Assignment 1

Aniket Sanghi (170110)
Sarthak Singhal (170635)

16th February 2021

We pledge on our honor that we have not given or received any unauthorized assistance.

System and Architecture Information

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz
Stepping:              3
CPU MHz:               800.005
CPU max MHz:           3200.0000
CPU min MHz:           800.0000
BogoMIPS:              4599.93
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
```

1 Task 1

1.1 How To Run

Provide the sampling rate via a command line parameter **-samplingrate** and the sampling scheme via a command line parameter **-samplingscheme** when using the **-tool FT2S**. Example usage:

```
./TEST -tool=FT2S -samplingrate=<> -samplingscheme=<> -array=FINE -field=FINE \
-noTidGC -availableProcessors=4 -benchmark=1 -warmup=0 RRBench
```

The sampling schemes supported by FT2S are: count, count_global, adaptive and adaptive_global. All these schemes are described in a later section.

1.2 Performance Bar Graphs

In the basic sampling algorithm($samplingscheme = count$), we kept per-thread counters and sampled r% of memory accesses randomly where $r = sampling\,rate$. The execution times(in secs) of the tools N, FT2, FT2S with 50% and 10% sampling rate for the desired 5 benchmarks are given below:

	N	FT2	FT2S 50%	FT2S 10%
avrora	8.82	40.97	32.17	26.06
luindex	6.56	17.45	21.48	18.73
lusearch	22.46	33.81	50.66	51.76
sunflow	29.49	185.08	153.50	142.44
xalan	14.57	30.82	51.20	45.56

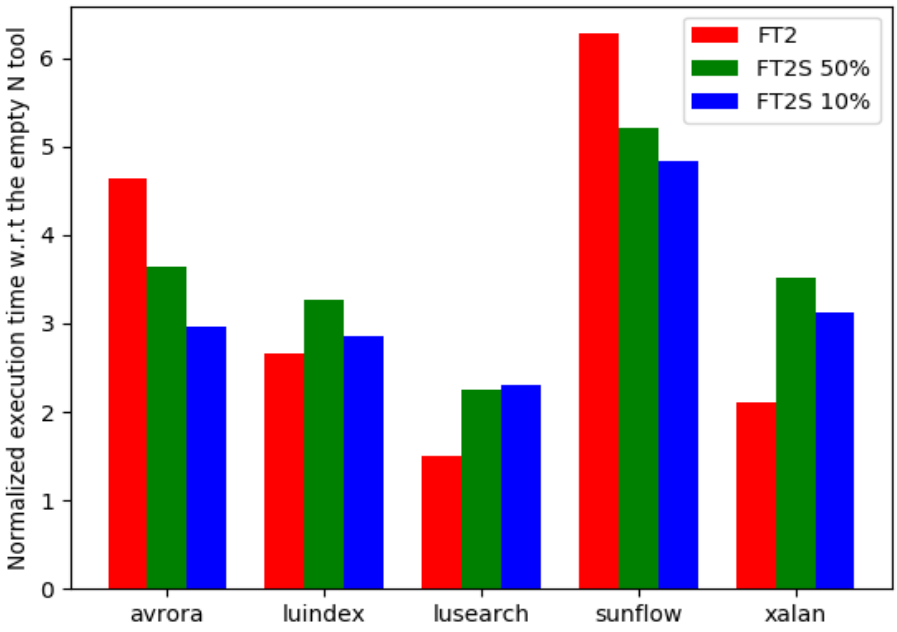

Figure 1: Performance of FT2 and FT2S(at 50% and 10% sampling rates) on the given benchmarks

The reason for the degradation in performance on the benchmarks luindex, lusearch and xalan by incorporating the sampling algorithm in FastTrack can be the overhead caused due to sampling algorithm on every memory access.

1.3 Data Race Coverage

The number of unique data races detected by running the tools FT2 and FT2S (at 50% and 10% sampling rates) over all the given benchmarks are given below:

	FT2	**FT2S 50%**	**FT2S 10%**
avrora	5	5	5
luindex	1	2	0
lusearch	0	0	0
sunflow	5	7	4
xalan	38	39	23

Table 1: Impact of sampling on the data race coverage

The unique data race(separated using @) detected in luindex using FT2 is:

```
org/apache/lucene/index/ConcurrentMergeScheduler.__$rr_message__$rr__Original_\
(Ljava/lang/String;)V(ConcurrentMergeScheduler.java):99:8
@
org/apache/lucene/index/ConcurrentMergeScheduler.__$rr_merge__$rr__Original_\
(Lorg/apache/lucene/index/IndexWriter;)V(ConcurrentMergeScheduler.java):144:2
```

Let the above race pair be represented as $B \Leftrightarrow C$.
While, the data races(separated using @) detected in luindex using FT2S with sampling = 50% are:

```
org/apache/lucene/index/ConcurrentMergeScheduler.__$rr_message__$rr__Original_\
(Ljava/lang/String;)V(ConcurrentMergeScheduler.java):99:8
@
org/apache/lucene/index/ConcurrentMergeScheduler.__$rr_merge__$rr__Original_\
(Lorg/apache/lucene/index/IndexWriter;)V(ConcurrentMergeScheduler.java):144:2
```

