CS636: Assignment 1

Paramveer Raol(170459) and Sumaiya Shaikh(19111414)

February 16, 2021

1 Task 1

1.1 Constant Sampling

The bar graph for Task 1.1 is shown in Figure 1

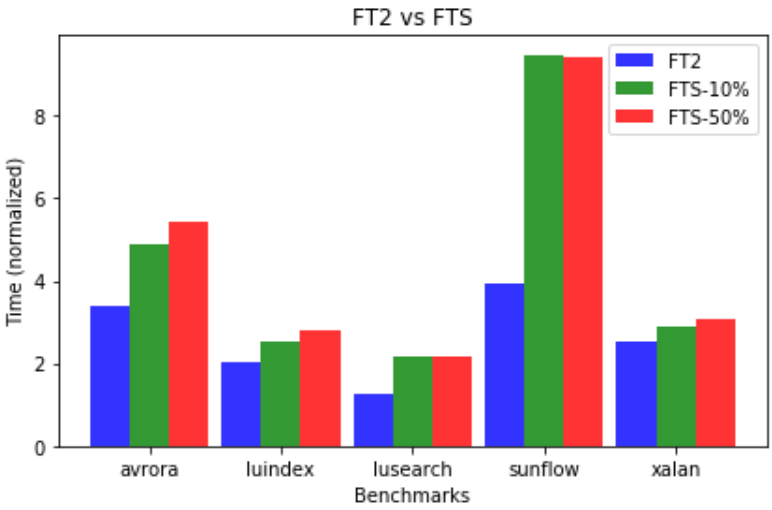

Figure 1: Performance comparison of FT2 with FTS

For constant sampling three strategies were experimented with. The strategies are as follows:

1. Generate a random number uniformly between 0-1 at every access and if the random number generated is less than (samplingrate/100) further checks and updates pertaining to the shadow variable were performed or else the updations and checks for that access of the shadow variable were not performed. This strategy works at global level but because of overhead that occurs during the random number generation the slowdown was close to 10x for various benchmarks hence this method was abandoned.

2. The second strategy is where a global boolean array of 100 size was constructed and then it was populated with "samplingrate" trues and 100-"samplingrate" falses. Then a counter was a global counter which was updated at every access. If the fieild of the array corresponding to the counter was true then the access was sampled (.ie the shadowvar/FTVarSate parameters checking and updating) or else the counter was merely incremented. This also had slowdowns close to 9x for the benchmarks and the reason for that is that whenever the global counter is incremented it has to be done in a synchronized way hence the other threads would be waiting during that time.

3. The third strategy that was employed is where for every thread there is an array of "samplingrate" trues and 100-"samplingrate" false(each permuted independently in the beginning), and there is a counter for every thread. The counter is incremented for the corresponding access's thread and the field of the array of the thread is read and if it is true then the access is sampled(same as in the previous 2 cases), or else not. Since this method does not require synchronization owning to the fact that the counter is of the thread hence there will not be more than 2 contentions for the thread's access. This is the method that is further used for the constant sampling part.

Note: The results presented below are obtained using the third strategy.

1.2 Race Pairs

	FT2	FT2 10	FT2S 50
avrora	5	5	5
luindex	1	0	2
lusearch	0	0	0
sunflow	5	3	7
xalan	38	8	40

Table 1: Data Race coverage of FT2

Brief description of the races that were obtained for each benchmark are as follows:

- **Avrora**: For the FT2 all the races occur in the Medium.java(present in the "avrora/sim/radio" directory) file majorly in the line 52 present in the access method, there are 4 races for the same file's line just for different offsets, the fifth race is between the 'end' method's 616 line and 'intersect' method's 533 line. For the FT2S with 10% sampling also the same races were detected as in the previous case same was the cse in FT2S with 50% sampling.

- **Luindex**: For the FT2 case only one data race is detected it is present in the file ConcurrentMergeScheduler.java (present in org/apache/lucene/index directory) between the line numbers 144 and 99. In FT2S with a sampling rate 10% no data race is detected. In FT2S with sampling rate 50%one more data race was detected it is present in the same file between lines 144 and 98.

- Luindex: For the FT2 case only one data race is detected it is present in the file ConcurrentMergeScheduler.java (present in org/apache/lucene/index directory) between the line numbers 144 and 99. In FT2S with a sampling rate 10% no data race is detected. In FT2S with sampling rate 50%one more data race was detected it is present in the same file between lines 144 and 98.

- **Lusearch**: No data race was detected for this benchmark in none of the tools.

- **Sunflow**: There are three data-races detected by the tool FT2 in the file Geometry.java(in the directory org/sunflow/core) between the lines 126 and 91, 93 and 119, 108 and 89 and two data-races are detected for the file NullAccelerator.java(in the directory org/sunflow/core/accel) between the lines 23 and 19,18 and 108. For FT2S with 10%sampling the data one data-race of Geometry.java(98 and 89), and both the races of NullAccelerator.java were missed but a new race at line 23 and 14 was captured. Then in the case of FT2S with 50%sampling, all the races of FT2 were captured along with those 2 new races one at Geometry.java's 24 and 13 line location, and the other at Nullaccelerator.java with line location 23 and 14 were discovered.

