Bidya Sarkar(18111011)
Jeevan Kumar(18111041)

Assignment 2

CS636
February 28, 2019

Road Runner API

- **Road Runner Data structures** : Road Runner uses several auxiliary data structures in implementation of each tool. A brief overview of those data structures as follows.
  - **Shadow Thread** : Road Runner maintains a shadow thread object for each thread object.
  - **Shadow Lock** : Road Runner maintains a shadow lock object for each lock object.
  - **Shadow Location** : Road Runner maintains a shadow location for each memory location.
  - To associate the tool specific information with each thread and lock Road Runner includes a **Decoration** class which provides map from keys to values.
  - Both Shadow Thread and Shadow Lock classes extends Decoratable and provide a generic makeDecoration method whereby tools can add Decorations for any desired type T to those structures.
- **Event** :
  - Each event in Road Runner extends Event super class which contains shadow thread object for the thread which performs that event.
  - To avoid allocation overhead, each type of event is created once for each target thread, and then re-used as that thread performs additional operation of that type. Thus, the thread field is final

```
public class Event  {
  protected final ShadowThread thread;
  public Event(ShadowThread td) {
    this.thread = td;
  }
}
```

- **Tool Class** :
  - Every tool (HB , FT2 ...etc) in Road Runner is defined by extending Tool class. Tool class contains event handlers for thread creation, lock acquiring, lock releasing and accessing non volatile memory.
  - Each event handler is called whenever one of the threads of the target program performs the corresponding kind of operation.
  - Return a fresh variable state for the tool for the location being accessed. Will be called only once per location per tool.

```
abstract class Tool {
  void create(NewThreadEvent e) {}
  void acquire(AcquireEvent e) {}
  void release(ReleaseEvent e) {}
  void access(AccessEvent e) {}
  abstract ShadowVar makeShadowVar(AccessEvent e);
}
```

- **Fast Paths** : Road Runner provides "Accss Fast Path" idiom to enable the direct inlining of the fast paths into target code. Full access events are only generated when the analysis falls of the fast path.

Happens Before (HB) :

- **create() :** When a thread is first created, create() method is called which creates the vector clock for that thread.
- Vector Clock is a class which contains an integer array of epochs and associated methods to support vector clock operations.

```
public void create(NewThreadEvent e) {
        ShadowThread td = e.getThread();
        VectorClock cv = new VectorClock(td.getTid() + 1);
        ts_set_cv_hb(td, cv);
        super.create(e);
}
```

- **access() :** event handler method is called for each memory access which will first find the type of access ie. whether it is a read access or write access.
- If it is a read access then it will check whether this read access is race free or not by checking whether it happened after last write of each thread or not, by calling checkAfter() method.
- If it is a write access then it will check whether this write access is race free or not by checking whether all the last writes from all threads have happened before current write and all the last reads from all the threads have happened before current write by calling checkAfter() method.

```
if (isWrite)
{
    // check after prev read
    passAlong |= checkAfter(p.rd, "read", currentThread, "write", fae,
   true, p);
    // check after prev write
    passAlong |= checkAfter(p.wr, "write", currentThread, "write", fae,
   true, p);
}
else
{
    // check after prev write
    passAlong |= checkAfter(p.wr, "write", currentThread, "read", fae,
   true, p);
}
```

- **checkAfter()** method is used to check the previous operations vector clock is lesser than current thread vector clock and returns error if it is not.
- HB maintains two vector clocks for each variable to detect conflicting memory accesses.
- When a variable is first accessed Road Runners HB tool calls its makeShadowVarMethod() which initializes shadowVar with two vector clocks (for both read and write) by calling VectorClockPair().

```
public ShadowVar makeShadowVar(AccessEvent fae) {
        return new VectorClockPair();
}
public class VectorClockPair implements ShadowVar {
    public final VectorClock rd;
    public final VectorClock wr;

    public VectorClockPair() {
        rd = new VectorClock(1);
        wr = new VectorClock(1);
        VectorClockPairs.inc();
    }
}
```

- HB tool creates a vector clock for each lock object by calling make() method for decoration of shadow lock class which stores mapping from shadowlock to vector clock in decoration map.

```
Decoration<ShadowLock, VectorClock> shadowLock = ShadowLock.
    decoratorFactory.make("HB:lock",
DecorationFactory.Type.MULTIPLE, new DefaultValue<ShadowLock,
    VectorClock() {
    public VectorClock get(ShadowLock ld) {
        return new VectorClock(1);
    }
});
```

- When a thread acquires a lock then HB tool updates Current thread vector clock, By finding the max of vector clocks of current thread and vector clock of the lock.

```
    public void acquire(AcquireEvent ae) {
        final ShadowThread currentThread = ae.getThread();
        final ShadowLock shadowLock = ae.getLock();
        synchronized (shadowLock) {
            get(currentThread).max(get(shadowLock));
        }
    }

```

max() method of Vector Clock class does the join of two vector clocks.

