Cloud Databases Final Project

NVRAM-aware Logging in Transaction Systems (VLDB'14)

Group 2
102061105 Yun-Sheng, Chang
102061212 Jian-Hao, Huang

June 19, 2017

Links to our repositories:

- https://github.com/yschang1206/vanillacore
- https://github.com/yschang1206/vanillabench

1 Main Idea

While logging is an essential means for guaranteeing the ACID properties of database systems, logging itself can also become a bottleneck. As analyzed in this paper, log operations account for 60.73 - 85.46% of the total time when logs are placed on HDD/SSD. Intuitively, placing logs on high-speed non-volatile memory technology (NVRAM) might be a good strategy to enhance the overall performance of database systems. However, simply replacing HDD/SSD with NVRAM without any modification to the logging subsystem could be a suboptimal design since conventional database systems are optimized for slow disk storage rather than fast memory technology. More precisely, conventional database systems use a log buffer in DRAM and a log partition in persistent storage. This design has two advantages:

- it reduces the number of I/O operations by coalescing multiple logging requests;
- it can perform sequential access pattern which favors the slow disk storage, particularly HDD.

But these two advantages no longer exist for a NVRAM-based database system because of the characteristics of NVRAM:

- NVRAM is byte-addressable (opposed to disk's block-accessed interface), so coalescing multiple logging requests will not decrease the number of I/O operations;
- NVRAM has its random access speed close to its sequential access speed.

Furthermore, this design can cause **contention for the centralized log buffer** on a multi-threaded database system because only one thread is able to populate log record or flush the log buffer at a time. Based on the analysis and observation above, this paper proposes a decentralized logging design to optimize NVRAM-based transaction systems. In the next section, we will describe the basic design of this paper, including its data structure and logging operations.

2 Design

2.1 Data Structure in DRAM

2.1.1 Per-transaction log record lists

Every transaction that is not yet committed or aborted will have a list containing its own log record objects, which will be used to rollback an transaction if it is aborted.

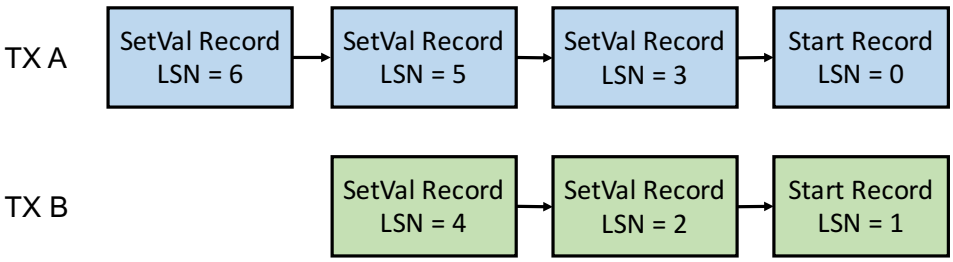

Figure 1: In-DRAM data structure

2.2 Data Structure in NVRAM

Data structure in NVRAM consists of a circular log buffer, a global log sequence number (LSN) and two log pointers.

