1. See below for tests on Null and Synchronized models. Both have 100% reliability because avoid race conditions.
2. Partial output of /proc/cpuinfo:

processor : 15
vendor_id : GenuineIntel
cpu family : 6
model : 44
model name: Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
stepping : 2
microcode : 12
cpu MHz : 1596.000
cache size : 12288 KB
physical id : 0
siblings : 8
core id : 10
cpu cores : 4
apicid : 21
initial apicid: 21
fpu : yes
fpu_exception : yes
cpuid level : 11
wp : yes

Partial output of /proc/meminfo
MemTotal: 32865700 kB
MemFree: 13502872 kB
Buffers: 523384 kB
Cached: 14558324 kB
SwapCached: 24032 kB
Active: 6002636 kB
Inactive: 9404264 kB

2. BetterSafe is faster than Synchronized because it provides better control on obtaining locks though the ability to interrupt and timeout on waiting for a lock. The ReentrantLock maintains the same functionality of the synchronized keyword, but also has this additional functionality. It is still 100% reliable because it implements a lock around the critical section, which prevents any race conditions.
3. BetterSorry is faster than BetterSafe because it does not implement a lock. It is also more reliable than Unsynchronized because the get and set operations are done atomically with the getAndDecrement/getAndIncrement methods.
   Race condition that Unsynchronized is liable to but BetterSorry avoids:
   java UnsafeMemory Unsynchronized 8 1000000 6 5 6 3 0 3 -> Hangs
   java UnsafeMemory Unsynchronized 8 1000 6 5 6 3 0 3 -> Gives incorrect sum

   Give a race condition that BetterSorry still suffers from:
   After you check value[i] is in the proper range, it could still change before you increment. I

was unable to write a test program that causes this race condition to occur with high probability. This is because even if there is a potential race condition in the code, it won't occur every time the code is executed.

4. **Synchronized**:
   java UnsafeMemory Synchronized 4 1000000 6 5 6 3 0 3 : 1739.28 ns/transition
   java UnsafeMemory Synchronized 8 1000000 6 5 6 3 0 3 : 3868.81 ns/transition
   java UnsafeMemory Synchronized 16 1000000 6 5 6 3 0 3 : 5793.10 ns/transition
   java UnsafeMemory Synchronized 8 1000 6 5 6 3 0 3 : 36260.9 ns/transition
   **Null**: Same test case as Synchronized
   Threads average 270.696 ns/transition
   Threads average 1390.99 ns/transition
   Threads average 2948.00 ns/transition
   Threads average 35648.4 ns/transition
   **Unsynchronized**:
   java UnsafeMemory Unsynchronized 1 1000000 6 5 6 3 0 3 : 57.8772 ns/transition
   java UnsafeMemory Unsynchronized 8 1000 6 5 6 3 0 3 : 32678.3 ns/transition - sum mismatch
   java UnsafeMemory Unsynchronized 16 10000 6 5 6 3 0 3 : Hangs forever
   **GetNSet**:
   java UnsafeMemory GetNSet 8 1000 6 5 6 3 0 3: 53475.6 ns/transition - sum mismatch
   java UnsafeMemory GetNSet 16 1000000 6 5 6 3 0 3: Hangs forever
   **BetterSafe**:
   java UnsafeMemory BetterSafe 8 1000000 6 5 6 3 0 3 : 1669.47 ns/transition
   java UnsafeMemory BetterSafe 16 1000000 6 5 6 3 0 3 : 3405.17 ns/transition
   java UnsafeMemory BetterSafe 8 1000 6 5 6 3 0 3 : 69867.1 ns/transition
   **BetterSorry**: Same test case as BetterSafe
   Threads average 2463.69 ns/transition
   Threads average 4318.85 ns/transition
   Threads average 41888.1 ns/transition

Null, Synchronized, and BetterSafe are all DRF.
Unsynchronized is definitely not DRF, and consistently runs into race conditions. Here is an example test case: java UnsafeMemory Unsynchronized 8 1000 6 5 6 3 0 3.
GetNSet is not DRF: java UnsafeMemory GetNSet 8 1000 6 5 6 3 0 3. This command produces a sum mismatch.
BetterSorry is also not DRF, but it is very difficult to produce a test case that consistently causes it to fail (see above).
BetterSorry outperforms BetterSafe when there are fewer number of swaps occurring. However, when there are many swaps, BetterSafe performs better.

5. BetterSafe or BetterSorry would be my best choice for GDI's applications. Since GDI doesn't care if they get a few errors in their data as long as it is computed quickly, then they could go with BetterSorry. BetterSorry is optimal when there are fewer number of swaps occurring, so it also depends on their data set. If they are going to do a lot of swaps, then BetterSafe might be a better approach. I would personally go for reliability over speed.