Unsafe Memory: Java Memory Model Synchronization

Abstract

Multithreading is a requirement for modern applications. Despite the complexity, the performance benefits are unavoidable. The Java Memory Model is designed to handle these complexities by creating a set of rules that compilers must obey. I examine how threading and various packages designed for multithreading interact with the JMM. I thoroughly test the performance and correctness of a subset of these implementations by implementing an interface for a multithreaded environment. The goal of my tests was to determine which implementation would be best for a multithreaded environment in which correctness need not be guaranteed and performance is a must. This piece concludes with which implementation suits such an environment best.

1. Implementing BetterSafe

The State interface specifies three different functions used for testing the Java Memory Model (JMM). The primary method used to test is swap, which is specified to increment one value in an array while decrementing another. The point of this State interface is to characterize how different implementations perform under the JMM and the reliability of such implementations.

The BetterSafe class is a completely reliable implementation of the State interface. It is designed to be faster than SynchronizedState without incurring the cost of fully synchronizing methods. My implementation uses ReentrantLocks from java.util.concurrent.locks. However, Java supports many other methods of implementing thread-safe, concurrent programs, including: concurrent, concurrent.atomic, and the language package VarHandle. All of which were candidates for implementing BetterSafe. The following is a discussion of the tradeoffs of each package and relevant classes.

1.1. Concurrent

The Java.util.concurrent package supports Executors, Queues,Timing, Synchronizers, and Concurrent Collections. When designing my solution to BetterSafe, I considered using Concurrent Collections and Synchronizers.

I quickly found that Synchronizers did not suit my implementation. The core problem with BetterSafe is the randomness of the array accesses. None of the accesses are planned prior to their request, so there is not possibility of scheduling or coordination between threads. In particular, CyclicBarriers are unsuitable due to their notion of threads working separately and cyclically coordinating with each other. In our use case, the threads could be assigned to do the same thing and CyclicBarriers would provide no protection. Semaphores are suitable for this case, but the same protections are provided by Concurrent Locks (see 1.3). Thus, I did not use the Synchronizers provided by concurrent.

Additionally, the Concurrent Collections, while thread-safe, were not suitable for this use. The datatypes provided by the concurrent package allow for many concurrent reads and writes. Unfortunately, the only method of mutual exclusion on these data types is the use of a single lock. Effectively, this would incur the same cost as using a single lock or synchronizing method (how SynchronizedState is implemented).

1.2. Concurrent Atomic & VarHandle

These packages contain data structures for atomic use. AtomicIntegerArray is the most relevant structure for our purposes. As suggested by the name, it is an array of atomic integers. Thus, accesses and updates to items in the array are atomic. Unfortunately, this does not guarantee 100% reliability in all cases. While updates to the structure are atomic and threads cannot disrupt each other in that regard, any interaction with the structure between get() and set() (reads and writes) could cause the implementation to be unreliable. It is feasible to foolproof this process, but at the cost of performance.

Considerations for VarHandle are the same as with concurrent.atomic. While atomic operations reduce chance of failure, they do not guarantee reliability.

1.3. Concurrent Locks

This is the package I ultimately chose for implementation. By using ReentrantLocks, I assigned a lock to each position in the array of bytes being accessed. This implementation is very similar to the SynchronizedState implementation, but at a finer locking granularity. Upon each call to BetterSafe.swap(), a thread must acquire two locks, one for each item it is attempting to access. For a large array, the chance of collision is reduced, increasing performance. Additionally, this implementation guarantees no data races by locking the entire critical section. I used a simple comparison to partially order locking and releasing, guaranteeing no deadlocks.

I took this design approach due to the use case described by the specification. GDI intended to do computation on large amounts of data concurrently. My implementation

assumes a large dataset and many accesses to the structure. This assumption is due to the overhead of locking, which is amortized as accesses increase.

2. Performance & Reliability

To test BetterSafe against other implementations, I designed a full test suite using bash scripts. My testing was performed on a UCLA Linux Server powered by an 8 core Intel Xeon E5-2640 at 2.30 GHz core clock with a 20MB cache and 64 GB of main memory.

State implementations used:

- SynchronizedState: swap() written as a synchronized method.
- UnsynchronizedState: no protections.
- NullState: Baseline, no-op.
- GetNSet: implementation using AtomicIntegerArray to lock.

