Assignment4 Report

105062206蔡哲維
105062306胡世昕

減少critical section：

在BufferMgr中，有不少地方去synchronized bufferpool，但實際上並不一定會真的去使用bufferpool，這個造成了一些運算資源上的浪費，所以第一步做的就是去減少這些不必要的critical section。

例如我在pin這個method中，將synchronized bufferpool改到進入等待模式的while迴圈之前，這麼做可以讓目前的bufferMgr在已經有需要的buffer的情況下，不需要去跟他人競爭bufferpool，而可以去做其他的事情（因為他已經拿到了要使用的buffer）。

```
// Try to find out if this block has been pinned by this transactio
PinningBuffer pinnedBuff = pinningBuffers.get(blk);
if (pinnedBuff != null) {
    pinnedBuff.pinCount++;
    return pinnedBuff.buffer;
}

/*
 * Throws BufferAbortException if the calling tx requires more buff
 * size of buffer pool.
 */
if (pinningBuffers.size() == BUFFER_POOL_SIZE)
    throw new BufferAbortException();
    // Pinning process
try {
    Buffer buff;
    long timestamp = System.currentTimeMillis();

    // Try to pin a buffer or the pinned buffer for the given Block
    buff = bufferPool.pin(blk);

    if (buff!=null) {
        pinningBuffers.put(buff.block(), new PinningBuffer(buff));
        return buff;
    }
```

又例如在flushAllMyBuffers這個method中，將synchronized bufferpool拿掉，因為其內部是直接呼叫buffer的flush，並不會使用到bufferpool。

```
     * Flushes the dirty buffers modified by the
     */
    public void flushAllMyBuffers() {
        for (Buffer buff : buffersToFlush) {
            buff.flush();
        }

    }
```

將buffer中的pins改成AtomicInteger

原本buffer這個類別的所有method都是synchronized，這也就造成針對一個buffer的instance，只能一次執行一個method。而將pins改成AtomicInteger的目的，就是為了讓pin跟unpin這些動作在執行的同時，也能夠去做setval跟getval的動作。

```
private AtomicInteger pins = new AtomicInteger(0);
```

```
void pin() {

    pins.incrementAndGet();


}

/**
 * Decreases the buffer's pin count.
 */
void unpin() {

    pins.decrementAndGet();

}
```

將bufferpoolMgr中flushAll的synchronized用lock取代

在bufferpoolMgr中，希望在flushAll的同時，pin、pinNew、unpin可以去尋找或是unpin掉bufferMgr需要的buffer，所以把flushAll前的synchronized拿掉，用ReentrantLock來取代。

這樣可以讓bufferpoolMgr的flushAll一次被一個bufferMgr呼叫執行，也能在同時去做其他工作。

```
 */
void flushAll() {
    poolLock.lock();
    try {
        for (Buffer buff : bufferPool)
            buff.flush();
    } finally {
        poolLock.unlock();
    }
}
```

將fileMgr中的read,write,append等method前的synchronized拿掉

在fileMgr這裡，希望能夠讓不同的page可以同時呼叫fileMgr的read和write，去執行讀寫的動作，所以在這裡將synchronized拿掉，改用各自專屬的lock來鎖定一次一個page呼叫該method。但這個修改卻對於程式沒有太大的優化，甚至有些情況會導致效能變差。而我認為主要的原因有兩個：**第一個，fileChannel是blocking mode，所以如果一次要同時針對同一個file做R/W，其中一個動作會被block，等到另一個做完才會去做，這讓預期達到的效果沒辦法達到。**另外，**第二個原因我認為是lock在建立跟call method的時候，cost是高的，**原本用synchronized method，fileMgr只會用到一個lock，但若改成我所使用的方法，則fileMgr會同時擁有三個lock，這**可能造成了整體cost變高，消彌原本想要讓不同page同時讀寫的優化。**

```
private Lock readLock = new ReentrantLock();
private Lock writeLock = new ReentrantLock();
private Lock appendLock = new ReentrantLock();
private Lock channelLock = new ReentrantLock();
```

```
 */
void read(BlockId blk, IoBuffer buffer) {
    readLock.lock();
    try {
        IoChannel fileChannel = getFileChannel(blk.fileName());

        // clear the buffer
        buffer.clear();

        // read a block from file
        fileChannel.read(buffer, blk.number() * BLOCK_SIZE);
    } catch (IOException e) {
        e.printStackTrace();
        throw new RuntimeException("cannot read block " + blk);
    } finally {
        readLock.unlock();
    }
}
```

