# Homework 3: Java shared memory performance races Report

Nilesh Gupta, *UCLA*

## Abstract

The goal of this homework was to experiment with the performance of a multithreaded Java application. The most effective way to do this is to tamper with the guidelines enforced by the Java memory model (JMM). The JMM is responsible for ensuring that a Java program runs as fast as possible yet still retains stability and safety. By ignoring the JMM, the program can run faster than it would normally, at the expense of its reliability. In some scenarios involving vast amounts of data, the resulting errors are outweighed by the pure processing speed, especially in competitive industries where programs are intended as heuristics rather than

## 1. Background

The main two types of data races encountered in this homework are:

a) read-write conflicts

b) write-write conflicts

The first type occurs when a thread reads a location directly after a different thread writes to that location. This causes various issues in the logic of the program.

The second type usually involves more out of bounds errors. Two threads may read an element currently within the max or min value, then simultaneously write to the element, causing its value to go outside of its possible range.

The way to prevent these data races is to utilize the Java keywords `synchronize, volatile, and final`, and make use of certain threading concepts, such as locks. These tools allow the programmer greater supervision over the threading behavior of the program, and make it possible to control its performance and predictability.

## 2. Classes

The five classes outlined in the Abstract each represent different coordinates on the plane enclosed by the two dimensions, speed and safety.

### 2.1. Synchronized

As the control class, Synchronized is the most basic implementation of the program. This class uses the `synchronize` keyword in the swap function header to ensure insulated modifications to the byte array. While this approach is extremely secure, the use of synchronized creates a bottleneck in the code. – each thread must wait its turn to execute the swap function.

The order in which the threads are permitted may not necessarily be a queue; the compiler is free to optimize this. Nonetheless, the fact that the swap function cannot be run serially greatly hurts performance. Thus, the Synchronized class, while safe, leaves much room for increases in speed.

### 2.2. Unsynchronized

The next class, Unsynchronized is identical to Synchronized except for the omission of the `synchronize` keyword. This leaves the class susceptible to both the (a) and (b) type data races prevented in the lock mechanism of synchronize.

As expected, this roughly equates to faster speeds, though the difference is marginal at best, and in many trials, reversed. In fact, out of the five most recent runs, the Unsynchronized class outperformed Synchronized only three times. In four out of five of these runs, the Unsynchronized class failed the sum test, meaning that the program experienced significant data races.

While the trend seems to improve slightly as the number of threads increases, it appears that the Unsynchronized class is more dangerous than the Synchronized class with approximately the same performance – a puzzling result since the elimination of the bottleneck should greatly increase execution speed. Perhaps this is caused by some internal failsafe against data races in the JMM. When the code fails past a certain point, perhaps the JMM steps in and inserts cautionary impediments to the program to avoid further damage.

### 2.1. GetNSet

The GetNSet class is designed to be a step up from the Unsynchronized class. The way it does this is it declares value, the byte array, as an AtomicIntegerArray, which allows for atomic operations; essentially, this ensures that any set call to the array must occur before any get call thus avoiding type (a), read-write conflicts.

The thing this does not account for is type (b), write-write conflicts in which two threads alter an element simultaneously. The question of the final value of the element after these two sets is an example of the undefined behavior possible within this implementation.

Still, though GetNSet is somewhat slower than Unsynchronized, it produces less sum mismatches. This increase in safety but comparability in speed means that it is successfully between Unsynchronized and Synchronized.

### 2.1. BetterSafe

One of the two main experimental classes, BetterSafe is constructed as a faster version of the Synchronized class. As such, the class avoids the `synchronize` keyword in favor of a reentrant lock in the swap function.

The lock performs faster than `synchronize` in high contention scenarios because of its greater flexibility in regards to lock interruptions and thread scheduling, and possibly its lighter internal representation.

The lock also maintains the sequential execution of swap possible with synchronize, since no two threads can make simultaneous accesses to any location in a deadlock. Thus, the program retains 100% reliability.

### 2.1. BetterSorry

BetterSorry represents the furthest coordinate along the performance axis of the plane, and the second furthest in safety. It features a similar implementation as GetNSet; however, instead of implementing value as a pointer to an AtomicIntegerArray, it declares value an array of AtomicIntegers.

The difference is that the second method does not lock the entire array between a set call and subsequent get; rather, each individual element is locked which leaves threads to perform work on multiple elements concurrently.

However, this means that BetterSorry is prone to the same errors as GetNSet – namely, write-write conflicts. Still, this is immensely better than the Unsynchronized class which safeguards against neither type (a) or (b) errors.

## 3. Results

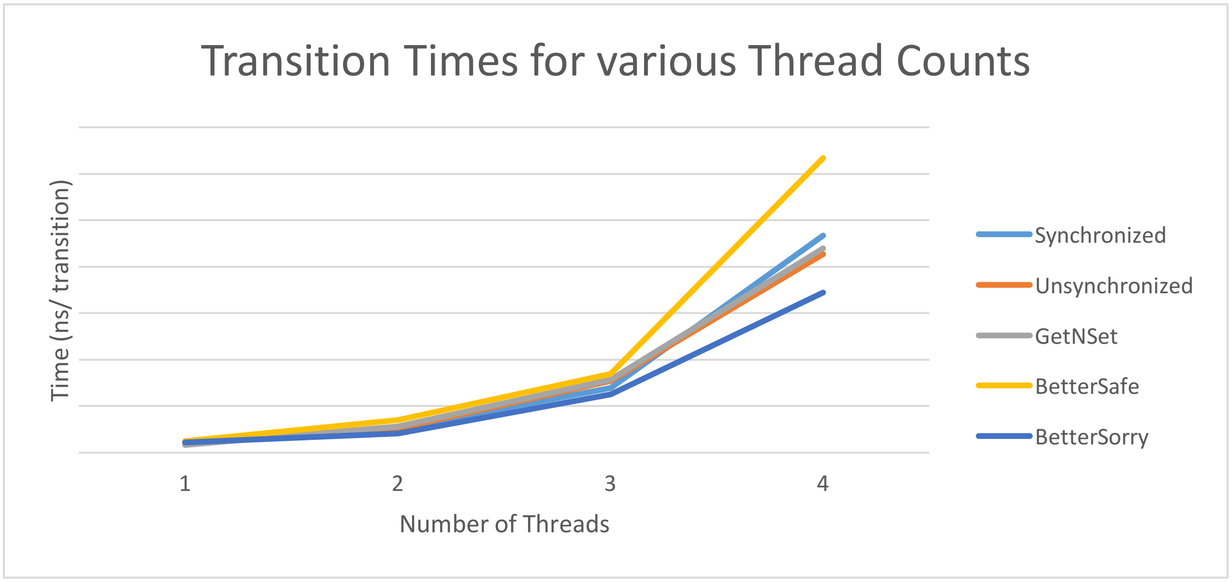
For the non-DRF classes, Unsynchronized, GetNSet, and BetterSorry, the most troubling issue in running the program the dead loops the programs would get caught in past a certain number of transitions. These were initiated when the elements of the array reached the edge of maxval, or passed it.

The command that most often derailed the less-than-reliable classes was something like:

```
java UnsafeMemory class 20 1000 5 5 0
0 0 0
```

One thing to note is when maxval is decreased, the number of dead loop timeouts increase for Unsynchronized which verifies the intuition that lowering the bound causes more type (b) data races.

From the chart below, the best choice for GDI applications would be the BetterSorry class since this maximizes performance while maintain a healthy level of reliability.


Transition Times for various Thread Counts
Time (ns/ transition)
Number of Threads
Synchronized
Unsynchronized
GetNSet
BetterSafe
BetterSorry