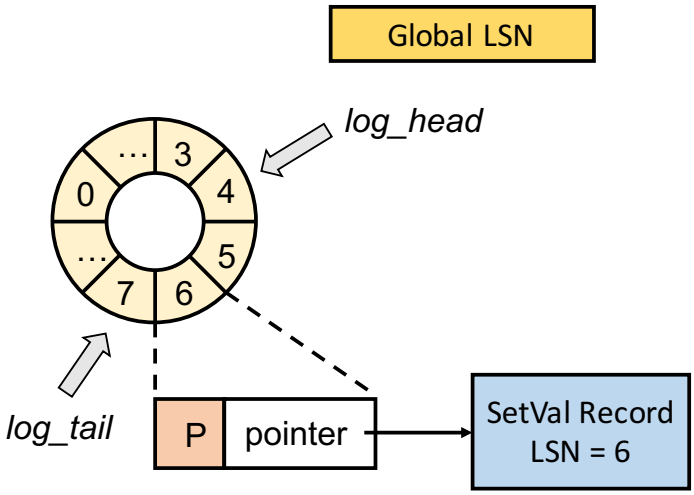

Figure 2: In-NVRAM data structure

2.2.1 Circular log buffer

The circular log buffer has a fixed (but parameterizable) number of entries, each entry consists of a persistent flag and a pointer points to a log record object in NVRAM. The persistent flag indicates whether the corresponding log record object has been fully populated in NVRAM. It is worth noting that the persistent flag should always be set **after** the log record object is persisted in NVRAM (more detail appears in Section 3.3); otherwise, the log records may be left in its intermediate form if crash happens (i.e., not atomic). During recovery, each valid log entry (defined in Section 2.2.3) will be examined one by one in a proper order to perform the ARIES-based recovery algorithm.

2.2.2 Global LSN

The global LSN is a 64-bit integer for maintaining the global order of all log records. When a transaction attempts to append a log record, the log record will be assigned an unique LSN. With the unique LSN, the log record will be able to know its location in NVRAM (more detail appears in Section 2.2.3) and can be used to tracked for its persistence (illustrated in Section 4.2).

2.2.3 Log pointers

The two log pointers (*log_tail and log_head*) have the same structure, both consisting of a LSN and the index of an entry in the circular log buffer. The log record entry pointed by the *log_head* is the first valid log record; the log record entry pointed by the *log_tail* is the first invalid log record. Once the log record entry pointed by the *log_tail* is persisted, the *log_tail* will move forward to the next entry.

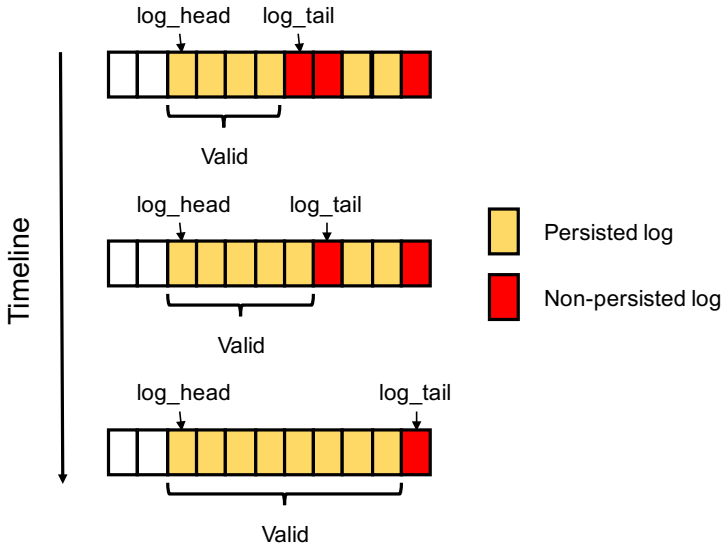

Figure 3: Move *log_tail* forward

Since the size of the circular log buffer is limited, a checkpoint must be initiated to truncate some log records when the number of valid entries reaches a predetermined threshold. During the checkpointing process, the *log_head* will move forward and the persistent flag in the log entries passed in that move will be reset.

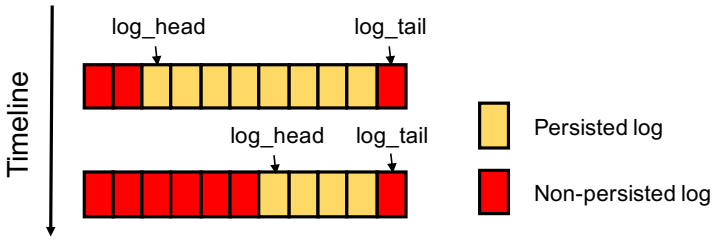

Figure 4: Move *log_head* forward

Moreover, *log_head* is responsible for locating a log entry with its LSN. The formula is straightforward, shown in below (C denotes the circular log buffer):

Algorithm 1 Locating a log entry (Input: lsn; Output: index)

$index \leftarrow log\_head.index + (lsn - log\_head.lsn)$
$index \leftarrow index\%C.size$

3 Implementation Issues

In this section, we will describe some implementation issues and present the paper's solutions and our solutions.

