated based on Equation 2 (Lenovo,MSI,Pi)

	COW	SAL	SLL	Vec
GetContains	100,79,71	100,43,93	79,79,86	100,36,86
GetIterate	71,43,100	100,79,86	100,100,93	93,64,86
GetAdd	67,81, 71	100,67,95	29,0,67	95,43,95
ContainsIterate	94,100,7	100,57,100	36,100,93	93,64,100
ContainsAdd	57,52,57	62,33,95	52,76,95	100,38,90
IterateAdd	57,86,100	100,38,62	33,71,86	90,14,71

While for some actions, the hypothesis seems to give reasonable predictions, in most cases, the results are not too impressive. However, when we add an error margin of 10%, we get significantly better results, as shown in Table 3. In this table, the Lenovo is left out since everything is correctly predicted for it, with the exception of the GetAdd operation for the SynchronizedLinkedList. These predictions are incorrect since they are predicted based on the Get operation with a static size of 100. The Add operation however adds elements, which means that the size is dynamic. This influences the results since it starts looping to find the item, either at the start or the end of the list, depending on which is closer to the target index [18]. In the statically sized Get environment, this will be the head half of the time and the tail half of the time, but in the dynamically sized case, it will only start at the head of the list. The reason for this is that the Get action is only performed on the first 100 indexes, making the tail always further away than the head. This slows down the Get part of the GetAdd operation compared to the expectation based on the hypothesis. Since the other implementations rely on random access, this does not influence their results [30].

Table 3. Percentage correctly estimated based on the hypothesis with a 10% error margin (MSI,Pi)

	COW	SAL	SLL	Vec
GetContains	100,100	71,100	100,100	71,100
GetIterate	100,100	100,100	100,100	100,100
GetAdd	100,100	100,100	43,90	81,100
ContainsIterate	100,100	71,100	100,100	100,100
ContainsAdd	67,90	86,95	100,100	95,100
IterateAdd	100,100	100,100	95,86	90,100

These predictions seem to be way more accurate (again, without taking the GetAdd for the SynchronizedLinkedList into account). However, it might seem unreasonable to arbitrarily add a 10% error margin to the predictions and call it a day. However, there is a reason to do this. The Raspberry Pi does not have any type of cooling and the other benchmarks run on laptops. Laptops do not have a lot of space for cooling, so especially when it is a bit hotter outside, it can be hard for them to get rid of their heat. Especially the MSI has problems with it, during the summer its keyboard can get scorching hot and with the benchmarks running over a period of 15 hours during a hot day [41], the MSI got hot. Moreover, during such a long run at which the CPU is frequently running at 100%, the CPU heats up. When a CPU gets too hot, it can start to thermal throttle [10]. This means that to protect the CPU, when it gets too hot, the CPU power will temporarily be reduced. Even in a two minute benchmark on a laptop, this can easily lead to a difference in clock speed of more than 10% [38].

By keeping this in mind, we can now use Equation 2 to estimate when an implementation is expected to outperform the others. The values are written down as ranges of the ratio between the first and second operation. For example a table entry of $0 - 700$ for the GetAdd operation means that the implementation is expected to be the fastest when there are between 0 and 700 Get operations for each Add operation. These results are show in Table 4 (Lenovo), Table 5 (MSI) and Table 6 (Raspberry Pi).

Table 4. Expected fastest implementations on the Lenovo

	COW	SAL	SLL	Vec
GetContains	$0 - \infty$			
GetIterate	$0 - \infty$			
GetAdd	$700 - \infty$	$0 - 700$		$0 - 700$
ContainsIterate	$0 - \infty$			
ContainsAdd	$300 - \infty$	$0 - 300$		$0 - 400$
IterateAdd	$10000 - \infty$	$0 - 300$	$300 - 10000$	$0 - 300$

Table 5. Expected fastest implementations on the MSI

	COW	SAL	SLL	Vec
GetContains	$0-\infty$			
GetIterate	$0-\infty$			
GetAdd	$1500-\infty$	$0-1500$		$0-1500$
ContainsIterate	$0-\infty$			
ContainsAdd	$150-\infty$	$0-150$		$0-150$
IterateAdd	$11000-\infty$	$0-300$	$300-11000$	$0-300$

Table 6. Expected fastest implementations on the Raspberry Pi

	COW	SAL	SLL	Vec
GetContains	$0-\infty$			
GetIterate	$0-\infty$			
GetAdd	$500-\infty$	$0-500$		$0-500$
ContainsIterate	$0-\infty$			
ContainsAdd	$150-\infty$	$0-150$		$0-150$
IterateAdd	$6000-\infty$	$0-100$	$100-6000$	$0-100$

5 ENERGY

5.1 Hypothesis

The hypothesis for the energy consumption is that it will be the average of the power consumed by each action in the composite action.

