SGPM: A coroutine scheduling model for wound-wait concurrency control[⋆]

Xinyuan Wang[a], Hejiao Huang[a,*]

[a] *Harbin Institute of Technology (Shenzhen), Shenzhen, 518071, Guangdong, China*

ARTICLE INFO

Keywords:
concurrency control
transaction process
coroutine
OLTP
database

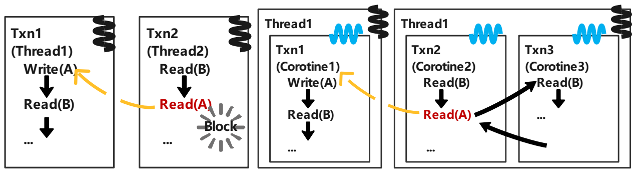
ABSTRACT

Concurrency control protocol with the pure wound-wait method including pessimistic 2PL, SS2PL, and deterministic Calvin, BOHM, and PWV. But compared with the no-wait or wait-die method, they bring a higher conflict rate by false-positive validation and suffer from the blocking by lock thrashing. Previous studies have found that coroutines can effectively reduce the blocking cost of wound-wait asynchronous tasks, but with transaction-to-coroutine schema, that also brings the higher conflict rate since more concurrency transactions. In this paper, we propose a novel coroutine scheduling model, SGPM, which is borrowed from goroutine's GPM model. We break the balanced schedule of goroutine. In this model, only the conflict can trigger the schedule, so the transaction concurrency amount can be decreased, while only the started transactions (which hold lock) are counted as concurrency. SGPM is also different from the current popular coroutine implement (boost, libco, goroutine, etc.), which only supports the swap-out scheduling function (yield, await, return, etc.). SGPM provides various scheduling functions that are designed to meet the scheduling requirements of a particular concurrency control protocol, for controlling the callback cycle more flexibly. We work on perfect of the schedule by adapting common concurrency control protocols to SGPM. In addition, we optimized the performance of four wound-wait concurrency control: 2PL, SS2PL, Calvin, and PWV. The final experiments showed that the improved 2PL and SS2PL outperformed OCC and MVCC, and Calvin, PWV brought 1.3x and 2.5x throughput improvements, respectively.

1. Introduction

No-wait [32], wound-wait [29] and wait-die [29] are most used transaction conflict resolution (subsection 2.1 for detailed). No-wait is mainly used for deadlock prevention in pessimistic concurrency controls (2PL [5], SS2PL [14]) which are traditional and practical protocols. Wait-die is a major mechanism for optimistic concurrency control (OCC [24], MVCC [39]) [23, 4] which is popular in OLTP systems [1, 43, 7, 30, 10, 38]. Deterministic concurrency control (Calvin [33], BOHM [15], PWV [16]) which without circle dependency, primarily use wound-wait mechanism to achieve the sequential execution of transactions. (concurrency control protocols detailed in subsection 2.2)

Wait-die and no-wait reduce the conflict rate of optimistic concurrency by reading (OCC) or writing (MVCC) validation, and of pessimistic concurrency (2PL_NW) by holding less lock. However, that also brings a higher conflict cost by re-execute the statements that have passed. Wound-wait is suffer from locks thrashing [40], and additional processing for deterministic order or deadlock prevention, but coroutine brings the opportunity to reduce the wound lost, by switching to execute the un-blocking transaction. We refer to the Transaction-to-Coroutine Paradigm [20] and propose a new coroutine scheduling model, simple GPM (SGPM).



(a) Transaction-to-Thread (b) Transaction-to-Coroutine

Figure 1: Two transaction models. The transaction-to-thread model causes threads to block, whereas the transaction-to-Coroutine model can switch transactions in case of conflicts to avoid the block.

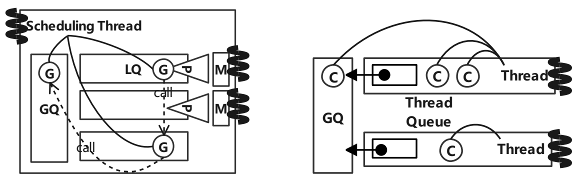
Transaction-to-Coroutine is a derivational concept from Transaction-to-Thread, which both shown in Figure 1a and Figure 1b. In former, Txn2's operation $Read(A)$ is blocked by $Write(A)$ of Txn1 until Txn1 commits. Before Txn1 completes, thread2 is in spin wait all the time. In the Transaction-to-Coroutine model, the conflict that happened in Txn2 will trigger the schedule with switching to execute Txn3 after preserving the Txn2 context. While Txn3 completes, Txn2 will resume at the exit point of the saving context.