3.1 NVRAM Access Latency

Since byte-addressable high-speed NVRAM is not available currently, and NVRAM is usually slightly slower than DRAM, additional access latency to NVRAM should not be ignored.
Paper's solution: Intels Persistent Memory Emulation Platform (PMEP) is used to emulate the additional latency.
Our solution: we simulate the additional latency through busy waiting.

3.2 In-NVRAM Data Structure

The key issue is that the pointers stored in the data structure are *virtual memory pointers*. Everytime DBMS is restarted, the *virtual-to-physical mapping* could be different and thus the data structure in NVRAM is corrupted.
Paper's solution: library support.
Our solution: we serialize/deserialize the data structure in NVRAM before DBMS is terminated/upon DBMS is started to maintain the state in NVRAM.

3.3 Data Persistence

On modern processors, data written to memory is not guaranteed to reach the memory as a write-back cache may have served the request. This is a problem because if we cannot guarantee the log record object to be persited in NVRAM, we might fail to recover the system correctly on system crash.
Paper's solution: instruction *clflush* (X86) can flush the data in cache to memory.

Algorithm 2 Correct way to make a log record object persistent

write the log record object to its location in NVRAM
invoke $clflush$
set the persistent flag

We do not have a solution for this problem, so our system is actually not guaranteed to recover from crashes. We believe that it is not a big issue as our purpose is to implement a prototype rather than a fully functional system.

4 Implementation Detail

The figure below is the overview of our design. We will discuss each part of them with details in this section.