If we use the notation introduced earlier and introduce the notation $P^{\lambda}_{i,t}$ for the energy consumed by the ith operation, for t threads on implementation λ, we can express the expected energy E in Equation 3.

$$E(t,l)=\frac{wP^{\lambda}_{1,t}+P^{\lambda}_{2,t}}{w+1} \tag{3}$$

5.2 Results

The hypothesis, again with a 10% error margin, is used to assess whether it is a good indication of the power consumption, the results can be found in Table 7. The Lenovo is not in this table since all its entries are predicted correctly.

Table 7. Percentage correctly estimated energy consumption based on the hypothesis (MSI,Pi)

	COW	SAL	SLL	Vec
GetContains	86,100	100,100	100,100	100,100
GetIterate	100,100	100,100	100,100	100,100
GetAdd	33,5	100,100	100,100	100,100
ContainsIterate	100,100	100,100	100,100	100,100
ContainsAdd	67,10	100,100	100,100	100,100
IterateAdd	33,5	100,100	100,100	71,100

While most actions are predicted correctly, the predictions for the composite actions where read and write operations are combined for the CopyOnWriteArrayList, seem to fall short. A straight forward explanation for this, is that the CopyOnWriteArrayList has such a slow Add operation, that it spends most of its time doing on the Add operation, so the Contains operation does not add a lot to the power consumption. However, the formula gives a higher weight to the Contains operation since there are more of them.

Based on this observation, it is possible to come up with a new hypothesis. This new hypothesis also relies on the hypothesis of the performance data. Instead of multiplying the power for each action by the ratio of how often the action is executed, it gets multiplied by the ratio of how long the actions take. This is shown in Equation 4.

$$E(t,\lambda,w)=P^{\lambda}_{1}\frac{wa^{\lambda}_{1,t}}{wa^{\lambda}_{1,t}+a^{\lambda}_{2,t}}+P^{\lambda}_{2}\frac{a^{\lambda}_{2,t}}{wa^{\lambda}_{1,t}+a^{\lambda}_{2,t}} \tag{4}$$

After using Equation 4 to predict the energy consumption, it turns out that it achieves a score of 100% on all three platforms. Therefore we will continue using this formula.

If one wants to find when the energy consumption at a given moment of two different implementation λ and σ are equal to each other, it is possible to solve for w. However, the equation is quite big, so instead the quadratic equation is given in Equation 5.

$$\begin{aligned}&(P^{\lambda}_{1,t}a^{\lambda}_{1,t}a^{\sigma}_{1,t}-P^{\sigma}_{1,t}a^{\lambda}_{1,t}a^{\sigma}_{1,t})w^2\\&\quad+(P^{\lambda}_{1,t}a^{\lambda}_{1,t}a^{\sigma}_{2,t}+P^{\lambda}_{2,t}a^{\lambda}_{2,t}a^{\sigma}_{1,t}-P^{\sigma}_{1,t}a^{\lambda}_{2,t}a^{\sigma}_{1,t}-P^{\sigma}_{2,t}a^{\lambda}_{1,t}a^{\sigma}_{2,t})w\\&\qquad+P^{\lambda}_{2,t}a^{\lambda}_{2,t}a^{\sigma}_{2,t}-P^{\sigma}_{2,t}a^{\lambda}_{2,t}P^{\sigma}_{2,t}=0\end{aligned} \tag{5}$$

While Equation 5 might be useful in some cases, we will focus more on how much energy is consumed per operation rather than calculating which implementation uses the least energy at a given moment. This can be calculated by dividing the energy consumption by the throughput. If we do this with the hypotheses for the performance and energy consumption, in the form of Equation 1 and 4, we get Equation 6.

$$w=\frac{P^{\sigma}_{2,t}a^{\sigma}_{2,t}-P^{\lambda}_{2,t}a^{\lambda}_{2,t}}{P^{\lambda}_{1,t}a^{\lambda}_{1,t}-P^{\sigma}_{1,t}a^{\sigma}_{1,t}} \tag{6}$$

This results in indications at which ratio each implementation is the most energy efficient on each platform (Lenovo: Table 8, MSI: Table 9, Raspberry Pi: Table 10). Keep in mind that these values are not absolutes. They are an indication of how many times the first operation (for example Get in GetContains) needs to be present for each second operation (for example Contains in GetContains) in order to be the expected to be the implementation with the least energy consumption.