SGPM is borrowed from Golang's GPM model as Figure 2a and Figure 2b. The coroutines (transactions) are first allocated to the Global Queue and then joined to one of the Thread queues for execution. Compared to GPM, we present the following advantages.

• **More simpler model** The Processor (P) and Machine (M) of GPM as Figure 2a, which are used to control the concurrency and parallel, are replaced by Thread as Figure 2b. Each Thread has a Queue that holds the coroutines

[⋆] This work is financially supported by Shenzhen Science and Technology Program under Grant No. JCYJ20210324132406016 and National Natural Science Foundation of China under Grant No.61732022.

[*] Corresponding author
huanghejiao@hit.edu.cn (H. Huang)
ORCID(s): 0000-0001-6359-7869 (X. Wang); 0000-0001-8728-3012 (H. Huang)



(a) GPM model (b) SGPM model

Figure 2: GPM model in goroutine and SGPM model. SGPM is simpler than GPM and without the scheduling thread.

is assigned to it. SGPM is a thread-based stackless model (decribed in section 5), and there is no need to manage call relationships between coroutines as GPM. Core switching in threads is scheduled by the operating system.

• **Fewer threads to use**. There is no scheduling thread of GPM as Figure 2a, and all coroutine scheduling is done by the worker thread as shown in Figure 2b. Thread is responsible for scheduling, and executing coroutines in its own Queue. Each Thread Queue maintains a pointer to Global Queue for getting the unassigned coroutines, which is thread-safe.

• **Fewer schedule times**. If scheduling occurs before the transaction completes, it leads to more resource occupied or higher concurrency, thus increasing the conflict rate. Golang schedules coroutines with constant time slices to ensure balanced execution but bring a higher conflict rate than threads. In SGPM scheduling will not occur if there is no conflict or commit, which can minimize the number of coroutine switching. Thus reducing the conflict rate.

• **More flexible scheduling** In SGPM, scheduling policies are controlled by specific transaction protocols which as Figure 7, and are detailed described in subsection 3.3. Each transaction operation returns a scheduling signal. NT signal indicates no conflict and no coroutine switching. AG, RR, SF, etc. indicate that a conflict has occurred and the coroutine will be move to a specified position, including the front and tail of the Thread Queue ,Global Queue, and the Next Batch. So each concurrency protocol can control the callback cycle of the transactions.

We consider transaction control scheduling to be mandatory in the Transaction-to-Coroutine schema, in which concurrency is much bigger than parallelism, otherwise, it will bring cascading conflict. Such as OCC and MVCC, if the rerunning conflicting transaction calls back earlier than the other concurrent transactions, it will bring new conflicts with the started transactions since the conflicting transaction is reassigned to a max COMMITID (or Timestamp). The other sample is Calvin, BOHM, and PWV, which expect to execute following the order of initial TID as much as possible for the least conflict. Such that, We perfect the SGPM scheduling strategy (with six signals) to accommodate common concurrency protocols as shown in Table 1. These protocols can be executed correctly and reasonably within the SGPM framework.

We use the SGPM coroutine model to optimize the performance of wound-wait concurrency control, including pessimistic concurrency 2PL, SS2PL, and deterministic concurrency Calvin and PWV. Which describe in section 4. When a conflict occurs, the thread can switch to the next transaction instead of thread blocking or abort. The conflict transaction can be resumed at the conflict point when scheduled next.

Note that coroutine optimization is not recommended for no-wait and wait-die mechanism transactions, since the conflict rate for each run is relatively independent even with the same COMMITID. Therefore, there is no need to use coroutines to perform subsequent transactions ahead.

SGPM has an important parameter, Thread Queue Size, to control the max number of coroutine concurrency. Too large a Queue Size will cause the conflict rate to increase, and too small will cause a continuing conflict within a blocking cycle after being called back. The details are analyzed in subsection 3.4. In the experiment, the concurrency protocols and SGPM are implemented with golang [37]. From which we study the effect of different Thread Queue sizes on the optimization rate and select a reasonable size to optimize the four wound-wait concurrency control protocols. The YCSB benchmark results showed that 2PL and SS2PL brought a 2X increase in performance which surpass OCC and MVCC. The throughput of Calvin and PWV increased to 1.3x and 2.5x which outperform the state-of-the-art deterministic concurrency controls.