2.1. Methodology

In total, I ran over 900 tests on each method to characterize performance and reliability of each implementation. By varying the number of threads used, transitions of each thread, max value passed, and size of the array passed to program, I created a set of base tests. I then ran those tests 10 times per method to come up with an average, accounting for any odd results.

2.2. Challenges

Due to the use of a UCLA Server, I encountered many issues surrounding the resource heavy nature of testing. Characterizing a "normal" result was difficult due to the varying stresses on the servers. Thus my averaging of multiple of the same tests.

Additionally, though BetterSafe, SynchronizedState, and NullState are datarace free, the other two implementations are not, leading to the possibility of infinite loops. During testing, this resulted in multiple retries for UnsynchronizedState in particular.

2.3. Performance Results

Ideal Use Case

Figure 1 characterizes the typical use case described by GDI. By design, this is also the ideal case for my BetterSafe implementation of State. As seen in Figure 1, Synchronized becomes very slow in the use case. Because each thread is fighting for the same structure, blocking consumes most of the programs time. This is the nature of the course grain locking that synchronized methods utilize. BetterSafe on the other hand, has linear shape in Figure 1, similar to the other implementations. Unlike the other implementations, BetterSafe is data race free, ensuring a correct result. The only difference in performance to the faster methods is the overhead associated with locking, which is amortized as transitionas and number of threas increase.

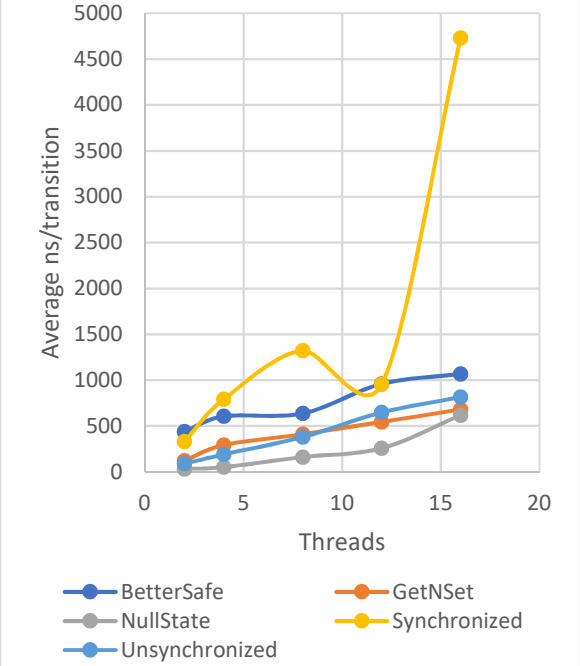

Figure 1: 10,000,000 transitions per thread with an array size of 2500 bytes. Maxval of 100. Notice the steep shape of SynchronizedState in comparison to the other implementations.

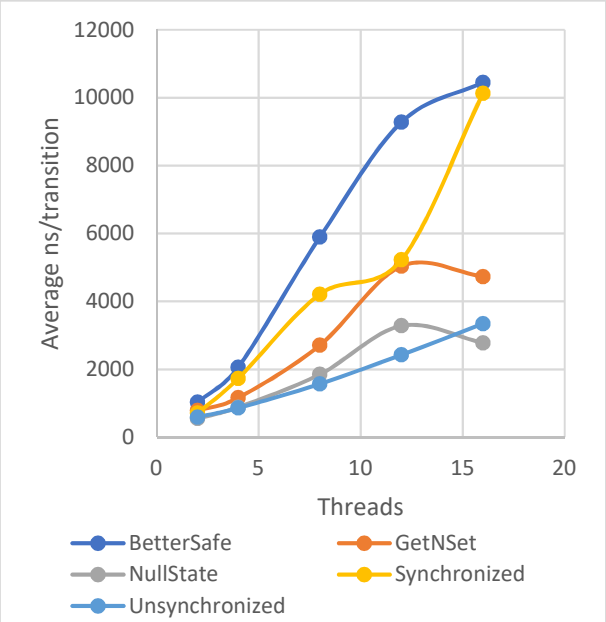

Figure 2: 100,000 transitions per thread with an array size of 500 bytes. Maxval of 100. Notice the trend of both Synchronized and BetterSafe. As Synchronized becomes steeper, BetterSafe becomes shallow.

Middle Case

Figure 2 presents a middle of the road use case. This case is not that of GDI, but it describes a smaller multi-threaded environment. As indicated by the graph, BetterSafe does not perform as well. SynchronizedState outperforms it for all the variations of the number of threads. Due to the design of BetterSafe, the performance is reduced by roughly a factor of 10 due to the cost of locking and the increased collisions between thread accesses to the byte array.

