Synchronized state:
For the synchronized state the swap function is made synchronized. This creates a monitor acquire event while a thread enters a function and a monitor release when a thread is done with the executing the function. In short, no thread B can interrupt a thread A if thread A is executing the synchronized function, swap(int I, int j). 100% reliable but for large values of # of threads tends to get very slow.

Unsynchronized state:
For unsynchroized state, I just removed the synchronized keyword and kept the rest of the code the same. There are ample places for the data race condition to occur, one such occurrence being a thread being interrupted when value[i] and value[j] is being read. If the interrupting thread now updates the value of value[i] and value[j] then we have a data race. Not 100% reliable but very fast performance.

GetNSet State:
For this state implementation, I used the java.util.concurrent.Atomic library. All the read/write operations performed are atomic and the byte array was converted to an atomic integer array. This will always be DRF since all the portions which induce a DR are made atomic.

BetterSafe State:
For this I looked into ***CAS synchronization*** . I had a flag (AtomicBoolean type) called "locked" to indicate whether the data race inducing portion (the read and write of values into the byte array) of the code is locked or not. I have pasted a code snippet to illustrate the point.

```
public boolean swap(int i, int j) {

if (locked.equals(false)){//check the flag to see if the volatile resource is locked or not

    locked.compareAndSet(false, true); //LOCK this part of the code for a thread to perform read/write operation

    if (value[i] <= 0 || value[j] >= maxval) {
            locked.compareAndSet(true, false); //UNLOCK after READ
            return false;
            }
    value[i]--;
    value[j]++;
    locked.compareAndSet(true, false);// or UNLOCK after WRITE
}
    return true;
```

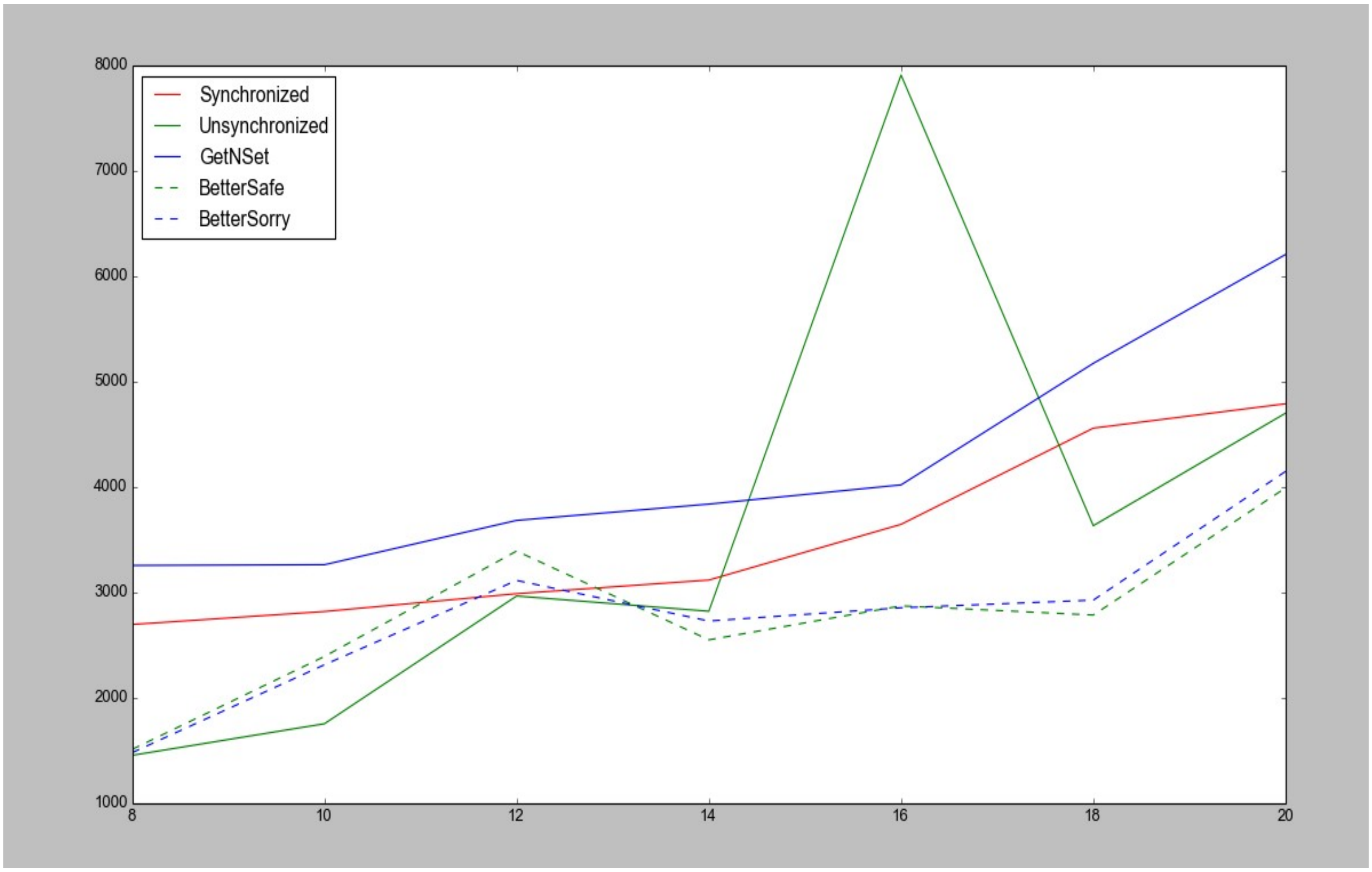
This implementation of state is also DRF since the atomic boolean flag, locked, acts as a lock on the read/write operations on value[i] and value[j]. If a thread is interrupted, it can only be interrupted after locking the volatile part hence making it inaccessible for the other threads. **This is very fast and DRF**.

I also considered other approaches like making the individual elements of the array atomic and then performing read/write operations on them, but even though that remains DRF, I found it to be slower than the current implementation. I also tried using **Reentrant Locks** but still found them slower than this approach.
This approach is 100% reliable and fast performance.

BetterSorry State:
For BetterSorry implementation of state, I mapped the read operations on to local variables, but the wrtie operations were done non-atomically. This also is as fast as BetterSafe (only occasionally being faster than the BetterSafe implementation). *BetterSorry has a data race condition if the values value[i] and value[j] change and a thread is interrupted after the old values of value[i] and value[j] are copied to the local variables.*

Performance Results:

# of threads* VS *ns /transition



As it can be seen from the graph above, Unsynchronized and BetterSorry are not DRF. BetterSafe has a much better performance than the other models which are DRF.
<u>BetterSafe and BetterSorry (considering the fact that GDI prefers speed to efficiency) seem to be better for GDIs sepcifications</u>

<u>On Reliability:</u>
Synchronized>=GetNSet>=BetterSafe>BetterSorry>UnSynchronized
<u>On Performance:</u>
UnSynchronized>BetterSorry>=BetterSafe>Synchronized>=GetNSet

Results were obtained on the following hardware:
4GB RAM