In summary we make the following contribution:

• We design and implement the SGPM coroutine model [37]. SGPM scheduling is controlled by the specific transaction protocol. In the transaction-to-coroutine schema, the SGPM model uses fewer threads than Golang's GPM, while achieving the identical degree of parallelism and a lower conflict rate.

• We adapt the practical concurrency control protocol to SGPM for making the scheduling mechanism complete.

• We use SGPM to optimize the pessimistic concurrency control SS2PL, S2PL, and deterministic concurrency control Calvin, PWV.

• We use YCSB for evaluation, the result shows that the four pessimistic concurrency control protocol all have an obvious performance improvement.

2. Background

In this section, we will introduce the basics covered in this article, including conflict handling mechanism and concurrency control.

2.1. Wound-Wait vs No-Wait vs Wait-Die

As Figure 3. In the wound-wait mechanism, conflict transactions are spin waiting until the dependence transaction grants access. The wound-wait method may bring circle dependence, so deadlock prevention should be adopted. The available method includes wait-for-graph circle detection, limit-wait-depth [17], and lock ordering. No-wait and wait-die break hold and wait deadlock conditions, but the statements that have been executed need to be repeated. When

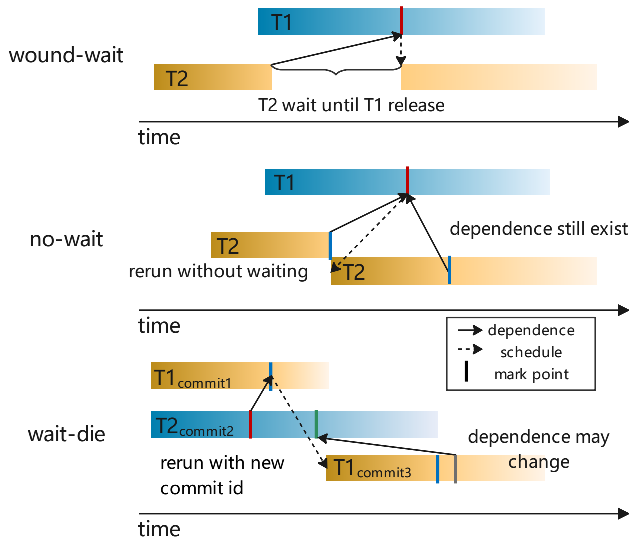


Figure 3: Examples of wound-wait, no-wait, and wait-die mechanisms in conflict handling.

conflict occurs wait-die is subject to reset the transaction with new COMMITID while no-wait just rerun the transaction. Such that the transaction with no-wait remains the dependency scenario of the previous run, and the transaction with wait-die is independent of the previous.

The final experiment shows that no-wait is better than wound-wait in high contention, otherwise, wound-wait is better.

2.2. PCC vs OCC vs DCC

The main idea of pessimistic concurrency control is to use preemption to achieve isolation. A typical serializable implement is two-phase lock which divides the transaction into two successive phases: locking and releasing. We can classify two-phase locking into general (default) 2PL and strong strict 2PL (SS2PL) depending on the lock release point as Figure 6. 2PL [5] allows lock release before commit, while SS2PL [14] must after commit. The disadvantage of pessimism is that the access detection are false positives while locking failure does not must cause a conflict. The advantages are simple implementation, and lower overhead under low contention, which we confirmed in the experiment.

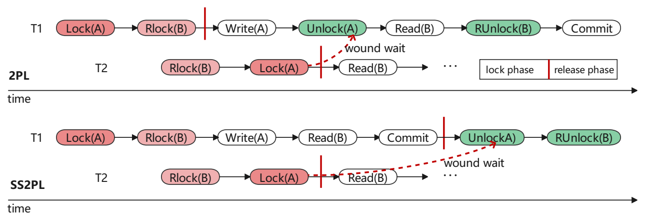


Figure 4: pcc, the difference of 2PL and SS2PL is the lock releasing started point.

Optimistic concurrency control uses validation instead of preemption. Validation can reduce unnecessary conflicts. OCC [24, 35] is read validation, while MVCC [39] writes validation. The bottleneck is that the cost of conflict is higher than PCC since it requires re-executing the conflicting transaction with new COMMITID (Timestamp). The advantage is to reduce the conflict rate. OCC generally performs better than the no-wait 2PL.