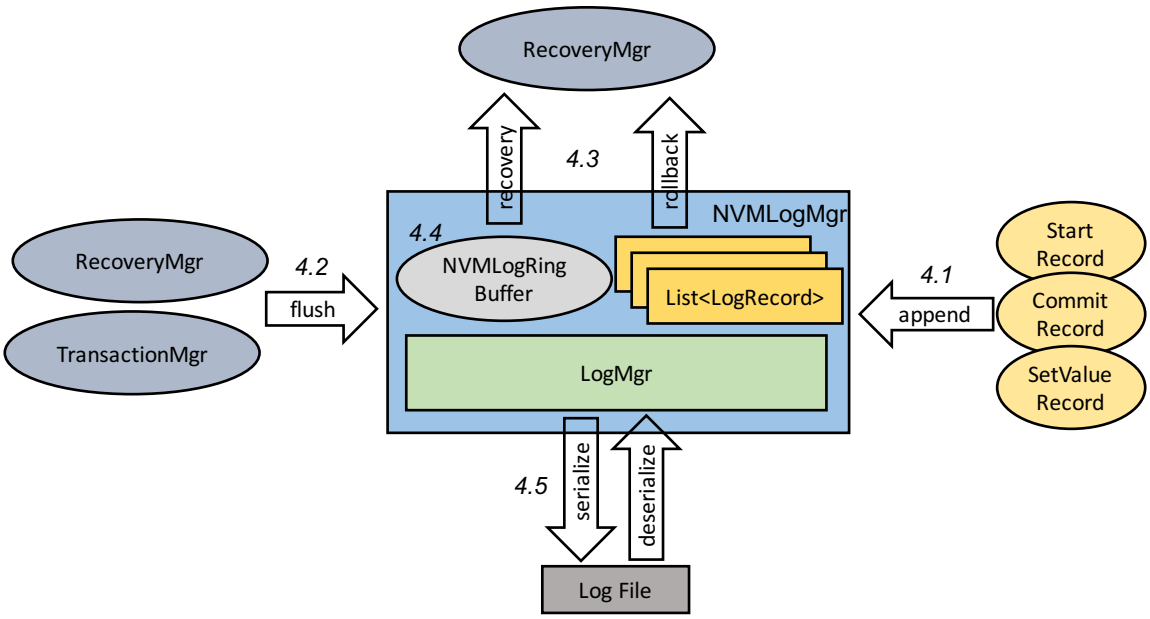

Figure 5: Overview of our design

4.1 Append Record

Appending a log record works as follows: the log record object would be added to the per-transaction log record list that it belongs to. Next, the log record will acquire a LSN and use the LSN to insert itself to the corresponding location.

```
public LogSeqNum append(LogRecord rec) {
  /* add to per-tx list (volatile) */
  long txNum = rec.txNumber();
  List<LogRecord> list = txLogListMap.get(txNum);
  if (list == null) {
    list = new LinkedList<LogRecord>();
    txLogListMap.put(txNum, list);
  }
  list.add(rec);

  /* add to circular buffer (non-volatile) */
  logMgrLock.lock();
  long lsn = globalLsn++;
  logMgrLock.unlock();
  ringBuffer.insert(rec, lsn);
  return new LogSeqNum(lsn);
}
```

NVMLogMgr.java

```
public void insert(LogRecord rec, long lsn) {
  logHeadLock.readLock().lock();
  int idx = logHead.idx + (int)(lsn - logHead.lsn);
  idx = idx % size;
  ring[idx].rec = rec;
  delay();
  ring[idx].isPersist = true;
  delay();
  logHeadLock.readLock().unlock();
}
```

NVMLogRingBuffer.java

4.2 Flush

The semantic of a *flush(lsn)* operation is to block until all the log records with LSN less than the specified LSN are persisted. Our design guarantees that all the log records in the region $[log\_head, log\_tail)$ are persisted. Thus, the actions that a transaction invokes *flush(lsn)* should take are:

- check if the *log_tail* has passed the specified LSN;
- move the *log_tail* forward as far as possible.

```
public void flush(LogSeqNum lsn) {
  ringBuffer.checkPersistence(lsn.val());
}
```

NVMLogMgr.java

```
public void checkPersistence(long lsn) {
  while (lsn >= logTail.lsn) {
    moveTailForward();
  }
}

public void moveTailForward() {
  synchronized (logTail) {
    int idx = logTail.idx;
    long lsn = logTail.lsn;

    while (ring[idx].isPersist) {
      idx = (idx + 1) % size;
      lsn++;
```

```
    }
    delay();
    logTail.idx = idx;
    delay();
    logTail.lsn = lsn;
  }
}
```

NVMLogRingBuffer.java

4.3 Rollback and Recovery

While the rollback and recovery algorithm remain the same, we no longer scan the log file and fetch the log records one by one. To rollback a transaction, we iterate through the per-transaction log record list in DRAM.

```
private void doRollback(Transaction tx) {
  Iterator<LogRecord> iter = VanillaDb.nvmLogMgr().
      getTxLogRecordIterator(tx.getTransactionNumber());
  // undo operations...
}
```

RecoveryMgr.java

```
public Iterator<LogRecord> getTxLogRecordIterator(long txNum) {
  List <LogRecord> list = txLogListMap.get(txNum);
  if (list == null)
    return null;
  return (new LinkedList<LogRecord>(list)).descendingIterator();
}
```

NVMLogMgr.java

To recover the database system, we iterate through the circular log buffer in NVRAM.

```
private void doRecover(Transaction tx) {
  ReversibleIterator<LogRecord> iter = VanillaDb.nvmLogMgr().
      getLogRecordIterator();
  // ARIES-based recovery algorithm...
}
```

RecoveryMgr.java

```
public ReversibleIterator<LogRecord> getLogRecordIterator() {
  return new NVMLogIterator(ringBuffer);
}
```

NVMLogMgr.java

The reversible iterator will start at the log record prior to the log record pointed by *log_tail* and end at the log record pointed by *log_head*.

```
private NVMLogRingBuffer ringBuffer;
private int lower, upper;
private int currentIdx;

public NVMLogIterator(NVMLogRingBuffer ringBuffer) {
  this.ringBuffer = ringBuffer;
  this.lower = ringBuffer.headIdx();
  this.upper = ringBuffer.tailIdx() - 1;
  this.currentIdx = upper;
}

@Override
public boolean hasNext() {
  return (currentIdx > lower);
}

@Override
public LogRecord next() {
```

```
19    LogRecord rec = ringBuffer.get(currentIdx);
20    currentIdx--;
21    return rec;
22  }
23
24  @Override
25  public boolean hasPrevious() {
26    return (currentIdx < upper);
27  }
28
29  @Override
30  public LogRecord previous() {
31    LogRecord rec = ringBuffer.get(currentIdx);
32    currentIdx++;
33    return rec;
34  }
```