Table 8. Expected greenest implementations on the Lenovo

	COW	SAL	SLL	Vec
GetContains	$0 - \infty$			
GetIterate	$0 - \infty$			
GetAdd	$700 - \infty$	$0 - 700$		$0 - 700$
ContainsIterate	$0 - \infty$			
ContainsAdd	$400 - \infty$	$0 - 400$		$0 - 400$
IterateAdd	$11000 - \infty$	$0 - 500$	$500 - 11000$	$0 - 500$

Table 9. Expected greenest implementations on the MSI

	COW	SAL	SLL	Vec
GetContains	$0 - \infty$			
GetIterate	$0 - \infty$			
GetAdd	$600 - \infty$	$0 - 600$		$0 - 600$
ContainsIterate	$0 - \infty$			
ContainsAdd	$200 - \infty$	$0 - 200$		$0 - 200$
IterateAdd	$8000 - \infty$	$0 - 400$	$400 - 8000$	$0 - 400$

Table 10. Expected greenest implementations on the Raspberry Pi

	COW	SAL	SLL	Vec
GetContains	$0 - \infty$			
GetIterate	$0 - \infty$			
GetAdd	$400 - \infty$	$0 - 400$		$0 - 400$
ContainsIterate	$0 - \infty$			
ContainsAdd	$150 - \infty$	$0 - 150$		$0 - 150$
IterateAdd	$5000 - \infty$	$0 - 200$	$200 - 5000$	$0 - 200$

6 MEMORY

In this research, two types of memory consumption are assessed, which we will refer to as *instant* and *running* memory. Instant memory is the memory a list requires to hold a given amount of objects. For the instant memory, the list already knows its target size in advance, so it is not the same as repeatedly adding items to a list. Running memory on the other hand, is the total amount of memory used by a collection if items get continuously added. This means that running memory includes memory that is no longer in use, while the instant memory is all in use.

6.1 Hypothesis

For the instant memory, the hypothesis is that the memory usage will be nearly equal. For the running memory, it is expected that the CopyOnWriteArrayList will be worse than the others, since it has to allocate a new array on each modification. Moreover, since the SynchronizedLinkedList does not create any wasted memory when objects are added, it is expected that it has the lowest running memory.

6.2 Instant

The instant memory is the total amount of bytes that are used for the list to store the given amount of items. In this case, Integers are used since they take 16 bytes of storage, which is the minimal size of a Java object [39]. The results are plotted in Figure 4.

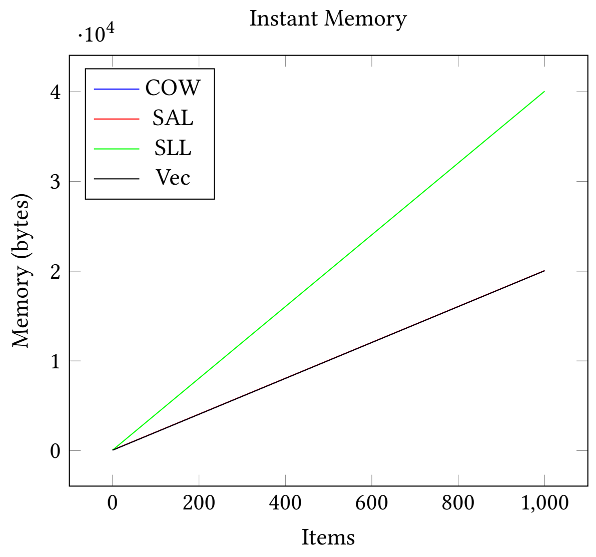

Fig. 4. Instant Memory Consumption (COW, SAL and Vec are stacked)

From Figure 4, it is clear that the SynchronizedLinkedList takes significantly more bytes (almost 2 times) to store the same data as the other implementations. It is not immediately clear how the other implementations compare, but they seem quite similar.