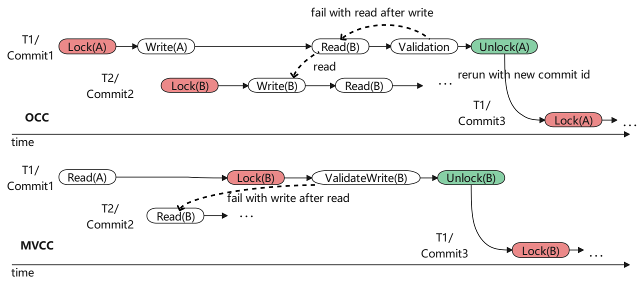


Figure 5: occ and mvcc, the validation-based concurrency control.

Deterministic concurrency control aim to have consistent results for the same transactions set. Calvin [33] initializes a lock manager for each record in the preparation phase and obtains record access in transaction order in the execution phase. BOHM [15] and PWV [16] are based on multi-version control, both in the first phase install batch's write versions with TID into the version chain but which are not available. In the second phase, the version of the BOHM is readable only after the writing transaction commits. PWV allows transaction uncommitted read but only commits after the writing transaction is done. The above concurrency control protocols can use the wound-wait mechanism. They are based on a priori transaction order and read-write-set to achieve determinism. Aria [27], without prior, executes transactions locally based on the consistent latest snapshot. In the commit or execution phase, that verifies the read-after-write and writes-after-write conflict, and moves the conflicting transactions to the next batch. Aria is classified as wait-die since the transaction order has changed after the conflict.

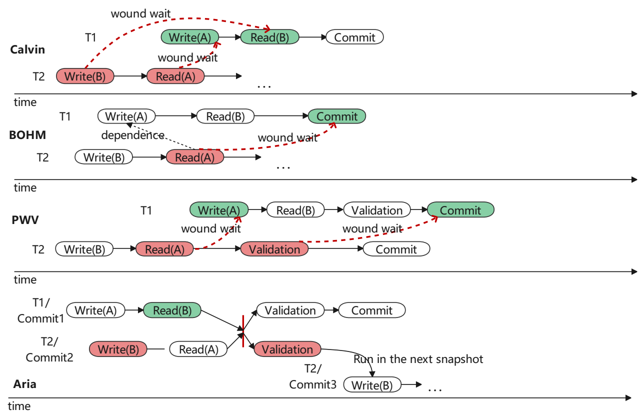


Figure 6: dcc, Calvin, BOHM, PWV, concurrent running in TID order, Aria which runs at one snapshot and moves conflicting transactions.

3. Coroutine architecture

The coroutine scheduling architecture as Figure 7 which consists of three parts. The left side of that is the **Batch Manager** which structured as a link list and executes on the main thread. The job of the main thread is to encapsulate the received transactions into batches and add them to the linked list. The CRUD of batch link list are atomic. Transactions of the same batch are executed concurrently, while different batches are executed sequentially.

Transaction Batch and **Global Queue** follow a one-to-one relationship and both of them can access each other. The Global Queue maintains a linked list of coroutines, each running one of the transactions. The Global Queue provides two public functions for the Batch Manager and Thread Queue, $InsertLast$ and $SwapFront$, which correspond with ML and SF signals we describe below. All operations on the Global Queue must be thread-safe.

Global Queue and **Thread Queue** follow a one-to-many relationship. Each Thread Queue allocates a worker thread responsible for coroutine running and scheduling. The size of the Thread Queue is specified by the user. When the worker thread detects that the number of coroutines in the own Queue is below a threshold, that will fetch the coroutines from the Global Queue.

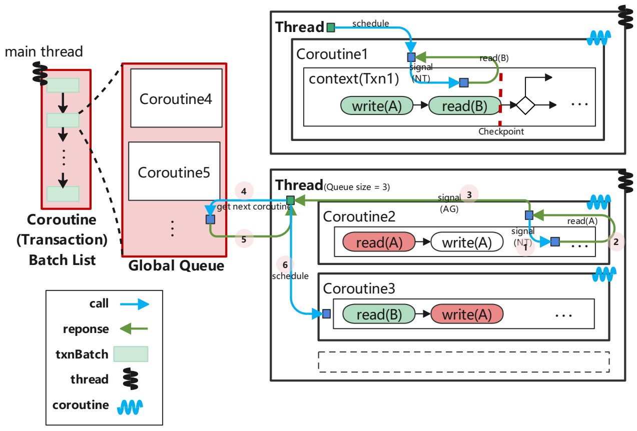


Figure 7: SGPM couroutine schedule model

3.1. Operation in Transaction Running

Transaction are run with operations **Read**, **Write**, **Reset**, **Commit** and **Init**. Each operation is a checkpoint for coroutines. The operation logic is specified in the concurrency control protocol. In this section, we will talk about how they work.