- **Xalan**: For FT2 total of 38 race pairs were found in various files OutputPropertiesFactory.java, CharInfo.java, ElemNumber.java, CharArrayWrapper.java, and XResources_en.java file the detailed locations are present in the repo. For the FT2S with 10% sampling rate, the number of data races detected was around 8 only and the and no races were found in the ElemNumber.java and file. In the case of FT2S with 50 %sampling rate, the 40 data-race pairs were detected in the same files as in the case of FT2.

1.3 Adaptive Sampling

The fundamental strategy that was employed in the case of adaptive sampling was that after "burstlen" number of accesses(could be at global or thread-local level) to the same memory location the "samplingrate" for that memory location(ie. FTVarState) was decreased(either linearly of exponentially) till the "samplingrate" reached "minsampling rate" then the sampling rate of the memory location was not altered. So in brief the various input parameters for the algorithm of adaptive sampling are as follows:

1. **Burst-length**: The number of accesses to that memory location after which the sampling rate would be altered. This access could be at the global level or thread level. For eg A has a burst length of 10 at the global level then after very 10 accesses of A at the global level the sampling rate of A would be altered, but if the level was at thread level in that cases the after 10 accesses of A in thread x the sampling rate of A in thread x would be changed irrespective of other threads.

2. **Decrement rate**: The rate at which the sampling rate fall after a burst-length number of accesses. This could be linear meaning after every burstlength number of accesses the new sampling rate of the memory location is "samplingrate - decrementrate". Whereas in the case of exponential decrement the new sampling rate would be "samplingrate/decrementrate".

3. **Minimum sampling rate**: The minimum rate beyond which the sampling rate cannot fall.

Note: The sampling rates were administered using the same randomly permuted array strategy as used in the constant sampling part. The initial sampling rate of all the memory locations is 100% and it then decreases progressively. The performance of the adaptive sampling algorithms was comparable to those with constant sampling(in case of thread-level sampling which was then used in accordance with the remarks of LiteRace paper) and in the case, data-race pairs that a number of data-race pairs that were detected are for the following parameter(not all the parameters are mentioned only the ones that gave good results are mentioned over here) and benchmarks are as follows:

- (100,5,10) (Linear) : avrora=5,luindex=2,lusearch=0,sunflow=7,xalan=40
- (100,10,10) (Linear) : avrora=5,luindex=1,lusearch=0,sunflow=5,xalan=38
- (200,2,20) (Exponential) : avrora=5,luindex=1,lusearch=0,sunflow=5,xalan=38
- (200,2,8)(Exponential) : avrora=5,luindex=1,lusearch=0,sunflow=5,xalan=38

Entries are the form of (burst-length,decrement-rate,min-samplingrate).

2 Task 2

Here are brief ideas for the algorithm for this task:

1. Take the possible pairs A-B as input and for the first accesses of A, first of all, set the execution flag of B to false and then sleep in an exponential backoff fashion till B's execution flag becomes true(ie. B has executed) then sample A.
2. The exponential backoff ensures that the least amounts of sleep occur and once the total time of sleep for A exceeds the max-time out A is sampled anyway.
3. Then for the next 'x' continuous accesses of A the max time out is increased linearly for the access since in previous cases the perturbation may not have been enough.
4. After 'x' such accesses in the future if A executes again then 'y' random accesses are decided at which the perturbation is introduced with the same time-out as in the case of 1.
5. After x+y accesses:
   (a) There is no time-out for A for any accesses in the future however it is sampled always(same with B).
   (b) If the A-B race pair is detected that is neither A nor B will be sampled ever again.
6. Locations that are not in race pairs are never sampled.

Both 5a and 5b have been used in the experiments they could be produced by simple procedure mentioned in the readme of the repo. So basically the input parameters for the algorithm are:

- Max-time out: Time maximum time a location of type A can go to sleep(will increase for continious sleeps till the max-continious sleep).
- Max-continuous sleeps: Maximum number of continuous sleeps from the first accesses where perturbation will take place.
- Max number of sleeps: Total number of sleeps where "Max-continuous sleeps" are always perturbed from the beginning and the rest of the "Max number of sleep"-"Max-continuous sleeps" are not sampled.
- Race-pairs: possible race pairs that must be perturbed to detect races.

Pseudo code for the algorithm is as follows:

```
if(location is present in the race pair)
    if(location of type B)
        execution[B] = true
    else
         execution[B] = false
            if(# access of A < x)
                 exponential sleep till either:
                 1.B executes
                 2.or sleep time >= (# access of A)*(maxTimeOut)
            else if(# access of A < y and (ramdom coin toss))
                 exponential sleep till either:
                 1.B executes
                 2.or sleep time >= (# access of A)*(maxTimeOut)
        sample location(can be either A or B)
do not sample
```

Note: The logic behind setting B's flag to false in the beginning is that suppose A occurs multiple time in that case suppose the execution order is B→A→A hence in the case of the second execution of A clearly B's execution hasn't occurred after the first execution and even if A sleep in its second execution there is a chance that B may execute again and if not still no harm is done as the A will pass any way after time out and it will be useful in cases like B→A→A→B as it may get converted to B→A→B→A thus ensuring that there is a chance for B to execute.

FT2 race pairs were passed on to the FT2SS and the results are as follows:

1. Avrora: all the races in FT2 were detected no new race detected.
2. Luindex: all the races in FT2 were detected no new race detected.

3. Luserach: tried giving random races but no new race detected.
4. Sunflow: all the races in FT2 were detected no new race detected.
5. Xalan: all the races in FT2 were detected no new race detected.