To understand the differences between the instant memory of the CopyOnWriteArrayList, SynchronizedArrayList and Vector, we will express their memory consumption as functions. The function $m(n, s)$ calculates the amount of bytes needed to store a list with n elements, in which each element has a size of s bytes. This means that $n, s \in \mathbb{N} \cup \{0\}$.

To understand these formulae, it is important to know that the JVM always requires an object to have a size that is a multiple of 8 bytes. This is in particular interesting for arrays, since a reference only requires 4 bytes [37]. This means that if there are an odd number of elements, the array still allocates memory for one more element. This is not the case with an even number of elements, then the required amount of memory is allocated.

To handle this padding, we need some notation to use in the formulae. This will be the notation $\lceil f \rceil_2$. It means that the result will be ceiled to an even number. The mathematical definition is given in Equation 7.

$$\lceil f \rceil_2 = \begin{cases} \lceil f \rceil, & \text{if} \lceil f \rceil \in 2\mathbb{Z} \\ \lceil f \rceil + 1, & \text{if} \lceil f \rceil \notin 2\mathbb{Z} \end{cases} \tag{7}$$

This behaviour can be achieved with regular operators by using $\lceil f \rceil_2 = 2\left\lceil \frac{f}{2} \right\rceil$. If $\lceil f \rceil \in \mathbb{Z}$, it is also possible to use $\lceil f \rceil_2 = f + \frac{1}{2} -$

$\frac{1}{2}(-1)^f$ which might have nicer mathematical properties depending on the use case.

With these symbols and notations it is now possible to express the instant memory for each implementation as a function of the number of elements and the size of each object. In this listing, we address the constant term for the CopyOnWriteArrayList in detail, but we will not go into that depth for the others.

CopyOnWriteArrayList

$$m(n,s) = 56 + sn + 4\lceil n \rceil_2 \tag{8}$$

- 56: 16 bytes for the lock, 16 bytes for the header of the array, 12 bytes of object header, 4 bytes to store the reference to the lock, 4 bytes to store the reference to the array, 4 bytes padding
- sn: the memory of the stored elements
- $4\lceil n \rceil_2$: the memory used for the data part of the array. 4 bytes per element, but must always be a multiple of 8 so it might have 4 bytes of padding

SynchronizedArrayList

$$m(n,s) = 64 + sn + 4\lceil n \rceil_2 \tag{9}$$

- 64: the memory for an empty SynchronizedArrayList
- sn: the memory of the stored elements
- $4\lceil n \rceil_2$: the memory used for the data part of the array. 4 bytes per element, but must always be a multiple of 8 so it might have 4 bytes of padding

SynchronizedLinkedList

$$m(n,s) = 56 + 24n + sn \tag{10}$$

- 56: the memory for an empty SynchronizedLinkedList
- $24n$: 12 bytes for the header of each Node, 4 bytes for the reference to the item, 4 bytes for the reference to the previous Node, 4 bytes for the reference to the next Node
- sn: the memory of the stored elements

Vector

$$m(n,s) = 48 + sn + 4\lceil n \rceil_2 \tag{11}$$

- 48: the memory for an empty Vector
- sn: the memory of the stored elements
- $4\lceil n \rceil_2$: the memory used for the data part of the array. 4 bytes per element, but must always be a multiple of 8 so it might have 4 bytes of padding

Equation 8, 9 and 11 show that the memory for the CopyOnWriteArrayList, SynchronizedArrayList and Vector, only differ by a constant factor. Moreover, this difference is not bigger than 16 bytes, which is the minimum size of an object. This means that in almost all practical use cases, this difference is negligible.

6.3 Running

At first, the running memory might not seem very interesting since not all of the memory is 'in use' at the same time. However, if memory is unused, it is not automatically available again. First the garbage collector needs to free the memory, which can be expensive [9]. The running memory is shown in Figure 5.