In operation of **Write**, 2PL, SS2PL, and OCC add a write lock to the record. 2PL Releases locks after holding all locks and reading. SS2PL and OCC release the write lock after transaction commits. If in no-wait, the all holding lock release as the conflict occurs. The MVCC verifies the read conflict by $version.rts > Timestamp$. Calvin [33] verifies the operation is in transaction order. BOHM and PWV write their own pre-installed version. Aria's read operation triggers validation for write-after-write conflicts and updates the write reserve.

In operation of **Read**, 2PL [5] and SS2PL [14] 2PL, SS2PL behaves like write, while replacing write locks with read locks. The OCC [24] verifies $record.wts < COMMITID$. The MVCC [39] finds the corresponding reading version in the version chain of record, which with $max(wts) \mid wts < TID$, and in state of committed. Calvin [33] verifies the operation as write. BOHM [15] and PWV [16] read the corresponding available version with committed and uncommitted respectively. Aria's [27] read operation triggers validation for read-after-write conflicts.

The **Reset** converts all intermediate states of the current transaction holding record to the final state and resets the context. In this operation, 2PL, SS2PL, OCC release all locks. The MVCC changes the state of all inserted versions to ABORT. The deterministic transaction has to do nothing in Reset since the transaction execution order is fixed and there is no resource hogging.

Commit occurs when the transaction completes. 2PL, SS2PL, and OCC Commit logic is the same as Reset. MVCC converts all inserted version states to COMMITED for other transaction read which same as BOHM and PWV. Because PWV supported serializable uncommit read. In the Commit operation, it spin verify all reading records state are committed. Aria needs to verify the write-after-write and read-after-write conflict in this operation.

The **Init** operation needs to be called before the execution. Init is idempotent which works only after Reset, or before first execution. so we can call Init in each time entering the coroutine run as algorithm 2 line 3. Init mainly serves for optimistic concurrency control like OCC and MVCC which must change COMMITID after the conflict in the validation. So after Reset, the Init needs to be called to assign the last COMMITID.

3.2. Coroutine structure

Each transaction is run by one coroutine. The coroutine structure is show in Figure 8. The coroutine class consists of four major parts. One is **Txn**, which is a interface class. All concurrency protocols must implement the five abstract methods of operation of TXN which are describe in subsection 3.1. Txn also has a Database pointer, which can be called for index searching or generating snapshots.

Context holds the execution state of the wrapping transaction. Each operation boundaries can act as a checkpoint of dynamic transaction execution. When the transaction needs to yield the thread due to a conflict, the last operation checkpoint will be saved. As the next wake up, the coroutine resumes running from the saved checkpoint.

Shift indicates the scheduling plan. The coroutine will move to a specific position according to the shift state while the schedule triggered. The shift state includes SF, ML, NB which correspond with the signals receiving we will describe below, and NONE, DONE indicates no dispatch and wait for removal respectively.

Table 1
Signals returned by each concurrency protocol

	(SS)2PL	SS2PL_NW	OCC_NW	MVCC	Calvin	Calvin_NW	BOHM	BOHM_NW	PWV	PWV_NW	Aria
NT	rwc	rwc	rwc	rwc	rwc	rwc	rwc	rwc	rwc	rwc	rwc
AG	rw			r	rw		rw		rwc	c	
RR		rw	w								
SF						rw		rw		rw	
ML			r	w							
NB											c

Note: [r] [w] [c]; which of them indicate the signal is returned by the operation of **Read**, **Write**, **Commit** respectively.

