In order to obtain a measure of the performance of each State I used two scripts. By typing: ./calc_average.sh *NameOfState* (which in turn calls runTests.sh) on the command prompt I was able to use these scripts to test the performance of each state. I used 4 different thread numbers (varying from 1 to 32) and for each thread number I performed 5 different numbers of operations. Then I took the average of those times:

Table 1 shows the times for small numbers of operation (100-105):

State	Null	Synchronized	Unsynchronized	GetNSet	BetterSafe	BetterSorry
Time(ns)	4846.38	5106.36	5001.56	5456.22	5205.64	5458.75

Table 2 shows the times for larger numbers of operation (10^6-10^6+5):

State	Synchronized	BetterSafe	BetterSorry	Null
Time (ns)	5679.87	3963.39	4042.72	4030.19

A) One difficulty I faced measuring all the results is that for States containing many data races, they seem to never terminate as we increase the number of operations. This is because they fall into an infinite loop caused by the Swap.Test file for values less than 0 or greater than maxval (which occur due to data races). As a result the second table, consisting of times for larger numbers of operations, could not contain the Unsynchronized GetNset and BetterSafe state. The reason why I included both tables is because the first one is misleading. Table1 presents Synchronized state to be faster than BetterSafe and BetterSorry which is not true for large number of operations.

Table 3: Reliability

State	Null	Synchronized	Unsynchronized	GetNSet	BetterSafe	BetterSorry
Reliability (%)	100	100	Close to 0	Close to 0	100	Close to 100

Synchronized
This model is DRF because it used the synchronized methods of java and therefore allows only one thread to do a swap at a time. Nevertheless, this implementation even though it is very safe it is also very slow.

Unsynchronized
Essentially the same model as Synchronized with the 'synchronized' keyword deleted from the swap implementation. Therefore multiple threads can access the same function simultaneously and cause data races yielding unexpected results. This command will probably yield a mismatch:
java UnsafeMemory Unsynchronized 8 100 6 5 6 3 0 3

GetNSet
Uses the AtomicIntegerArray class but with the get() and set() methods. This fact makes this state very unreliable. For instance if a thread's execution is paused after the first get() without finishing the corresponding set(), then a data race would occur. This command will probably yield a mismatch:
java UnsafeMemory GetNSet 8 100 6 5 6 3 0 3

BetterSafe
BetterSafe state implementation has the same structure as GetNSet nevertheless it does not use separate get()-set() methods but instead makes use of the AtomicIntegerArray functions decrementAndGet and incrementAndGet. These functions are atomic and replace a get-set pair, which makes the functions more efficient as well as DRF (therefore 100% reliable)

BetterSorry
BetterSorry uses an array of AtomicIntegers instead of an AtomicIntegerArray. This allows multiple threads to be working simultaneously on the same array as long as they operate on different elements of the array. This makes BetterSorry faster than BetterSafe.

It is better than Unsynchronized State in the sense that it avoids many errors that Unsynchronized would fall into due to the fact that it used AtomicIntegers to prevent simultaneous access to the same array index.

I was able to generate a program that causes BetterSorry to have data races.
By typing on the command line:
./runTests BetterSorry
This runs BetterSorry for thread numbers varying from 1 to 32 and operations in the order of magnitude of 10^3. I observed that for 8 threads and 616 swap operations BetterSorry experienced a data race.

Best for GDI:
I would probably use the BetterSorry State since results don't need to be perfect but fast since we are dealing with large amounts of data.