Both GetNSet and Unsynchronized maintain a roughly linear shape and are much fast than both BetterSafe and Synchronized, indicating the benefits of the atomic implementation of GetNSet.

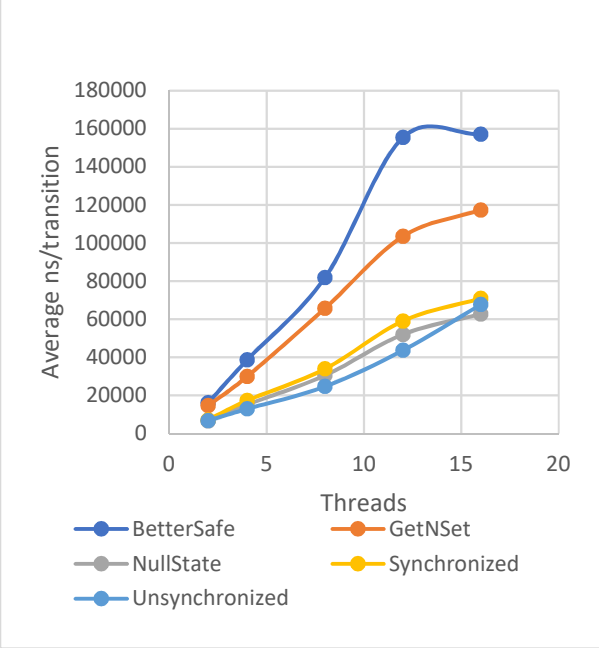

Figure 4: 1000 transitions with an array size of 25. Maxval of 30. Notice the lacking performance of both BetterSafe and GetNSet compared to the methods.

Worst Case

Figure 3 displays the downside of the BetterSafe implementation. Despite the design of BetterSafe, the rate of collision is too high with a small array. Thus, each thread is liable to block twice while locking due to contention on the locks. By comparison, Synchronized has similar performance to Unsynchronized and NullState, incurring very little cost with syncrhonized methods.

Interestingly, GetNSet also performs poorly in this case. This is due to the atomic accesses preventing the compiler from performing optimizations/reorderings of code. This effect is noticed due to the small size of the array, causing atomic accesses to interact with the ordering of each other.

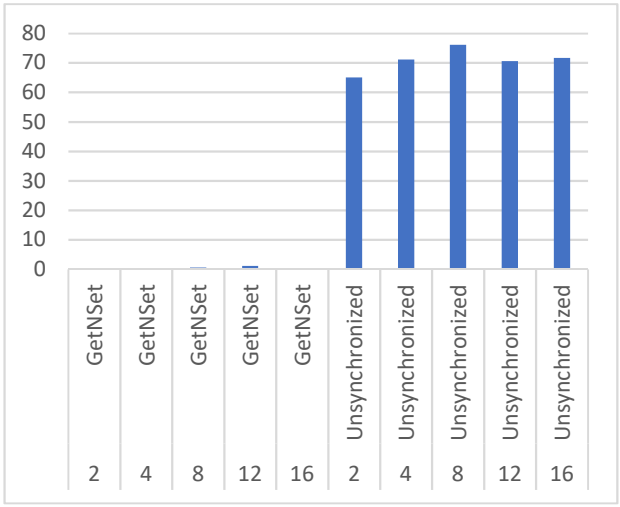

Figure 3: Percentage of tests in which a sum mismatch occurred across all tests performed. Horizontal axis is number of threads and method used.

2.3. Reliability Results

Across all tests performed, BetterSafe and SynchronizedState never incurred sum mismatches. Given the implementation and the testing results, BetterSafe is therefore 100% reliable.
Figure 4 indicates the reliability results of the implementations containing data races. As seen, Unsynchronized is very unreliable. During 70.89% of tests performed, Unsynchronized encountered a mismatch, compared to the 0.33% of GetNSet.

3. Conclusion

Ginormous Data Inc. has two possible implementations to consider. As shown, BetterSafe's implementation is 100% reliable and has impressive performance under the circumstances typical to GDI's use case. Compared to Synchronized, the locking method I implemented excels with a large dataset and high number of threads. Thus, BetterSafe is a good option for GDI
However, Ginormous Data Inc.'s ideal implementation does not require 100% reliability. Therefore, GetNSet is the ideal implementation for them, given that it consistently outperformed BetterSafe and it is almost always reliable.