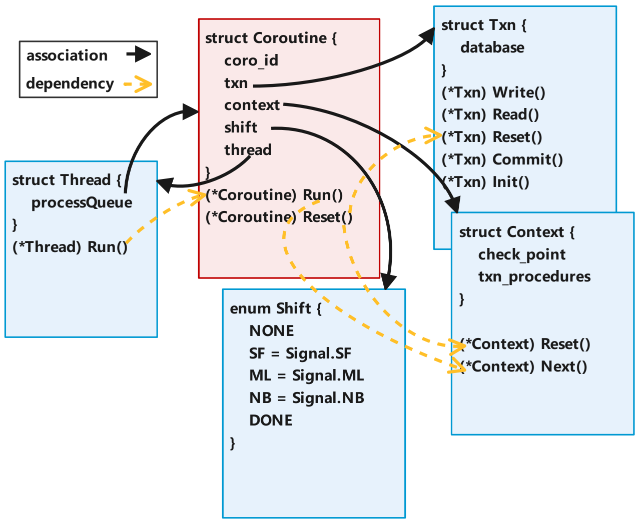


Figure 8: Couroutine class diagram

Algorithm 1: Thread Run

```
1  Function (t *Thread) Run():
2      while coroutine :← t.threadQueue.Schedule();
3      coroutine do
4          coroutine.SetThread(t)
5          sig :← coroutine.Run()
6          switch sig do
7              case NT do
8                  coroutine.Shift ← DONE; break
9              case RR do
10                 coroutine.Reset(); break
11             case SF or ML or NB do
12                 coroutine.Shift ← sig;
13                 coroutine.Reset();break
```

Thread are the entry points in which the main program executes transactions. There is a Queue in the thread that caches coroutines (transactions) that are assigned to this thread. Thread Queue also contains a pointer to Global Queue, with that the thread can get coroutines from the Global Queue since the worker threads are also responsible for the scheduling.

Coroutine and Thread both have function *run* and there is a calling relationship between them. The *run* in the Coroutine executes the transaction procedure in the Context from the last checkpoint, and when reaching the data access statement, it will call Txn's Operation (Write, Read, Reset, etc.) for execution. If the operation returns success (Signal NT), the *run* will continue. Otherwise, it returns a conflict Signal as algorithm 2 line 16. Based on the returned signal, Thread modifies the Shift of the Coroutine and obtains the next scheduled coroutine through own Queue as algorithm 3 line 10.

3.3. Coroutine scheduling with Signal

For each transaction operation, there is one signal feedback for the coroutine schedule. The type of signal received depends on the conflict handling strategy of the concurrency protocols as Table 1. When the thread receives the signal, it will change the coroutine execution route to achieve flexible resource scheduling as Figure 9.

Algorithm 2: Coroutine Run

```
1  Function (c *Coroutine) Run() Signal:
2      var sig Signal
3      c.txn.Init()
4      while command :← c.context.Get();
5      command do
6          switch command.type do
7              case Write do
8                  sig ← c.txn.Write(command.Key);
                    break
9              case Read do
10                 sig ← c.txn.Read(command.Key);
                    break
11             case Abort do
12                 return c.txn.Reset();
13         if NT = sig then
14             c.context.Next()
15         else
16             return sig
17     return c.txn.Commit()
```

The signal includes NT(Next), AG(Again), RR(Rerun), SF(Swap with the Front of Global Queue), ML(Move to the last of the Global Queue), and NB(Move to next batch), where NT, AG, and RR are scheduling within transactions and the others are scheduling between transactions.

When coroutine receive **NT** which means the operation is passed and can be continued. NT signals are normally processed inside coroutine *run* as algorithm 2 line 14 unless they are returned from the operation of Commit which handle in algorithm 1 line 8. All concurrency control protocols can return NT signals.

AG means there is conflict happened during the operation. Concurrency protocol that adopts wound-wait must return AG signal after operation Read or Write. Besides, Commit operation may also feedback with AG in PWV, which cannot pass commit until all write transactions of reading version are committed. As algorithm 1 the coroutines that return AG signals do not need to do anything and just continue execution in the next schedule.

RR means the transactions need to be executed from scratch. This is a conflict handling strategy for the non-deterministic concurrency control protocol with no-wait (_NW in Table 1) mode. Deterministic concurrency without the case of deadlocks and the transaction order is fixed, so rerun is unnecessary. However, if the Thread Queue's size is 1 (no coroutine) when an operation is stuck for a conflict with a previous long transaction, there is no way to execute subsequent transactions until the current transaction commits. Here we employ **SF** signal. The no-wait mode in the deterministic transaction will return a SF signal which swaps the currently executing transaction with the first transaction in the Global Queue. If the blocking happened again, the first transaction will be rescheduled.

In the case of Thread Queue size greater than 1, if the OCC and MVCC are scheduled with RR when validation conflicts occur, transactions already in Thread Queue could have a massive conflict due to the conflicting transaction rerun with newly allocated max COMMITID. So it's necessary to move the conflicting transaction out of the Queue by signal **ML**, for concurrent running in COMMITID order as much as possible.

The **NB** signal moves the current transaction to the next Batch. NB mainly serves Aria [27] which executes transactions of the same batch based on one snapshot. So the conflicting transaction must be rerun in the next batch (snapshot).

