Maintaining Sequential Consistency in Java: An Atomic Approach

Andres Martinez Paz
UCLA CS131

Abstract

In accordance to the Java Memory Model (JMM), a program is defined as data-race free if no two conflicting accesses take place without synchronization. In Java, two sequential accesses are deemed conflicting if both are coming from separate threads, and if the targeted memory location is not declared as volatile. [1] In order to maintain thread safety, one may use the synchronized approach, which will implement a system of Lock-and-release accesses to memory locations, ensuring sequential consistency for a program. This approach, while safe, may suffer in terms of performance due to the access control strategy used. We argue that utilizing an atomic approach maintains the necessary sequential consistency, while boosting program performance across platforms.

1 Introduction

The Java Memory Model introduces different memory modes for handling concurrency safely. One big concern when dealing with multi-threaded applications is to have data-race free implementations that maintain sequential consistency. When looking at Java's synchronized approach, while the integrity of the memory being accessed is very much guaranteed, there is also an introduction of data-races. this is due to the nature of the implementation which deals with lock-and-release approaches, similar to an Operating System's resource access control. This approach blocks access to resources when a thread enters a synchronized section. These resources are not released until the thread is done with the operation. The introduction of data-races is apparent, as multiple threads try gain access to a same synchronized segment. In this case, we can see that if too many threads seek access to the same synchronized segment, we lose the benefit of multi-threading. An alternative approach to this is that of atomic data structures. Java's own implementations of atomic structures introduce method calls for memory access in atomic operations—that is, operations which are done in a single uninterruptible step, undisturbed by other threads. It is clear to see that this atomic structures are data-race free due to the nature of the call which allows the data structure to remain undisturbed during reads and writes. We take advantage of Java's atomic package in order to explore the performance benefits of using an atomic DRF approach which maintains sequential consistency, versus the more common synchronized approach which ensures consistency at the cost of data-races. [3]

2 Methodology

In order to measure the performance benefits between both memory approaches, we utilize a testing algorithm in which an array of "long" values is subjected to a single "swap" action. We define this swap action as a method accepting two index parameters, i and j, wherein the index i of the array is decremented, and the index j is incremented. The synchronized version of the algorithm simply uses the **synchronized** keyword in the swap method signature. Following the standard of the Java Memory Model, this locks access to the methods resources and only releases them when the method retires. [2] We explore two alternative methods, an **unsynchronized** approach where we simply omit the use of the synchronize keyword—this introduces risks of sequential inconsistency—and an **atomic** approach. We argue that this atomic approach not only provides boost in performance compared to the synchronized algorithm, but it manages to keep the integral safety of the array.

An important aspect to analyze is the performance of each algorithm across different platforms. This should give us a better idea as to the concrete advantages of utilizing one algorithm over the other. We present the CPU specs as follows:

CPU1 Intel Xeon Silver 4116 @ 2.1 GHz. This processor has 12 cores, 16.5MB L3 cache, with DDR4-2400 with 6 memory channels.

CPU2 Intel Xeon E5620 @2.4GHz.This processor has 4 cores, 12MB L3 cache, with DDR3-800 with 3 memory channels.

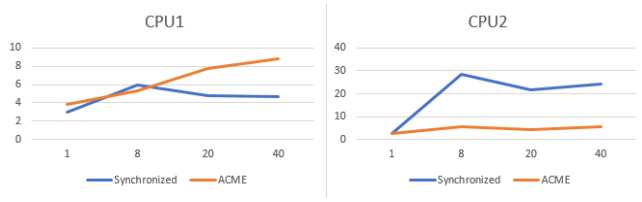

Figure 1: This figure shows the real time performance of both the synchronized algorithm and the AcmeSafe algorithm as a function of thread count across both CPUs used.

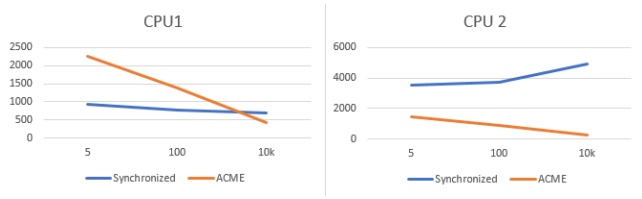

Figure 2: This figure shows the average swap time of both the synchronized algorithm and the AcmeSafe algorithm as a function of thread count across both CPUs used.