- When a thread releases the lock, It updates vector clock of that lock to the vector clock of the current thread by calling copy() method on vector clocks and increments the vector clock of current thread.

```
public void release(ReleaseEvent re) {
  final ShadowThread currentThread = re.getThread();
  final ShadowLock shadowLock = re.getLock();
  synchronized (shadowLock) {
    get(shadowLock).copy(get(currentThread));
  }
}
```

FastTrack 2 (FT2) :

- When a thread is first created Road Runner calls create() method which will create a vector Clock for that thread similar to HB tool.
- FT2 tool associates a vector clock for each lock object. When a thread releases the lock, It updates vector clock of that lock to the vector clock of that thread similar to HB tool. When a thread acquires a lock then FT2 updates Current thread vector clock by finding the max of vector clocks of current thread and lock similar to HB tool.
- For every memory location when the first access is made makeShadowVar() method is called which initializes the epoch for both read and write unlike the HB tool which creates vector clock for both read and write.
- Following is the sample code for makeShadowVar() method when a first non volatile access is made for a memory location. Which creates a **FTVarState** object, which contains clock state (epoch for write, epoch or vector clock) for each shared variable. Intially this FTVarState object contains two epoch one for read another for write.

```
public ShadowVar makeShadowVar(final AccessEvent event) {
   return new FTVarState(event.isWrite(), ts_get_E(event.getThread()));
}
//—————— FTVarState Class ————————————
public class FTVarState extends LongVectorClock implements ShadowVar {
  public volatile long/* epoch */ W;
```

```java
  public volatile long/* epoch */ R;
  public FTVarState(boolean isWrite, long/* epoch */ epoch) {
    if (isWrite) {
      R = LongEpoch.ZERO;
      W = epoch;
    } else {
      W = LongEpoch.ZERO;
      R = epoch;
    }
  }
  public synchronized void makeCV(int len) {
    super.makeCV(len);
  }
}
```

- access() is called for each memory access, which will identify the type of memory access (read or write) and call the corresponding method (read() or write()) which checks whether it is a race free access or not.
- But Road Runner maintains the counter for each state of both read and write, and updates the counter of the corresponding state whenever read or write access enters into that state if **RR-Mode** is set to **slow** in msetup file , COUNT_OPERATIONS flag will be set RR_mode is set slow.
- **read() :**
  - Read Same Epoch => RR wont update the state. But Road Runner maintains the count the increments the count of readSameEpoch counter if RRMode = slow.
  - Read Shared Same Epoch => Road Runner handles it same as it is for Read Same Epoch.
  - Write Read Race => Road Runner detects it by comparing the thread id of previous thread with current thread id and comparing the epochs then print error message if there is a race. RR increments count of writeReadError counter if RRMode = slow.

```java
if (wTid != tid && !LongEpoch.leq(w, tV.get(wTid))) {
  if (COUNT_OPERATIONS)
      writeReadError.inc(tid);
  error(event, sx, "Write-Read Race", "Write by ", wTid, "Read by ",
    tid);
}
```

  - Read Share => create a vector clock for read and initialize it. Update the read to read_shared. Increment the readShare counter if RR_Mode = slow.
  - Read Shared => Update the vector clock of read with respect current thread epoch.Increment the readShared counter if RR_Mode = slow.

- **write() :**
  - Write Same Epoch => Road Runner wont update the state. Increments the writeSameLongEpoch counter if the RR_Mode = slow.
  - Write Exclusive => Road Runner update Write epoch to current epoch and Increment the WriteExclusive() counter if RR_Mode = slow.
  - Write - Shared => Road Runner handles it same as Write Exclusive.
  - Write - Write Race => Road Runner detects it by comparing thread ids of previous write with current write and epochs between them. Prints the error upon detecting the race. Increments the WriteWriteError count if RR_Mode = slow.
  - Read - Write Race => Road Runner detects it by comparing thread ids of previous read with current write and epochs between them. Prints the error upon detecting the race. Increments the ReadWriteError count if RR_Mode = slow.
  - Shared - Write Race => Road Runner detects it by checking for Read-Shared access or not and comparing the epochs. Finds read thread Id with which current thread write is conflicting and prints error message. Increments SharedWriteError count if RR_Mode = slow.

Sampling

- **Strategy :** Implemented naive random sampling.
  if Sampling Rate is 5% sampling every 20th access.
  if Sampling Rate is 50% sampling every 2nd access.