3.4. Effectiveness Analysis

In this section, we analyze how the Thread Queue size affects the performance. We suppose that the record access rate is w; thread count is n; Thread Queue size is q; Transaction rest of execution length is l, coroutine switch spend time is s; duration of each operation is d, where w, l, s and d all follow normal distribution. Such that, we can get the record conflict rate by C as Equation 1, i.e., at least one of other concurrent transactions are accessing that record.

$$C(w, n, q) = 1 - (1 - w)^{nq-1} \tag{1}$$

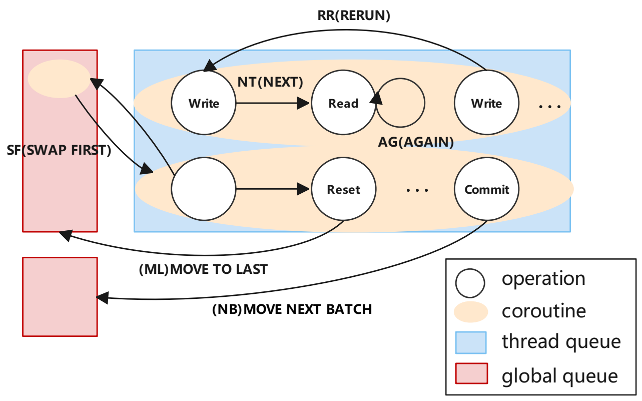


Figure 9: Signal and scheduling policies

Algorithm 3: Thread Queue Schedule

```
1  Function (tq *ThreadQueue) Schedule() *Coroutine:
2      while true do
3          if ¬tq.Full() ∧ ¬tq.globalQueue.Empty() then
4              tq.InsertLast(tq.globalQueue.Next())
5          if tq.Empty() then
6              return ∅
7          if ∅ = tq.curCoro then
8              tq.curCoro ← tq.Front()
9          preCoro :← tq.curCoro
10         tq.curCoro ← tq.Next()
11         switch preCoro.Shift do
12             case DONE do
13                 tq.Remove(preCoro); break
14             case SF do
15                 firstCoro :←
                     tq.globalQueue.SwapFront(preCoro)
16                 tq.InsertLast(firstCoro)
17             case ML do
18                 tq.globalQueue.InsertLast(preCoro)
19             case NB do
20                 tq.batchManager.InsertNext(preCoro)
21         preCoro.Shift = NONE
22         tq.Remove(preCoro)
23         if tq.curCoro ≠ ∅ then
24             return tq.curCoro
```

The running time of each scheduled transaction D as Equation 2. $D(d, C)$ is an expectation value; Each operation's duration is multiplied by its probability of being interrupted. So the call back duration after conflict is $D'(q, D) = (q - 1) \times D(d, C)$.

$$D(d,C) = \sum_{i=1}^{l} d \times C(w,n,q)^i \quad (2)$$

We assume that the conflict disappear only after the dependent transaction completes. The wound (blocking) duration after conflict is $B(l,d,q,n,w)$. The dependence of B is complicated, and all parameters related to the conflict rate will affect B, where $\frac{\sigma B}{\sigma l} > 0$, $\frac{\sigma B}{\sigma d} > 0$, $\frac{\sigma B}{\sigma q} > 0$, $\frac{\sigma B}{\sigma n} > 0$, $\frac{\sigma B}{\sigma w} > 0$.

The loss is the coroutine switch cost by the conflict as Equation 3. s is a constant, and the size relationship between B and D' becomes the key to the loss. The decrease of q will have a positive impact on both B and D', and how to minimize the loss function? We will find an appropriate q through experiments to local improve the performance.

$$Loss(s,q,B,D') = s \times q \times (\lceil \frac{B}{D'} \rceil) \quad (3)$$

4. Experiment

In this section, we will make the evaluation of SGPM model in YCSB [9]. Because deterministic transactions have the additional overhead of ensuring consistent results, the comparing experiments separate deterministic and non-deterministic concurrency protocols. the SGPM and comparing concurency control protocols all implement in golang [37] with version 16.6. The server we use is equipped with Intel(R) Core(TM) i5-9500 CPU with 16G memory RAM. The operating system is CentOS release 6.5 (Final).

4.1. Non-deterministic concurrency control

The non-deterministic concurrency protocols in the experiment include 2PL, SS2PL, OCC, and MVCC. The _NW suffix indicates implementation by the no-wait mechanism. We run the experiment through high contention and low contention. First, we compare the raw performance, then we study the relationship between Queue size and performance, and finally optimize 2PL and SS2PL with the selected Queue size.