Never Do it Again:

```
34    public BlockId(String fileName, long blkNum) {
35        this.fileName = fileName;
36        this.blkNum = blkNum;
37        this.hashCodes = toString().hashCode();
38    }
```

storge.file.BlockId中，在創建BlockId物件時已決定fileName與blkNum，且不會再更改。BlockId. hashCode()將呼叫BlockId. toString().hashcode()，將fileName與blkNum與轉成字串並加入固定字串後運算出雜湊碼。因此一個BlockId在創建後所執行BlockId. hashCode()的回傳不會改變。而BlockId. hashCode()會多次被其他物件呼叫，因此將BlockId. toString().hashcode()計算放入創建時期並記錄，BlockId. hashCode()僅回傳this.hashCodes，以減少重複計算。

```
87    @Override
88    public int hashCode() {
89        return hashCodes;
90    }
```

Replacement Strategy

```
112   public Buffer pin(BlockId blk) {
113
114       // Try to find out if this block has been pinned by this transaction
115       PinningBuffer pinnedBuff = pinningBuffers.get(blk);
116       if (pinnedBuff != null) {
117           pinnedBuff.pinCount++;
118           return pinnedBuff.buffer;
119       }
120
121       /*
122        * Throws BufferAbortException if the calling tx requires more buffers than the
123        * size of buffer pool.
124        */
125       if (pinningBuffers.size() == BUFFER_POOL_SIZE)
126           throw new BufferAbortException();
127           // Pinning process
128       try {
129           Buffer buff;
130           long timestamp = System.currentTimeMillis();
131
132           // Try to pin a buffer or the pinned buffer for the given BlockId
133           buff = bufferPool.pin(blk);
```

當bufferMgr進行pin() 或pinNew()時，在該block 未被transaction pinned時呼叫bufferPoolMgr的pin()或pinNew()。

```
137   synchronized Buffer pinNew(String fileName, PageFormatter fmtr) {
138       Buffer buff = chooseUnpinnedBuffer();
```

```
103   synchronized Buffer pin(BlockId blk) {
104       Buffer buff = findExistingBuffer(blk);
105       if (buff == null) {
106           buff = chooseUnpinnedBuffer();
```

bufferPoolMgr.pinNew或是bufferPoolMgr.pin()在找不到符合的<BlockId, Buffer>配對[findExistingBuffer(blk)=null]時會呼叫bufferPoolMgr. chooseUnpinnedBuffer()來獲得一未被pinned的Buffer，與需要pin或pinNew的block進行配對。

bufferPoolMgr.chooseUnpinnedBuffer()--clock

```
206     private Buffer chooseUnpinnedBuffer() {
207         int currBlk = (lastReplacedBuff + 1) % bufferPool.length;
208         while (currBlk != lastReplacedBuff) {
209             Buffer buff = bufferPool[currBlk];
210             if (!buff.isPinned()) {
211                 lastReplacedBuff = currBlk;
212                 return buff;
213             }
214             currBlk = (currBlk + 1) % bufferPool.length;
215         }
216         return null;
```

預設Clock replacement方案為traverse所有bufferPool的buffer，並回傳第一個Buffer.isPinned為False的Buffer[該Buffer pinned個數為0]。該Buffer成為下一次traverse的起點。

