Java Shared Memory Performance Races

Lawrence Chen, *University of California Los Angeles*

Abstract

When programs utilize multithreading to speed up applications, there are several concerns to consider if they operate on shared-memory representations of the state. Companies such as Ginormous Data Inc. (GDI) that utilize such functionalities need to worry about the tradeoffs between speed and accuracy. In the case of Java, synchronization is based on the Java memory model (JMM), which defines how an application can safely avoid data races when accessing shared memory. However, although Java's *synchronized* keyword can ensure that the state is updated safely, it can also be a bottleneck in the code. Our goal is to experiment with other methods that may be inadequate but faster. We will test the different methods using sequential consistency and reliability tests to determine if they achieve better performance, and ideally behave to be data-race free (DRF) as defined by JMM. The four version of classes implementing the simulation state that we will be testing are the following: Synchronized, Unsynchronized, GetNSet, and BetterSafe, with Synchronized being the original bottlenecked code we are looking to replace. We will be comparing these results with that of the Null class, a state class that does nothing, to get a baseline benchmark for the overall simulation.

1. Packages and Classes

There are several packages and classes that were considered in implementing the BetterSafe class. In the following sections we will discuss some of functionalities in each of these packages, and the pros and cons that are featured in each when deciding which to use.

1.1. java.util.concurrent [1]

This package contains a lot of utilities and tools that are really useful for helping create concurrent programs, which is something we need for our experiments. It provides many interfaces and classes, such as *ThreadFactory*, *Semaphore,* and some other concurrent versions of common data structures. These extensible frameworks and class functionalities are standardized and may be difficult to implement on our own, so it is convenient to have this package provide it for us. The immediate pro of this package is the flexibility it offers in its usage, which is notably different from using the *synchronized* keyword. It gives programmers flexibility in how to approach concurrent programming. However, a con is that many of these frameworks are too low-level, working at the queue and thread level in terms of ordering memory access. This incredible amount of customizability might be useful in other situations, but for our problem, it is an overkill and complicates the program.

1.2. java.util.concurrent.atomic [2]

This toolkit provides classes that support lock-free thread-safe programming on single variables. For example, there are *AtomicInteger* objects and *AtomicBoolean* objects. In fact, in our GetNSet class, we used the *AtomicIntegerArray* class from this package [3]. The pro of this class is that it allows really fine granularity control on read, modify, and write access of atomic data values. This means that multiple threads can modify different data structures within one object and not lock the entirety of it, increasing the parallelism achieved. There is no need to work with locking mechanisms, and help relieve the concern of deadlocks and remembering to free locks. The con is that if we are working with multiple variables that interact with each other or our updates are not atomic due to additional calculations that needs to be done, there may be inconsistencies as a whole, such as in an array. In *AtomicIntegerArray*, each element might be atomically accessed, but if multiple threads are working with this array, inconsistencies can happen if threads read and write across the shared memory space at the same time. This can be problematic as we will later see.

1.3. java.util.concurrent.locks [4]

This package contains interfaces and classes that provide a framework for locking and waiting for conditions. The mechanism is much more flexible than the built-in monitors and synchronization. For example, the locks are able to be applied in finer granularity than the *synchronized* keyword. This allows us to only lock the critical points of the code, or even lock and unlock across methods. The pro is that they are simple to use and allows a finer granularity than built-in monitors. However, a con would be that there are many different types of locks to choose from that a programmer needs to understand to use correctly. Unlocking is crucial to remember, and potential deadlocks are possible. Another possible con is that it locks the entire object, not just a specific variable, so multiple threads cannot access different parts of the shared memory space at the same time. For our situation, however, this is could be a good thing, because we are working with an array of numbers that interact with each other. Thus, it has 100% reliability, and a good candidate to experiment with for

this problem. The *ReentrantLock* class is a mutual exclusion lock with the same behavior as the *synchronized* keyword, but with extra capabilities [5]. Therefore, we will use this class to implement our BetterSafe class.

1.4. java.lang.invoke.VarHandle [6]

This class object is a dynamically strongly typed reference to a variable, or to a parametrically-defined family of variables. The class provides methods that allow various access modes, such as both plain and volatile read/write and compare-and-set. Similar to the *atomic* package, it allows lock-free thread-safe access to single variables. This package is the generalized version of that package, extending the functionality beyond just the primitive variables. Therefore, it has the same pro as *java.util.concurrent.atomic*, but extended to additional variable types, but also the same cons. Therefore, it was no used for our BetterSafe class.

2. Data and Results

Next, we will present the data and results gathered from running our sequential consistency and reliability tests, with different hyper parameters to see how each of the classes perform under different circumstances. To compare performance, we look at the time it took per swap transition performed. To compare reliability, we indicate whether or not the class appears to be DRF from our empirical tests. The following results were achieved by running the tests on OpenJDK version 11.0.2, with OpenJDK runtime environment 18.9 and 64-Bit Server VM 18.9. We also ran similar tests with additional parameters and on OpenJDK version 9 with OpenJDK runtime environment and 64-bit Server VM (build 9+181). This is all performed on a server with an Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz and 65755884 kB total memory.

Figure 1. Swap Transition Time (ns/transition) with Varying Number of Swap Transitions. State size set at 1000. Thread count set at 16. Max set at 127. Randomized state entries.