NVMLogIterator.java

4.4 Checkpoint

Since we have limited number of log entries, we create a checkpoint when the utilization of the circular log buffer is higher than 0.7.

```
1  public void createCheckpoint() {
2    // other checkpoint methods...
3    else if (MY_METHOD == METHOD_NVM) {
4      if (VanillaDb.nvmLogMgr().utilization() > 0.7) {
5        if (logger.isLoggable(Level.INFO))
6          logger.info("Start creating checkpoint");
7        Transaction tx = VanillaDb.txMgr().newTransaction(
8            Connection.TRANSACTION_SERIALIZABLE, false);
9        VanillaDb.txMgr().createCheckpoint(tx);
10       tx.commit();
11     }
12   }
13 }
```

CheckpointTask.java

```
1  public double utilization() {
2    long headLsn = ringBuffer.headLsn();
3    long tailLsn = ringBuffer.tailLsn();
4    int size = ringBuffer.size();
5    return (double)(tailLsn - headLsn) / (double)size;
6  }
```

NVMLogMgr.java

```
1  public void createCheckpoint(Transaction checkpointTx) {
2    List<Long> txNums;
3    synchronized (this) {
4      txNums = new LinkedList<Long>(activeTxs);
5      checkpointTx.bufferMgr().flushAll();
6      LogSeqNum lsn = checkpointTx.recoveryMgr().checkpoint(txNums);
7      VanillaDb.nvmLogMgr().flush(lsn);
8    }
9  }
```

TransactionMgr.java

```
1  public LogSeqNum checkpoint(List<Long> txNums) {
2    VanillaDb.nvmLogMgr().checkpoint(txNums);
3    return new CheckpointRecord(txNums).writeToLog();
4  }
```

RecoveryMgr.java

```
public void checkpoint(List<Long> txNums) {
  ringBuffer.moveHeadForward(txNums);
}
```

NVMLogMgr.java

The actual tasks of a checkpoint are defered to this point. During the checkpointing process, the *log_head* will move forward until

- the *log_head* reaches the *log_tail*, or
- the *log_head* reaches a *StartRecord* of a active transaction (uncommitted transaction during checkpointing).

All the persistent flags in the log entries passed in that move will be reset.

```
public void moveHeadForward(List<Long> txNums) {
  logHeadLock.writeLock().lock();
  int idx = logHead.idx;
  long lsn = logHead.lsn;
  long tailLsn = logTail.lsn;

  while (lsn < tailLsn) {
    if (ring[idx].rec.op() == LogRecord.OP_START &&
        txNums.contains(ring[idx].rec.txNumber()))
      break;
    ring[idx].rec = null;
    ring[idx].isPersist = false;
    idx = (idx + 1) % size;
    lsn++;
  }
  delay();
  logHead.idx = idx;
  delay();
  logHead.lsn = lsn;
  logHeadLock.writeLock().unlock();
}
```

NVMLogRingBuffer.java

4.5 Rebuild Data Structure in NVRAM

In order to serialize the data structure in NVRAM before our system is terminated, we create a new action for the TPCC benchmarker. Now, setting the program argument to 3 will serialize the datastructure in NVRAM.

```
public static void main(String[] args) {
  ...
  switch (action) {
  case 1: // Load testbed
    benchmarker.loadTestbed();
    break;
  case 2: // Benchmarking
    benchmarker.benchmark();
    break;
  case 3: // Persist
    benchmarker.persist();
    break;
  }
  ...
}
```