使用LRU replacement方案替代Clock replacement，回傳久未被pinned的Buffer，期望減少bufferPoolMgr.pin()在找不到符合的< BlockId, Buffer> entry [findExistingBuffer(blk)=null]的機率。

java.util.LinkedHashMap

```
58          blockMap = new ConcurrentHashMap<BlockId, Buffer>(numBuffs);
59          SyncLinkedMap = Collections.synchronizedMap(new LinkedHashMap<BlockId, Buffer>(numBuffs));
```

LinkedHashMap擴展HashMap並保有entry插入的順序，形成一有序的HashMap。LinkedHashMap在每個access指令[get, put]會將該entry 放到LinkedHashMap的末端，可用此特性實作LRU。使用java.util.Collections.synchronizedMap()對LinkedHashMap進行執行緒保護。

```
120                 SyncLinkedMap.put(blk, buff);
121                 //blockMap.put(blk, buff);
```

使用java.util.LinkedHashMap取代原先用來配對<BlockId, Buffer> entry的java.util.concurrent. ConcurrentHashMap。

bufferPoolMgr.flushAl()

```
73      void flushAll() {
74          poolLock.lock();
75          try {
76              for (Map.Entry<BlockId, Buffer> entry : SyncLinkedMap.entrySet()) {
77                  Buffer buff = entry.getValue();
78                  buff.flush();
79              }
80
81          } finally {
82              poolLock.unlock();
83          }
```

bufferPoolMgr.flushAl()改為traverse LinkedHashMap，並對其value做Buffer.flush()。

```
164    synchronized void unpin(Buffer... buffs) {
165        for (Buffer buff : buffs) {
166            buff.unpin();
167            if (!buff.isPinned()) {
168                numAvailable++;
169                SyncLinkedMap.get(buff.block());
170            }
```

在bufferPoolMgr.unpin()時若Buffer.isPinned=false，對其在LinkedHashMap中的entry進行get()，將該entry拉至Map的末端。及越晚Buffer.isPinned=false的<BlockId, Buffer> entry會在Map的越末端。

---

bufferPoolMgr.chooseUnpinnedBuffer()--LRU

```
192    private Buffer chooseUnpinnedBuffer() {
193        if(SyncLinkedMap.size() < bufferPool.length)
194            return new Buffer();
```

當LinkedHashMap的大小比bufferPool還小時代表bufferPool cache中還有未分配的<BlockId, Buffer> entry，因此直接返回new Buffer();

```
196        Set<BlockId> keys = SyncLinkedMap.keySet();
197        for(BlockId k:keys){
198            Buffer buff = SyncLinkedMap.get(k);
199            if (!buff.isPinned())
200                return buff;
201        }
202        return null;
```

當LinkedHashMap的大小等於bufferPool表示bufferPool cache空間已飽和，必須選出一Buffer.isPinned()=false的Buffer做置換。在進行pin()以及pinNew()時會對該<BlockId, Buffer> entry進行LinkedHashMap.put()，將其放置於末端。而unpinned至Buffer.isPinned()=false時會對該<BlockId, Buffer> entry進行LinkedHashMap.get()，同樣將其放置於末端。LinkedHashMap中越前端表示其最久未被使用[Buffer.isPinned()=false期間最久]。因此traverse LinkedHashMap並返回其第一個buff = entry.value()中buff.isPinned()=false的Buffer極為返回LRU Buffer。