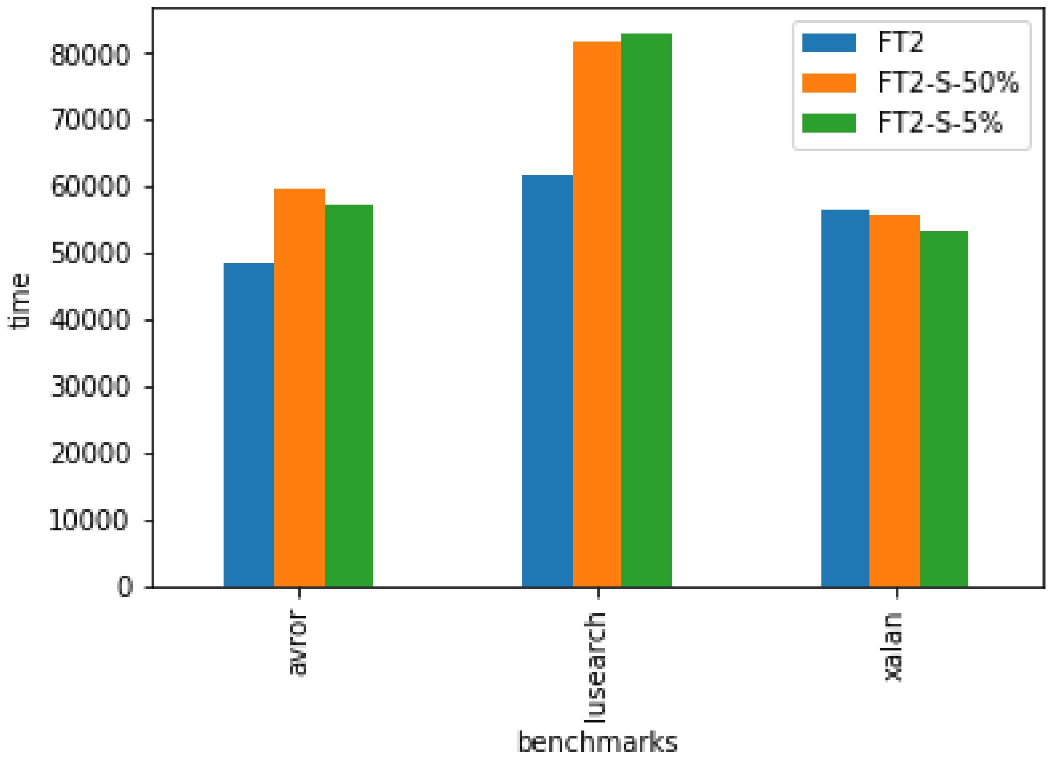
	FT2	FT-S w/ 50% sampling	FT-S w/ 5% Sampling
avrora	300	300	195
lusearch	0	0	0
xalan	180	7	0

Table 1: Impact of sampling on the data race coverage of FastTrack

	FT2	FT-S w/ 50% sampling	FT-S w/ 5% Sampling
avrora	48403	59767	57319
lusearch	61894	81970	82987
xalan	56490	55756	53317

Table 2: Impact of sampling on the performance(Time) of FastTrack

Performance Plot



Program Locations for Data races in FT2

- Xalan BenchMark

```
org/apache/xml/serializer/OutputPropertiesFactory.m_text_properties_Lja
org/apache/xml/serializer/CharInfo.firstWordNotUsed_I
org/apache/xml/utils/res/CharArrayWrapper.m_char_\[C
org/apache/xml/serializer/CharInfo$CharKey.m_char_C
org/apache/xalan/templates/ElemNumber.m_alphaCountTable_Lorg/apache/xml
org/apache/xml/serializer/CharInfo.onlyQuotAmpLtGt_Z
org/apache/xml/serializer/CharInfo.m_charToString_Ljava/util/HashMap;
```

- Avrora Benchmark

```
avrora/sim/radio/Medium$Transmission.lastBit_J
avrora/sim/radio/Medium.Pr_D
avrora/sim/radio/Medium.Pn_I
```

Program Locations for Data races in FT2-S 50%

- Xalan BenchMark

```
org/apache/xml/utils/res/CharArrayWrapper.m_char_\[C
org/apache/xml/serializer/CharInfo$CharKey.m_char_C
rd_array@org/apache/xml/utils/res/CharArrayWrapper.java:36:5
```

- Avrora Benchmark

```
avrora/sim/radio/Medium$Transmission.lastBit_J
avrora/sim/radio/Medium.Pr_D
avrora/sim/radio/Medium.Pn_I
```

Program Locations for Data races in FT2-S 5%

- Avrora Benchmark

```
avrora/sim/radio/Medium$Transmission.lastBit_J
avrora/sim/radio/Medium.Pr_D
avrora/sim/radio/Medium.Pn_I
```