App.java

We now show the implementation of serialization here. First, the global LSN, *log_tail*, *log_head* and size of the circular log buffer are written to file.

```
public void persist() {
  try {
    DataOutputStream dos;
    File f = new File(FileMgr.getLogDirectoryPath(), NVM_DATA_STRUCTURE_FILE);
    dos = new DataOutputStream(new FileOutputStream(f));
```

```
    dos.writeLong(globalLsn);
    dos.writeInt(ringBuffer.size());
    dos.writeInt(ringBuffer.tailIdx());
    dos.writeLong(ringBuffer.tailLsn());
    dos.writeInt(ringBuffer.headIdx());
    dos.writeLong(ringBuffer.headLsn());
    dos.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
  ringBuffer.persist(logMgr);
}
```

NVMLogMgr.java

Then, the log records stored in the circular log buffer are written to file by the original *LogMgr*.

```
public void persist(LogMgr logMgr) {
  LogPosition p = null;
  for (int i = logHead.idx; i < logTail.idx; i++) {
    List<Constant> l = ring[i].rec.buildRecord();
    p = logMgr.append(l.toArray(new Constant[l.size()]));
  }
  logMgr.flush(p);
}
```

NVMLogRingBuffer.java

Deserialization is similar. First, the global LSN, *log_tail*, *log_head* and size of the circular log buffer is rebuilt upon the *NVMLogMgr* is constructed.

```
public NVMLogMgr() {
  File f = new File(FileMgr.getLogDirectoryPath(), NVM_DATA_STRUCTURE_FILE);
  if (f.exists()) {
    try {
      DataInputStream dis = new DataInputStream(
          new FileInputStream(f));
      this.globalLsn = dis.readLong();
      int size = dis.readInt();
      int tailIdx = dis.readInt();
      long tailLsn = dis.readLong();
      int headIdx = dis.readInt();
      long headLsn = dis.readLong();
      dis.close();
      ringBuffer = new NVMLogRingBuffer(size, tailIdx, tailLsn,
          headIdx, headLsn);
      ringBuffer.rebuild();
    } catch (Exception e) {
      e.printStackTrace();
    }
  } else {
    this.globalLsn = 0;
    ringBuffer = new NVMLogRingBuffer(NVM_RING_BUFFER_SIZE, 0, 0, 0, 0);
  }
}
```

NVMLogMgr.java

Then, log record is fetched one by one by scanning the log file and is inserted to the circular log buffer with a proper LSN.

```
public void rebuild() {
  ReversibleIterator<LogRecord> iter = new LogRecordIterator();
  long lsn = logTail.lsn - 1;
  while (iter.hasNext()) {
    LogRecord rec = iter.next();

    rec.setLSN(new LogSeqNum(lsn));
    insert(rec, lsn);
    lsn--;
  }
}
```

NVMLogRingBuffer.java

5 Evaluation

5.1 Experiment Environment

Intel Core i7-4790 CPU @ 3.60 GHz
32 GB RAM
1TB HDD
Ubuntu Linux 16.04
TPCC benchmark

5.2 Performance

Number of RTEs	Bufferpool size	NVM extra latency	Execution time	Circular buffer size
10	102400	variable (ns)	1 (min)	15000000