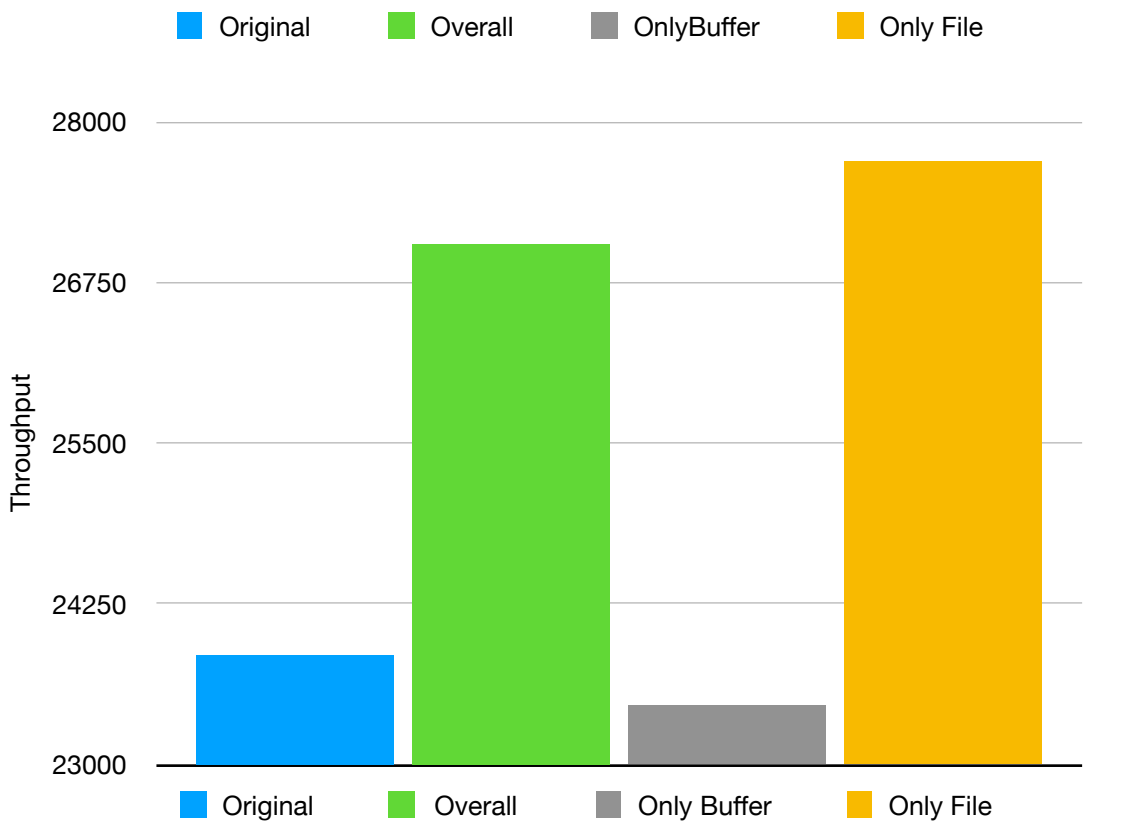
實驗結果與分析

(2.4 GHz 四核心Intel Core i5、8GB RAM、512GB SSD、 macOS Catalina 10.15.4)

TPCC Benchmark

RTE:2

BufferPool: 102400



分析：在這裡可以看到，Overall的優化對於throughput來說有了不少的進步。但也可以發現，其實FileModule的修改dominant了整個程式的優化。我認為這是因為TPCC BenchMark中

NewOrder跟Payment這兩個operation有很多修改資料的動作，為了確保修改的資料能夠寫到硬碟中，會有很多I/O的動作。在這個情況下，節省去等待bufferpool release的時間，所得到的效益反而變得非常小。而減少重複算blockID hashCode，則在這個過程的則得到了很好的發揮。這也造成了在file Module的優化dominant整體優化的原因。