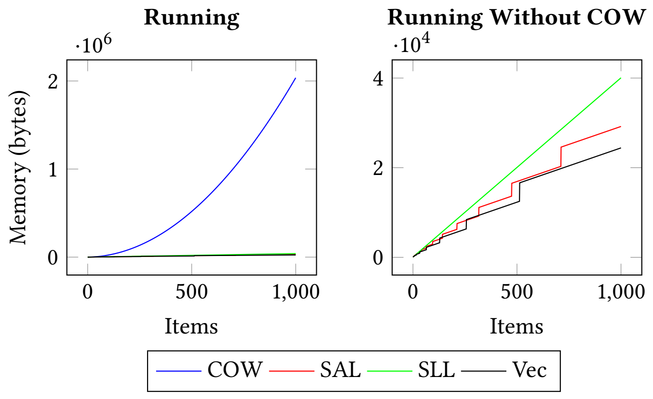

Fig. 5. Running Memory Consumption (with and without COW)

The first plot shows that the CopyOnWriteArrayList gathers significantly more running memory than the other implementations. This makes sense since it has to create a new array after every single modification. This renders the CopyOnWriteArrayList inappropriate when running memory is a concern for the developer.

The SynchronizedLinkedList's running memory is growing linearly, in fact, it is equal to the instant memory, since it does not need to discard anything when more elements are added. The SynchronizedArrayList and the Vector on the other hand, grow in steps, since they grow the capacity of their underlying array is full. The size of the new underlying array, given the previous capacity n are $\lfloor \frac{3}{2}n \rfloor$ and $2n$ respectively. This means that the SynchronizedArrayList has less redundant capacity in its array after a resize, but the Vector has a lower running memory footprint.

To be able to understand the differences between the implementations, we shall express the running memory in the Equations 12, 13, 14 and 15. We will not cover the size of the empty collections in depth.

CopyOnWriteArrayList

$$m(n) = 57 + sn + 20n + 2n^2 - (-1)^n \tag{12}$$

- 56: the memory for an empty CopyOnWriteArrayList
- sn: the memory of the stored elements
- $16n$: 16 bytes for each header of an allocation of an array
- $1 + 4n + 2n^2 - (-1)^n$: the sum of the memory used for the data part of the array, with 4 bytes for each element with compensation for the padding to 8 bytes $(1 - (-1)^n)$

SynchronizedArrayList

$$m(n) = 64 + sn + 16\left(\left\lceil \log_{\frac{3}{2}} \frac{n-1}{K} \right\rceil + 1\right) + 4 \sum_{i=0}^{\left\lceil \log_{\frac{3}{2}} \frac{n-1}{K} \right\rceil} \left\lceil K\left(\frac{3}{2}\right)^i \right\rceil_2 \tag{13}$$

Where $K \approx 1.08151366859$, it is $\frac{2}{3}K(3)$ where $K(3)$ is a constant related to the Josephus problem [1].

- 64: the memory for an empty SynchronizedArrayList
- sn: the memory of the stored elements

- $16\left(\left\lceil \log_{\frac{3}{2}} \frac{n-1}{K} \right\rceil + 1\right)$: 16 bytes for each header of an allocation of an array, the first 16 are added because the first allocation does not get covered by the rest of the formula
- $4\sum_{i=0}^{\left\lceil \log_{\frac{3}{2}} \frac{n-1}{K} \right\rceil} \left\lceil K\left(\frac{3}{2}\right)^i \right\rceil_2$: goes over each allocation and uses 4 bytes for each element in an array, but pads to a multiple of 8 bytes if required

SynchronizedLinkedList

$$m(n,s) = 56 + 24n + sn \tag{14}$$

- 56: the memory for an empty SynchronizedLinkedList
- $24n$: 12 bytes for the header of each Node, 4 bytes for the reference to the item, 4 bytes for the reference to the previous Node, 4 bytes for the reference to the next Node
- sn: the memory of the stored elements

Vector

$$m(n) = 48 + sn + 16(\lceil \log_2 n \rceil + 1) + 8 \cdot 2^{\lceil \log_2 n \rceil} \tag{15}$$

- 48: the memory for an empty Vector
- sn: the memory of the stored elements
- $16(\lceil \log_2 n \rceil + 1)$: 16 bytes for each header of an allocation of an array, the first 16 are added because the first allocation does not get covered by the rest of the formula
- $8 \cdot 2^{\lceil \log_2 n \rceil}$: the sum of the memory used for the data part of the array, with 4 bytes for each element with compensation for the padding to 8 bytes

From Equation 14 it is clear that the SynchronizedLinkedList is still linear, and in fact equal to the instant memory. The CopyOnWriteArrayList however, does change quite a lot. According to Equation 12 it is quadratic, which means that it will always consume more memory than the SynchronizedLinkedList.