Table 1: Experiment parameters

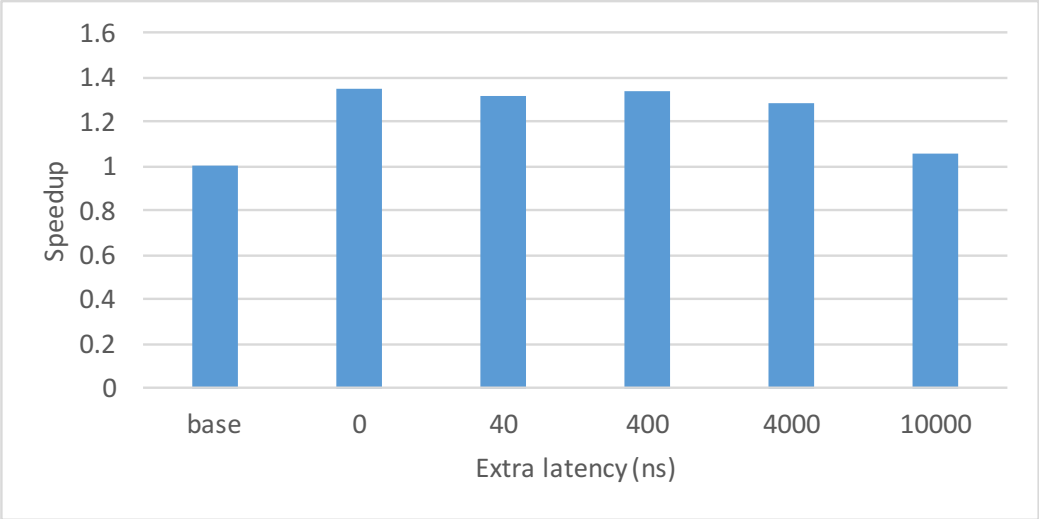

Figure 6: Speedup

To observe the speedup of our design, we properly choose the baseline design for comparison. The baseline design use the latest version of *VanillaDB* on *GitHub* without any modification, but the log file is placed on a *ramdisk*. The additional latency for NVM is set from 0 ns to 10 us and we find the throughput increases by 1.05 - 1.34x. Observing from the result, even though the additional latency for accessing NVRAM is added, we still have a better performance. We think the speedup is mainly come from:

- decentralized logging design eliminates the contention for accessing single centralized buffer by multiple threads;
- rollback a transaction is faster because we no longer need to fetch the log records from disk.

5.3 Concurrency

Number of RTEs	Bufferpool size	NVM extra latency	Execution time	Circular buffer size
variable	102400	400 (ns)	1 (min)	15000000