Micro Benchmark

RTE:4

Bufferpool:10240

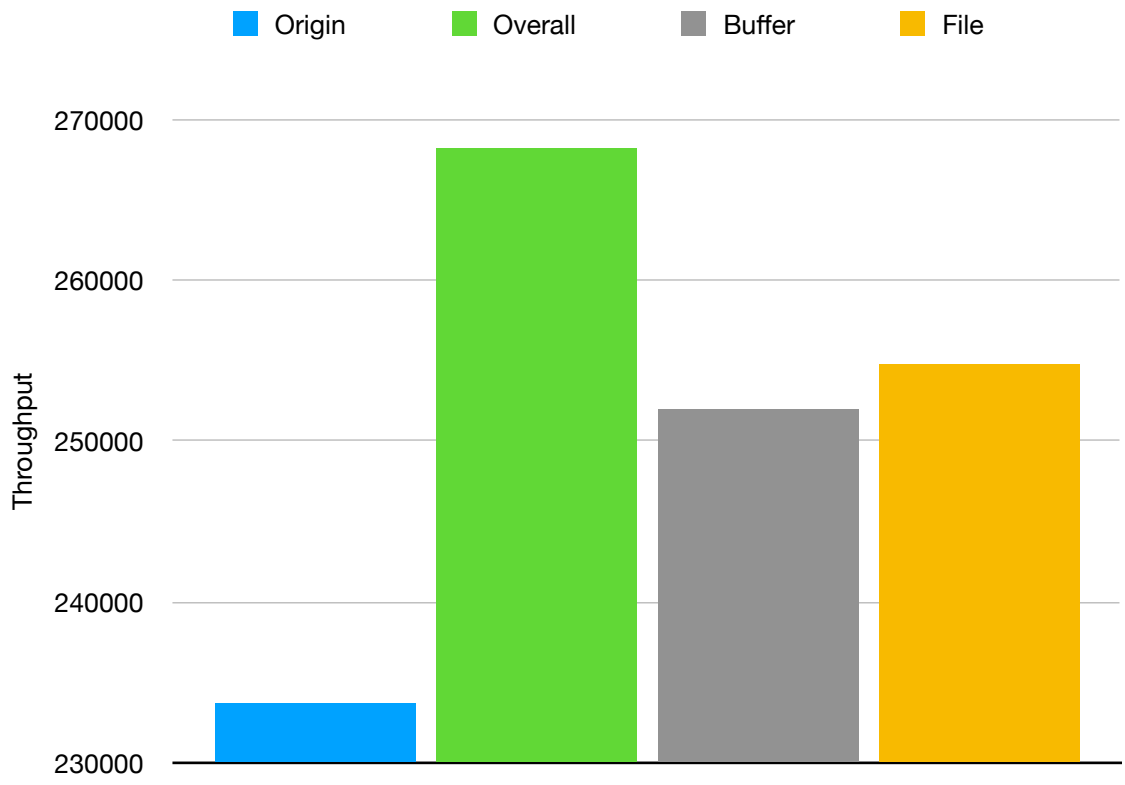
RW_TX_Rate : 0.2

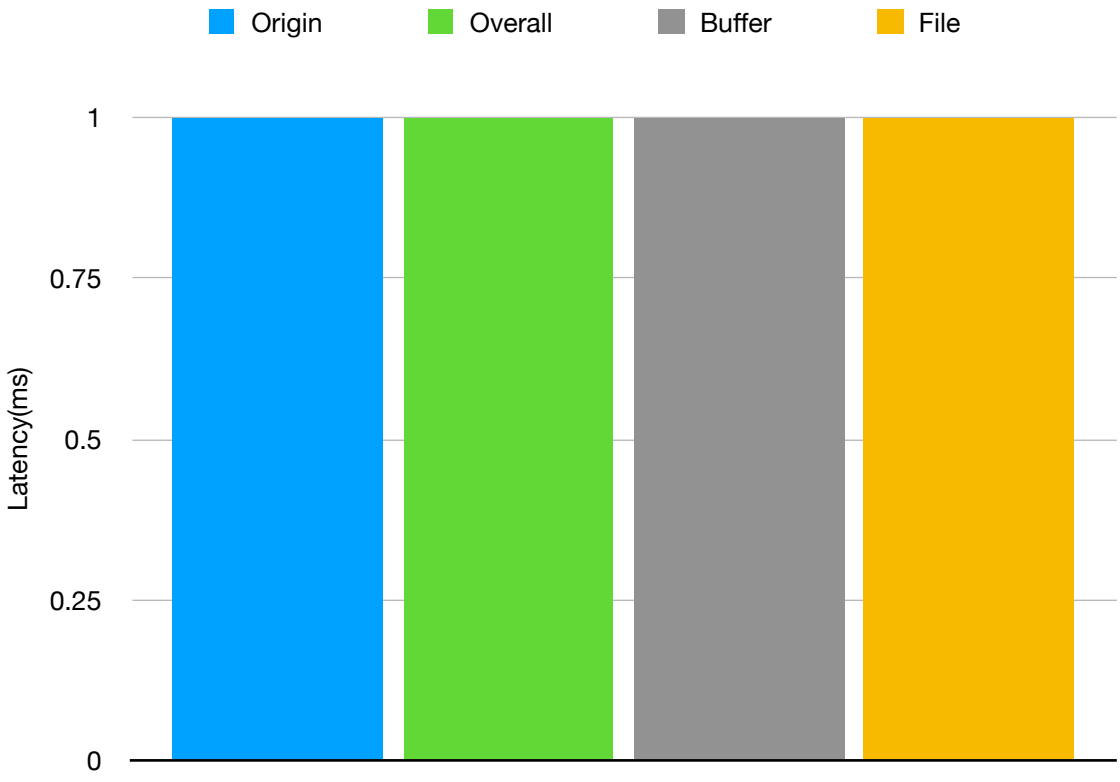
Write_Ratio_in_RW_Tx : 1.0

Total_Read_Count : 10

Local_Hot_Count : 5

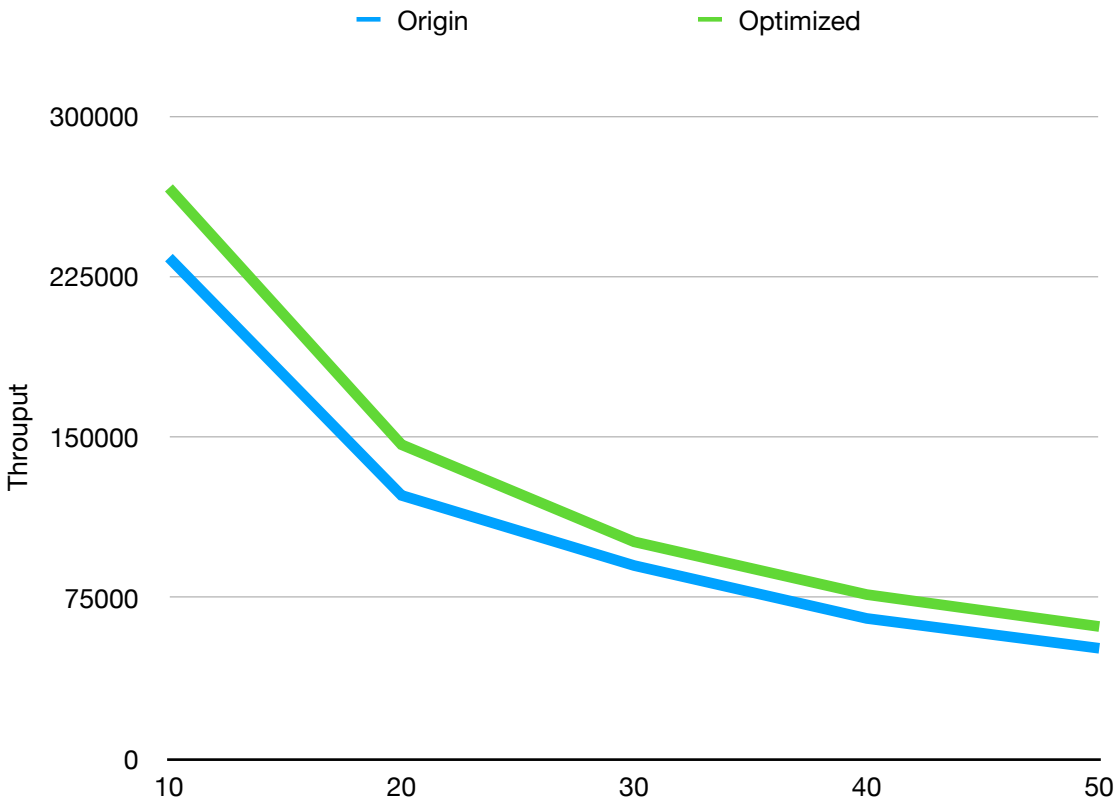
Conflict Rate : 0.001



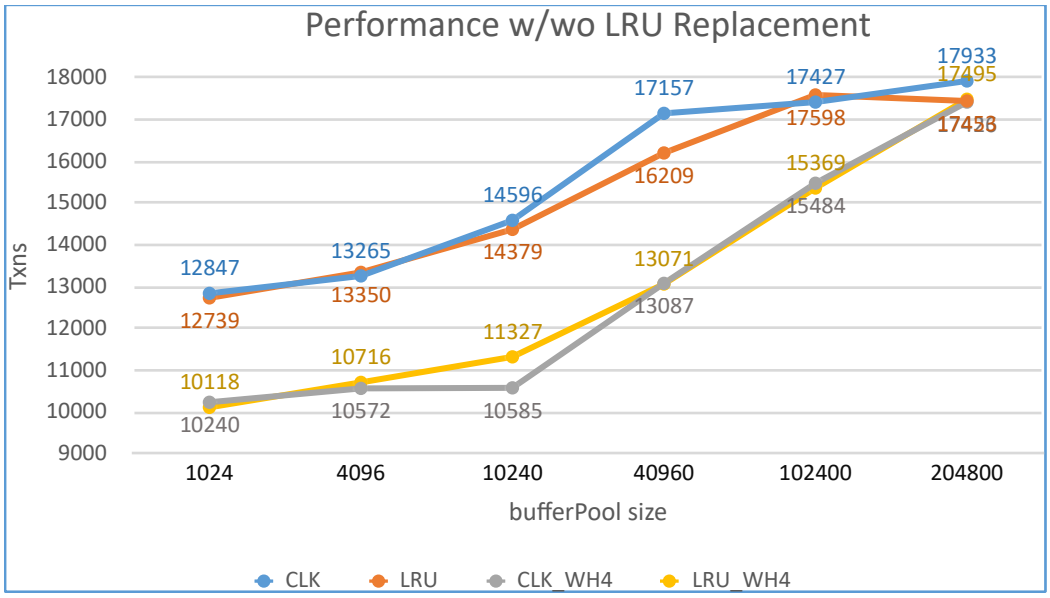


分析：在這個MicroBenchmark的實驗中，看到buffer module和file module的優化都對於throughput產生了一定的效益。