State Class	1,000 Swaps	10,000 Swaps	1,000,000 Swaps	DRF?
Null	73240.5	14555.3	3787.20	Yes
Synchronized	110209	33754.1	6453.29	Yes
Unsynchronized	93596.5	15983.6	4798.04	No
GetNSet	7314970	768180	_	No
BetterSafe	187504	34642.6	3596.18	Yes

3. Analysis

As we can see from the data that we collected, out of the four classes we tested, the BetterSafe class perfomed the best with the fastest results.

3.1. Synchronized Performance

This is the original state memory used by GDI. It has clear 100% reliability, and during low computation and low contention, seems to perform comparably with a memory with no synchronization at all. However, as the contention increases, Unsynchronized was able to compute in almost half the time. Clearly, there is a bottleneck to be addressed, like we suspected. Below, we analyze the three new state classes: GetNSet, Unsynchronized, and BetterSafe, with the original Synchronized class to see if there are any viable candidates to replace GDI's memory state.

3.2. GetNSet Performance

First, it is worth noting that GetNSet was clearly the worst performer, running almost 100x slower than the Null baseline for both 1,000 transitions and 10,000 transitions. It was usually unable to complete at 1,000,000 swap transitions to get a rigorous analysis done. In addition, GetNSet was not DRF, because it frequently runs into race conditions and bugs. Like mentioned before, this is due to the fact that *AtomicIntegerArray* only maintains atomicity for each entry of the array, and not the entirety of the object. Therefore, in between checking the two swapped values and updating them, other threads could intervene and interrupt, making the swap inconsistent. If two threads both get the value of an entry, and update the entry in staggered pattern, instead of getting two updates, there would only be one. Thus, it could result in a sum mismatch. A command that often failed the reliability test was `java UnsafeMemory GetNSet 16 100 127 50 34 22 35 67 94 101 12 38 42 79 98 9 114 63.` Therefore, we completely remove this implementation from consideration.

3.3. Unsynchronized Performance

Next, we see that Unsynchronized performed significantly faster than Synchronized at every number of transition level. However, it was also not DRF for the same reason as GetNSet. The class had no synchronization functionality whatsoever, so race conditions happened almost consistently. Multiple threads could read and write into the array at any time, so if one thread reads a value, another thread could completely change the expected state without the other thread knowing and before it can complete the transaction fully.

For example, if an entry is at 1, one thread may begin to decrement the value. If another thread interrupts and also notices

this value of 1, both threads will eventually finish the transaction and decrement the value twice, thus resulting in a negative number. It may run into negative numbers or sum mismatch, indicating sequential inconsistency. An example command that would give reliability failures was `java UnsafeMemory Unsynchronized 16 10000 127 50 34 22 35 67 94 101 12 38 42 79 98 9 114 63.` Thus, the lack of reliability is concerning, and will most likely not be a reasonable replacement for Synchronized.

3.4. BetterSafe Performance

Finally, we look at BetterSafe. This class performed worse than Synchronized and Unsynchronized at first, for smaller numbers of swap transitions, i.e. 1,000 swaps and 10, 000 swaps. However, once we increased the number of computation to 1,000,000, BetterSafe became significantly faster than the other two classes by a large margin. BetterSafe completed in half the time it took Synchronized. In addition, it was DRF, yet was also able to run faster than our completely Unsynchronized class, which was significant, because it clearly performed better in both reliability and speed. Most surprisingly, it was able to rival the Null class in its performance once a large enough computation was being done.

BetterSafe is faster than Synchronized, and is still 100% reliable. This is because while it reduced the overhead of having to use the *synchronized* keyword wrapping the entire method, it still provided mutual exclusion around the critical point of the code where read and write access was being performed. This is done by using the aforementioned *ReentrantLock* in the swap transition, locking the object before any read or write access, and unlocking write before the method returns. Only the critical point is locked, making the code more efficient. Because only a single thread can access these critical points, race conditions are avoided and thus guaranteed to have 100% reliability. While running with less computation, i.e. less contention among threads, the benefits of BetterSafe may not be apparent. However, in high computing and high contention scenarios (which GDI exclusively works in), BetterSafe is the clear and significant winner in both performance and its ability to maintain 100% reliability.

4. Difficulties and Problems

While testing and running the code, I also ran into lots of problems to evaluate the memory states. First, for the classes that were not DRF, i.e. GetNSet and Unsynchronized, it was easy to get caught in an infinite loop and have the program hang, which is useless for our analysis. Therefore, I ran a lot of tests on both Java 11.0.2 and Java 9, as well as tuned the parameters of number of threads, size of state, number of transactions, etc. to find a range of comfortable hyper parameters. Therefore, our data would be more complete, with actual results being returned. I ended up keeping the Maxval set at 127, state size at 1000, and thread count at 16 for the majority of the tests.

Another problem I faced was the fluctuations of results, because performing the tests on the server meant contention between other peoples' threads and jobs. Therefore, certain performance values and speed may not always be consistent. To try and address this issue, I ran my final tests in a short time frame, and made sure to average the time over about 10 tries. That way my results more accurately reflected the typical performance for each class.

5. Conclusion

Due to the tests and analysis performed above, it is clear that the BetterSafe class is the best choice for GDI's applications. It provides the 100% reliability that we would ideally like, yet performs better than Synchronized in high computation scenarios that GDI applications typically encounter.

6. References

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/package-summary.html

[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

[3] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicIntegerArray.html

[4] https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/locks/package-summary.html

[5] https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/locks/ReentrantLock.html

[6] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/invoke/VarHandle.html