For the Vector (Equation 15), it might be a bit harder to spot how it scales. However, even though the term $2^{\lceil \log_2 n \rceil}$ might look like it is exponential at first glance, the log in the exponent makes the term effectively linear. This means that this equation is essentially linear with an additional $\lceil \log_2 n \rceil$ term. Logarithms grow slower than linear terms, so the Vector should always stay lower than the SynchronizedLinkedList.

Lastly, there is Equation 13, that describes the growth of the SynchronizedArrayList. This one is the most complicated looking formula and therefore perhaps the hardest to compare to the others. To understand it, we need to understand where the formula comes from. When a factor resizes, it multiplies its capacity by $\frac{3}{2}$, whereas the Vector multiplies its capacity by 2. This means that it has the same growth properties as the Vector, which means that it will always consume less running memory than the CopyOnWriteArrayList and the SynchronizedLinkedList, but a slightly more than the Vector.

7 DISCUSSION

For both the performance and the energy consumption, the read-only operations are dominated by the CopyOnWriteArrayList. The operations which also contain the Add action, are dominated by the Vector and SynchronizedArrayList, until there are a few hundred times more read operations than write operations, at which time the CopyOnWriteArrayList is the most optimal implementation (Tables 4-6, 8-10).

It is important to mention that especially due to the thermal throttling, these estimations should not be treated as absolute numbers. They can only be used to get a feeling for the order of magnitude for when which implementation is faster. Moreover, the device used to run a program that uses this information might not have the same results as the devices used in this research. The Lenovo, MSI and Raspberry Pi already get results differing quite a lot, but the order of magnitude stays the same.

Another thing to keep in mind, is that the results are based on lists that start of with a size of 100. Moreover, the Get, Contains and Iterate actions are limited to only the first 100 elements of the Lists. This means that the numbers might differ for larger lists.

For the Iterate, we have also not tested different types of bodies. Since contradictory to the other implementations, the CopyOnWriteArrayList does not require locking, it could be that especially with a heavier calculation in the body, the numbers start to shift in favour of the CopyOnWriteArrayList.

The instant memory at each moment in time is almost equal for the CopyOnWriteArrayList, SynchronizedArrayList and Vector. The SynchronizedLinkedList is the only implementation with a different, but worse, memory profile. The running memory is a different story. Since the CopyOnWriteArrayList has to make a copy on each write, it has a way worse running memory usage. The SynchronizedArrayList and the Vector both grow in steps when the capacity is reached. The Vector doubles in capacity at each grow, while the SynchronizedArrayList multiplies its capacity by a factor of 1.5. This means that the Vector typically has more unused space than the SynchronizedArrayList, but in return has a better running memory footprint.

For the instant memory it is also important to mention that the lists are not grown by adding each items individually, but in one operation. This means that the lists have the possibility to get a capacity that exactly matches the final size. This means that for the Vector and SynchronizedArrayList the redundant capacity is not presented in this research.

8 CONCLUSIONS AND FUTURE WORK

Our results show that in most practical scenarios, which use the methods we tested (Add, Contains, Get, Iterate), the SynchronizedArrayList and the Vector are the best choices for thread-safe list implementations. Only when there are hundreds of times more read operations than write operations, the CopyOnWriteArrayList becomes more feasible to use, but only if running memory is not a concern to the developer.

As future work, we recommend that the experiments run in a temperature controlled environment to attempt to achieve more accurate results. Moreover, to get a better picture of the behaviour of servers and desktops, it would help to run the benchmarks on these types of machines. It would also be useful to see the Remove operation being investigated. Lastly, the impact of the body of the Iterate functionality on the performance could still be investigated.