這是因為，當Local Hot Count佔了整個transaction的一半時，該transaction的操作會有很大的機率在同一個block，這也就讓原本bufferMgr 已經pin到的buffer重複被用到，減少了使用bufferpool的機會，也就減少了等待取得lock的時間，使得效能提高。另外，在file module的情況也是因為減少了消耗資源很多的hashCode運算，進而讓整體的throughput有了提升。

(以下為在各個Total_Read_Count，Local_Hot_Count為Total_Read_Count一半時，跑出來的結果折線圖)



Replacement Strategy LRU V.S. CLOCK



- Performance comparison with LRU/Clock replacement strategy under TPC-C benchmark. CLK, LRU, CLK_WH4, LRU_WH4 indicates # warehouses in the testing data set.
  Environment: i7 4712MQ @2.3GHz, 16G DDR3 1600MHz, 512G SATA SSD, Win10

分析：將 Replacement方案從CLOCK改成LRU的效能差異並不明顯，推測為於bufferPoolMgr.unpin()中Buffer.isPinned = false時會執行LinkedHashMap.get()，會讓bufferPoolMgr的執行時間增加，而bufferPoolMgr為一singleton物件，會受到執行緒保護，及一次僅有一執行緒可使用。bufferPoolMgr的執行時間增加會導致其他執行緒等待時間增加，因此使用本方法實作之LRU replacement方案結果利弊相消，甚至有時效能不及CLOCK方案。