Table 2: Experiment parameters

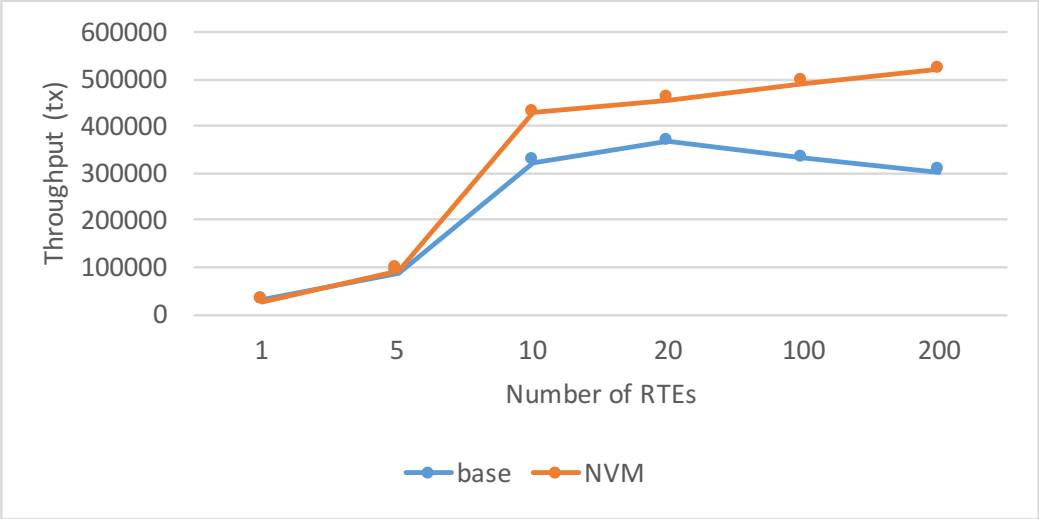

Figure 7: Throughput trend as number of RTEs increases

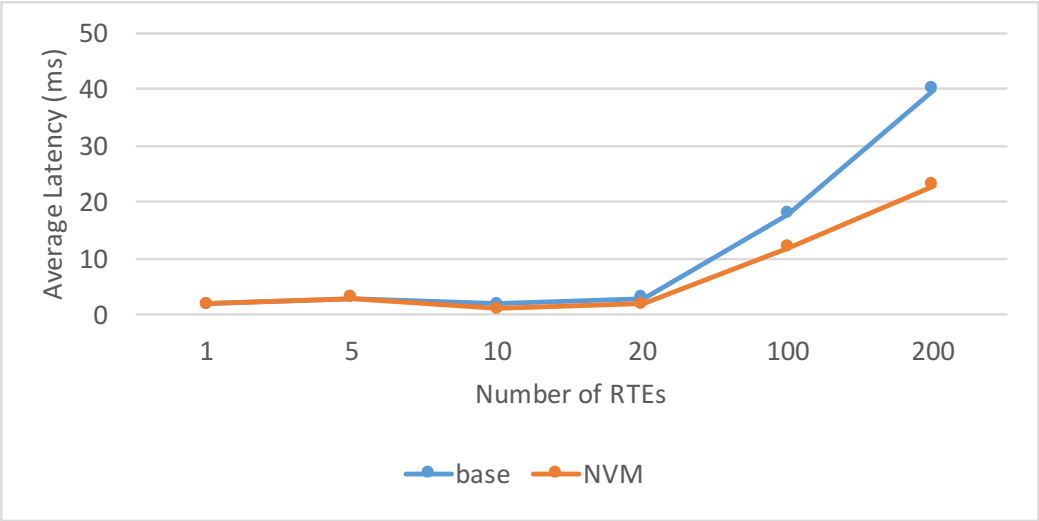

Figure 8: Latency trend as number of RTEs increases

A potential advantage of decentralized logging design is its resilience to concurrent requests. To prove our assumption, we vary the number of RTEs from 1 to 200. We find that the throughput does not drop even when the number of RTEs exceeds 20. Also the increasing tendency of latency is less obvious in our design.

5.4 Impact of Checkpointing

Number of RTEs	Bufferpool size	NVM extra latency	Execution time	Circular buffer size
10	102400	400 (ns)	3 (min)	variable

Table 3: Experiment parameters

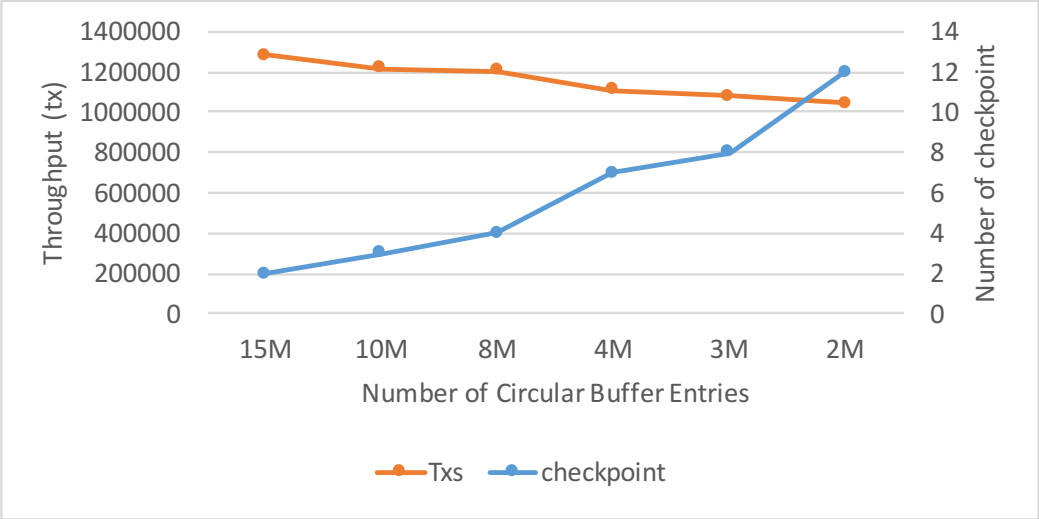

Figure 9: Impact of Checkpointing

The size of the circular log buffer would determine the frequency of checkpointing. The larger the circular buffer is, the less frequent a checkpoint will be initiated. A large circular buffer, however, would consume a huge amount of memory resources. We vary the size of the circular log buffer from 15,000,000 entries to 2,000,000 entries. In our experiment, it's obvious that the number of checkpoints increases as the number of the log entries decreases; at the same time, the throughput drops as the number of checkpoint increases.