We present the methods used in both approaches. We note that **AtomicLongArray's** getAndDecrement and getAndIncrement are both atomic in nature, which ensures a data-race free implementation.

```
private long[] value;
public synchronized void swap(int i, int j)
{
      value[i]--;
      value[j]++;
}

private AtomicLongArray value;
public void swap(int i, int j)
{
     value.getAndDecrement(i);
     value.getAndIncrement(j);
}
```

3 Results

In order to analyze the main aspects for both algorithms, we focus on two criteria, the real execution time, and the average swap time. These two metrics give a clear view of the performance difference between the approaches. We collect these metrics across several trials of varying parameters. To coalesce the results into a meaningful metric, we ran trials while changing the three main parameters exhaustively. The first trial is ran on both algorithms with a single thread, an array of size 5, and 100 million swap transactions, and end with a runs of 40 threads, an array of size 10 thousand, and 1 billion swap transactions. When collecting results, we analyze

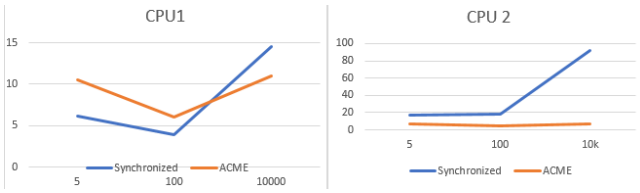

Figure 3: This figure shows the the real time performance of both Synchronized and AcmeSafe algorithms as functions of state array length across both CPUs used.

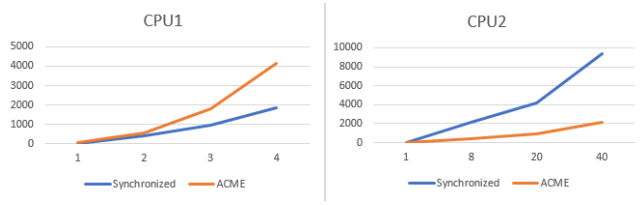

Figure 4: This figure shows the average swap time of both the synchronized and the AcmeSafe algorithms as functions of the state array length across both CPUs used.

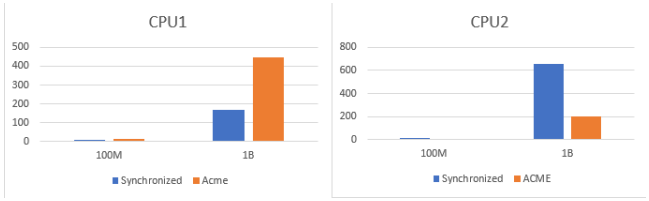

Figure 5: This figure shows the real time performance of both the synchronized and the AcmeSafe algorithms as a function of swap transitions across both CPUs used.

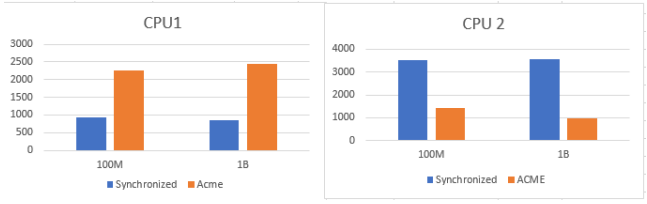

Figure 6: This figure shows the average swap time of both the synchronized and the AcmeSafe algorithms as a function of swap transitions across both CPUs used.

the average values of the metrics with a single fixed parameter, across the trials where the other two parameters changed. We end up with 6 comparisons across CPU. The most interesting find in this case was that of contradicting results across CPUs for several of the runs. Consider figure 5 and figure 6, in this case the performance of both algorithms with respect to the number of swap transactions are completely opposite. This could be a result of the memory architecture of each system. Since CPU has 12 cores and a memory architecture with 6 memory channels, the impact of the threads racing and locking resources is not too apparent, while the atomic algorithm will see no benefit in this case. It is clear across the results that the atomic algorithm offers much better performance in CPU2.

4 Conclusion

We can see from our results that the atomic approach is preferred when dealing with a more limited machine. We see the biggest performance boosts with the 4-core CPU vs the 12-core CPU. While the results do not show a solid improvement, we show a reliable alternative free of data-race conditions which can be easily introduced to multi-threaded applications.

4.1 Afterthoughts

It should be noted that very little was mentioned of the **unsynchronized** approach. This is not due to a lack of unsynchronized state runs, but rather, since the unsynchronized implementation completely breaks sequential consistency, it was deemed irrelevant when discussing viable alternatives to synchronized locks. In terms of performance, the unsynchronized implementation runs much faster across most, if not all, of the parametric runs, while returning completely unsuccessful results.

References

[1] Doug Lea. *Concurrent Programming in Java, Design Principles and Patterns*. Addison-Wesley Professional, 2.00 edition, 1999.

[2] Doug Lea. Using jdk 9 memory order modes. 2018.

[3] Andreas Lochbihler. Making the java memory model safe. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 35, 12 2013.