```
org/apache/lucene/index/ConcurrentMergeScheduler.__$rr_message__$rr__Original_\
(Ljava/lang/String;)V(ConcurrentMergeScheduler.java):98:1
@
org/apache/lucene/index/ConcurrentMergeScheduler.__$rr_merge__$rr__Original_\
(Lorg/apache/lucene/index/IndexWriter;)V(ConcurrentMergeScheduler.java):144:2
```

The first race pair above is $B \Leftrightarrow C$. Let the second one be $A \Leftrightarrow C$. We can observe that A and B belong to same class and method. They differ just by a single line where A comes before B w.r.t the source locations. While using FT2, A will occur before B, so only $B \Leftrightarrow C$ race will get detected if B occurs before C. While in the sampling version, it is possible that B doesn't get sampled, due to which C will happen after A instead of happening after B. Due to this, there is an extra race detected in FT2S with 50% sampling. Also, the increase in the number of data races detected in case of FT2S with 50% sampling w.r.t to FT2 can also be reasoned similarly.
There is a decrease in number of data races detected with FT2S 10% sampling. This happens because we are sampling relatively fewer memory accesses which is a reason of missed data races.
The complete details of the unique data races can be found in the git repository under the *output* directory. The files with unique race pairs are named as $\langle tool \rangle$_$\langle benchmark \rangle$_unique_racepairs.txt.

1.4 Sampling Algorithms

1.4.1 Global Sampling

- A 100-length randomly shuffled boolean array with r of the entries as true and rest as false is used for performing a sampling-rate of **r%**
- A synchronized global counter is used to sample accesses globally using the above boolean array
- Following command line parameter is used to execute it - **samplingscheme = count_global**

1.4.2 Thread Local Sampling [Default]

- A boolean array like the above one is created and thead-local counters are used to track accesses frequency.
- Per-thread sampling ensures more uniform sampling over threads and is expected to have more data-race coverage (*LiteRace* also uses per-thread adaptive sampling)
- Following command line parameter is used to execute it - **samplingscheme = count**

1.4.3 Global Adaptive Sampling

- Sampling frequency is tracked globally and handled with synchronization.
- The Sampling frequency is halved after every 4 accesses to a particular variable.
- Following command line parameter is used to execute it - **samplingscheme = adaptive_global**

1.4.4 Local Adaptive Sampling [Default]

- Sampling frequency is tracked locally per-thread and adapts to frequency of access per-thread
- The Sampling frequency is halved after every 4 accesses to a particular variable.
- Sampling per-thread ensures uniformity. Like if a particular thread accesses a variable huge number of times, then the samplingRate for this thread will be adapted to lower value but not for other threads. (*LiteRace* also uses per-thread adaptive sampling)
- Following command line parameter is used to execute it - **samplingscheme = adaptive**

1.5 Observations in adaptive sampling results

- The unique data races detected for all the benchmarks except luindex were exactly same as FT2. Race-detection in luindex was unsuccessful at the current adaptive sampling rate, it gets detected with more light-weight adaptive strategy like halving after every 8 accesses. Our decision was a trade-off between time and race coverage. The execution time was almost similar to the execution times we got from FT2S at sampling rate = 50%.

---

2 Task 2

2.1 How To Run

Provide the name of the file containing the list of race-pairs via a command line parameter **-racepair** when using the **-tool FT2SS**. Example usage:

```
./TEST -tool=FT2SS -racepair="racepairs.txt" -array=FINE -field=FINE -noTidGC \
-availableProcessors=4 -benchmark=1 -warmup=0 RRBench
```

The input file is assumed to contain the list of data-race pairs in the following format (Race pairs separated by @)

```
<SourceLocation.toString()>@<SourceLocation.toString()>
```

2.2 Highlights from RaceMob

- It generates all possible race pairs using a static data-race checker and then validate each of the race-pair using dynamic data race detection. Each system/user validates only 1 data-race pair.
- Since there can be scenarios where for a particular dynamic-interleaving, the race is undetected, so to avoid it, it uses Schedule Steering to perturb the execution to promote other possible interleaving as well.

2.3 Heuristic

1. Update vector clocks and other FT instrumentation for only those sourceLocations that are given in the race-pairs list
2. For a given race pair $A \leftrightarrow B$
   (a) Until a max frequency of A is reached or the data race is detected for the pair do the following
      i. (Start hearing if B is accessed by any other thread)
      ii. Until B is accessed or Timeout reached
         A. Make the thread of A sleep adaptively doubling the time in each sleep cycle
      iii. (Stop listening if B is accessed by any other thread)
      iv. Continue with the execution of A
3. For every B check if corresponding A is listening, Tell A that it was accessed.
4. For every data race, if the race detected was some $A \leftrightarrow B$ we were checking for, mark it as accessed.

2.4 Implementation Details

- We adaptively doubled the pause time for **A** starting from 100ms to 3.2s
- We set the timeout to be 4s as it appeared reasonable based on executions
- We tried perturbing about 5 times per race pair, It appeared of not much use to do more attempts

2.5 Results

- FT2SS found all the data race pairs in the given 5 benchmarks that were detected using FastTrack.
- In the sunflow benchmark, the schedule steering and perturbation was successful. In one of the given race pairs $A \Leftrightarrow B$, the execution was perturbed and B executed before A while A was sleeping.
- We tested on only race-pairs given by FT on the corresponding benchmarks, hence the chance of un-detection was considerably low.