REFERENCES

[1] Henry Bottomley. 2001. *Entry A061418 in The On-Line Encyclopedia of Integer Sequences.* OEIS Foundation Inc. https://oeis.org/A061418 *Last accessed: 26/06/2022.*

[2] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. 2017. Empirical Study of Usage and Performance of Java Collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering.* ACM, New York, United States, 389–400. https://doi.org/10.1145/3030207.3030221

[3] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. 2021. An Exploration of ARM System-Level Cache and GPU Side Channels. In *37th Annual Computer Security Applications Conference.* ACM, New York, United States, 784–795. https://doi.org/10.1145/3485832.3485902

[4] ARM Developer. 2011. *Memory Ordering.* ARM Developer. https://developer.arm.com/documentation/den0042/a/Memory-Ordering *Last accessed: 26/06/2022.*

[5] Dave Dice and Danny Hendlerand Ilya Mirsky. 2014. Software-based contention management for efficient compare-and-swap operations. *Combined Special issues on Euro-Par 2013 and Java Technologies for Real-Time and Embedded Systems* 26, 14 (2014), 2386–2404. https://doi.org/10.1002/cpe.3304

[6] Blake Ford, Apan Qasem, Jelena Tesic, and Ziliang Zong. 2021. Migrating Software from x86 to ARM Architecture: An Instruction Prediction Approach. In *15th IEEE International Conference on Networking, Architecture and Storage.* IEEE, New York, United States, 1–6. https://doi.org/10.1109/NAS51552.2021.9605443

[7] Brian Goetz. 2006. *Java Concurrency in Pratice.* Addison-Wesley, Glenview, United States.

[8] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy profiles of Java collections classes. In *Proceedings of the 38th International Conference on Software Engineering.* ACM, New York, United States, 225–236. https://doi.org/10.1145/2884781.2884869

[9] IBM. 2022. *Garbage collection impacts to Java performance.* IBM. https://www.ibm.com/docs/en/aix/7.2?topic=monitoring-garbage-collection-impacts-java-performance *Last accessed: 26/06/2022.*

[10] Intel. 2021. *Why does Intel® Core™ Processors Enable the Throttling While Gaming?* Intel. https://www.intel.com/content/www/us/en/support/articles/000029868/processors/intel-core-processors.html *Last accessed: 26/06/2022.*

[11] Lun Liu, Todd Millstein, Madanlal, and Musuvathi. 2017. A volatile-by-default JVM for server applications. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–24. https://doi.org/10.1145/3133873

[12] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: a software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering.* ACM, New York, United States, 503–514. https://doi.org/10.1145/2568225.2568297

[13] Adel Noureddine. 2022. *JoularJX.* JoularJX. https://www.noureddine.org/research/joular/joularjx *Last accessed: 26/06/2022.*

[14] Adel Noureddine and Mark van Wijk. 2022. *JoularJX.* JoularJX. https://github.com/Chickenpowerrr/joularjx *Last accessed: 26/06/2022.*

[15] Scott Oaks. 2014. *Java Performance: The Definitive Guide.* O'Reilly, Sebastopol, United States.

[16] Scott Oaks and Henry Wong. 2004. *Java Threads: Understanding and Mastering Concurrent Programming.* O'Reilly, Sebastopol, United States.

[17] Wellington Oliveira, Renato Oliveira, Fernando Castor, Benito Fernandes, and Gustavo Pinto. 2019. Recommending Energy-Efficient Java Collections. In *IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).* IEEE, New York, United States, 160–170. https://doi.org/10.1109/MSR.2019.00033

[18] OpenJDK. 2011. *view src/share/classes/java/util/LinkedList.java.* OpenJDK. https://hg.openjdk.java.net/jdk7/jdk7/jdk/file/9b8c96f96a0f/src/share/classes/java/util/LinkedList.java#l567 *Last accessed: 26/06/2022.*

[19] OpenJDK. 2022. *Java Microbenchmark Harness (JMH).* OpenJDK. https://github.com/openjdk/jmh *Last accessed: 26/06/2022.*

[20] OpenJDK. 2022. *Java Object Layout (JOL).* OpenJDK. https://github.com/openjdk/jol *Last accessed: 26/06/2022.*

[21] Oracle. 2014. *Summary of Implementations (The Java Tutorials > Collections > Implementations).* Oracle. https://docs.oracle.com/javase/tutorial/collections/implementations/summary.html *Last accessed: 26/06/2022.*

[22] Oracle. 2014. *Synchronized Methods (The Java™ Tutorials > Essential Java Classes > Concurrency).* Oracle. https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html *Last accessed: 26/06/2022.*

