Homework 4 Report

In this homework, we are dealing with two data race problems. One is that each thread may has local cache. Thread gets data from local cache but it may not be aware of changes in time. The other is race condition which I summarized below:

DRF	Thread 1	Thread 2	Value	Dara Race	Thread 1	Thread 2	Value
	Get v[i]		0		Get v[i]		0
	Increase v[i]		0			Get v[i]	0
	Set v[i]		1		Increase v[i]		0
		Get v[i]	1			Increase v[i]	0
		Increase v[i]	1		Set v[i]		1
		Set v[i]	2			Set v[i]	1

To solve local cache problem, the method is to get data from main memory directly. To archive DRF, the method is to make the “Get-Increase-Set” procedure atomic. To avoid deadlock situation, I always execute smaller index first.
For Unsynchronized module, I just remove synchronized keyword from Synchronized module. It is totally not DRF.
For GetNSet module, I convert value array to AtomicIntegerArray first, then use atomic Get and Set method to do the job. Although Get and Set is atomic respectively, but “Get-Increase-Set” is not, so it's not DRF, but it solve local cache problem.
For BetterSafe module, I use ReentrantLock to lock the whole “Get-Increase-Set” procedure. This time it's DRF and the performance is good.
For BetterSorry module, I simply use static and volatile keywords. I mark value array as static volatile so that each thread has to get data from main memory, and it solve local cache problem. But it's also not DRF.

To test the performance, I did following experiments. Parameters: 8 or 32 threads, 10,000,000 transactions, value1=[100,200,300,400,500], (to prevent all values reduce to 0) value2=[10,20,30,40,50,10,20,30,40,50,10,20,30,40,50]. The reason I choose 10,000,000 transactions is that I think the more transactions it takes, the harder it is disturbed by outer conditions.

Performance	NullState	UnSynchron	BetterSorry	GetNSet	BetterSafe	Synchron
8 10,000,000 value1	49ns	162ns	205ns	328ns	1104ns	1753ns
32 10,000,000 value1	231ns	737ns	826ns	1432ns	4660ns	7875ns
32 10,000,000 value2	296ns	699ns	866ns	1591ns	4607ns	6917ns

From the chart we can see that more threads slow down the transaction speed but the length of the array does not really affect the speed, sometimes it's even faster for longer array.
The reason that BetterSafe module is faster than Synchronized module is that ReentrantLock is a

low level implement and I only locked the “Get-Increase-Set” procedure.
The result running on SEASNET machine is different. GetNSet module took longer time than BetterSafe module. I didn't record the statistics because sometimes it hangs there and I'm pretty sure that it's not because of deadlock and I even got an email from SEASNET saying that I consume to much resource. My guess to this scenario is that there is something with SEASNET machine because it only has 3GB RAM, which may not be adequate for multi-threads access to AtomicIntegerArray.

To measure the reliability, I tried several ways. One is to use a synchronized block to check the sum of array after every swap was done, to see if there is a race condition or not, and update the sum value for next round. I actually finished this implementation(and comment out), and had the result. I think it does reflect the relation between different modules. But this is not a precise way because since “Get-Increase-Set” is not atomic, other threads may change the sum during checking. The other is to use cylicBarrier class to calculate the sum after every threads finish once swap operation. But this does not work because this will treat one thread making mistake with all threads making mistake as the same thing. Finally I came up with a tricky way to measure reliability that change value[i]-- to value[i]++, that is making the swap function only does plus operations. In this way, each race condition will cause the result value one less than the correct value (as shows in the chart at very beginning). So then I can calculate **accurate rate** by using current sum divides (2 * transactions+original sum). The result is followed, with same parameters as above.

Reliability	NullState	UnSynchron	BetterSorry	GetNSet	BetterSafe	Synchron
8 10,000,000 value1	100%	48.7%	52.3%	58.9%	100%	100%
32 10,000,000 value1	100%	37.5%	49.5%	48.1%	100%	100%
32 10,000,000 value2	100%	62.9%	73.9%	73.5%	100%	100%

From the chart we can see that the more threads we use, the less the accurate rate is, because more threads cause more conflict (race condition). But if there are more values in the array, the accurate rate increased, because the possibility that two threads select the same index decreased.

In conclusion:
Reliability
100% = NullState = SynchronizedState = BetterSafeState > BetterSorryState ≥ GetNSetState > UnsynchronizedState
Performance
NullState > UnsynchronizedState > BetterSorryState ≥ GetNSetState > BetterSafeState > SynchronizedState

So for my opinion for GDI's application, BetterSorry module is best choice. It has good reliability (even better if there are tons of data) and it is faster than GetNSet module.

Environment: (I didn't use SEAS server because it always fail when tried on 32 threads)
8 processors, 4 cores, Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz, 8GB Ram, 64 bit Ubuntu 12.04 with java version "1.7.0_51" .