[23] Oracle. 2022. *ArrayList (Java SE 18 & JDK 18).* Oracle. https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/ArrayList.html *Last accessed: 26/06/2022.*

[24] Oracle. 2022. *Chapter 17. Threads and Locks.* Oracle. https://docs.oracle.com/javase/specs/jls/se18/html/jls-17.html *Last accessed: 26/06/2022.*

[25] Oracle. 2022. *Collection (Java SE 18 & JDK 18).* Oracle. https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Collection.html *Last accessed: 26/06/2022.*

[26] Oracle. 2022. *CopyOnWriteArrayList (Java SE 18 & JDK 18).* Oracle. https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/concurrent/CopyOnWriteArrayList.html *Last accessed: 26/06/2022.*

[27] Oracle. 2022. *CopyOnWriteArraySet (Java SE 18 & JDK 18).* Oracle. https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/concurrent/CopyOnWriteArraySet.html *Last accessed: 26/06/2022.*

[28] Oracle. 2022. *Instrumentation (Java SE 18 & JDK 18).* Oracle. https://docs.oracle.com/en/java/javase/18/docs/api/java.instrument/java/lang/instrument/Instrumentation.html#getObjectSize(java.lang.Object) *Last accessed: 26/06/2022.*

[29] Oracle. 2022. *LinkedList (Java SE 18 & JDK 18).* Oracle. https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/LinkedList.html *Last accessed: 26/06/2022.*

[30] Oracle. 2022. *RandomAccess (Java SE 18 & JDK 18).* Oracle. https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/RandomAccess.html *Last accessed: 26/06/2022.*

[31] Oracle. 2022. *Vector (Java SE 18 & JDK 18).* Oracle. https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Vector.html *Last accessed: 26/06/2022.*

[32] Rui Pereira, Marco Couto, João Cunha, and João Paulo Fernandes. 2016. The influence of the Java collection framework on overall energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software.* ACM, New York, United States, 15–21. https://doi.org/10.1145/2896967.2896968

[33] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate?. In *Proceedings of the 36th International Conference on Software Engineering.* ACM, New York, United States, 256–267. https://doi.org/10.1145/2568225.2568297

[34] Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. 2018. jStanley: Placing a Green Thumb on Java Collections. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* IEEE, New York, United States, 856–859. https://doi.org/10.1145/3238147.3240473

[35] Gustavo Pinto, Kenan Liu, Fernando Castor, and Yu David Liu. 2016. A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections. In *32nd IEEE International Conference on Software Maintenance and Evolution.* IEEE, New York, United States, 20–31. https://doi.org/10.1109/ICSME.2016.34

[36] Aleksey Shipilev. 2017. *JEP 318: Epsilon: A No-Op Garbage Collector (Experimental).* OpenJDK. https://openjdk.java.net/jeps/318 *Last accessed: 26/06/2022.*

[37] Aleksey Shipilev. 2020. *Java Objects Inside Out.* Aleksey Shipilev. https://shipilev.net/jvm/objects-inside-out/#_observation_array_base_is_aligned *Last accessed: 26/06/2022.*

[38] Jelle Stuip. 2018. *Throttling van laptops.* Tweakers. https://tweakers.net/reviews/5909/all/throttling-van-laptops-vandaar-die-variatie-in-prestatie.html *Last accessed: 26/06/2022.*

[39] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. 2007. Java Object Header Elimination for Reduced Memory Consumption in 64-bit Virtual Machines. *ACM Transactions on Architecture and Code Optimization* 4, 3 (2007), 17. https://doi.org/10.1145/1275937.1275941

[40] Maarten Voorberg. 2021. *A performance analysis of membership datastructures in Java.* Bachelor's Thesis. University of Twente. http://essay.utwente.nl/87064/

[41] Weer1. 2022. *Hengelo historisch weer per uur | Weer1.cm.* Weer1. https://www.weer1.com/europe/netherlands/overijssel/hengelo?page=past-weather#day=18&month=6 *Last accessed: 26/06/2022.*

[42] Guoqing Xu. 2013. CoCo: Sound and Adaptive Replacement of Java Collections. In *27th European Conference on Object-Oriented Programming.* Springer, Heidelberg, Germany, 1–26. https://doi.org/10.1007/978-3-642-39038-8_1