4.1.1. high contention

In this workload, the raw performance of non-deterministic transactions is shown as Figure 10a. OCC running in the no-wait mode shows higher performance, keeping the first place until 16 threads and reaching the peak performance of 33 MTPS at 4 threads. SS2PL_NW is in the second place, which has a similar curve to OCC, peaking at 25 MTPS at 4 threads. The third is MVCC, which fluctuates more than others, because even little write conflicts will lead to great scheduling and validation overhead. Next comes SS2PL, which peaks at 2 threads and crash at 8 threads. The last one is 2PL which has the best performance single thread, but the declining trend is lower than SS2PL.

Figure 10c shows the number of RR and ML signals, which only happened in transactions with no-wait mode and MVCC. We can see that the conflicts count in both 2PL and SS2PL increases dramatically after 12 threads. MVCC saw a soar up of conflict after 16 threads which leading to the throughput crash in Figure 10a.

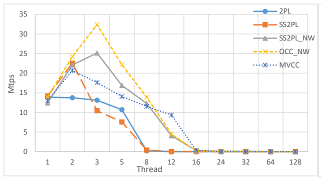
The number relationship between AG signals and threads is shown in Figure 10d. For 2PL and SS2PL, the amount of conflict increases sharply at 8 threads, corresponding to the performance plunge shown in Figure 10a. MVCC returns an ML signal for writing conflicts and an AG signal for reading conflicts. We can see that read conflict start to climb after 16 threads and the TPS drops to below 30K.

We use the SGPM model to improve 2PL firstly. Figure 11a shows TPS fluctuations as the size of Queue increases in the case of 3 threads. We can see the periodic ups and downs in TPS. The 2PL outperformed the first baseline OCC in Queue size at 2, 9, and 15, and peaked at 37.3 M at the size of 15. Performance in most cases is better than MVCC and SS2PL. Figure 14a line of 3t show the optimization rate against the 2PL baseline. We can see that the optimization rate is mainly between -50% and 150% and reaches 180% at the Queue size of 15. But also a small number of cases have a negative impact on performance.

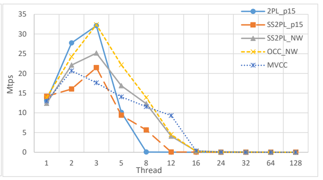
The Figure 11b shows the performance in 8 threads which is a point that most concurrency protocols start to have a sharp conflict increase. The OCC still reached 17.2 M in TPS with ranked first. At Queue size 22, 2PL overtakes the SS2PL_NW which is in the second rank. Although the performance of 2PL fluctuates between SS2PL and MVCC, it still maintains a high optimization rate. Figure 14a shows that there is no negative impact on performance at 8 threads, indicating that coroutines can alleviate performance crashes in high contention.

SS2PL is similar to 2PL. In the case of 3 threads as Figure 11c, at Queue size is 9, SS2PL performance reaching 33.2 MTPS and exceeds OCC. The Figure 14b shows that the optimization rate is over 200% when Queue size in 2,3 and 9, and peaking at 218% at size 9. In 8 thread, the performance of SS2PL is decrease to between 1 and 7 MTPS as Figure 11d. The optimize rate of SS2PL which is top at 0.8 is lower than 2PL in 8 thread. However, SGPM had no negative impact on SS2PL performance with a minimum optimization rate of 0.8%.

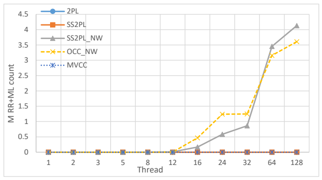
The optimization comparison results of nondeterministic pessimistic transactions under the high contention workload are shown in Figure 10b. Comparing with Figure 10a, the throughput of 2PL climbed from the last to the first. 2PL surpassed the OCC with 27.7 MTPS at 2 threads and peaked at 3 threads with 32.3 MTPS. The performance crash happened in 8 threads like the original 2PL. The optimization effect of SS2PL is not obvious as 2PL, and the peak is almost flat with the original at about 22 MTPS. However, the performance decline eased, with 5.6 MTPS at 8 threads and crash at 12 threads, while original SS2PL on 8 threads has down to 0.4 MTPS by large conflict.



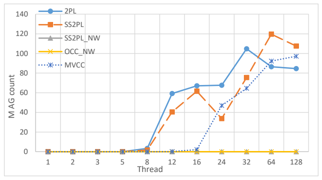
(a) raw throughput(Mtps)



(b) 2PL, SS2PL optimized in 15 q_size

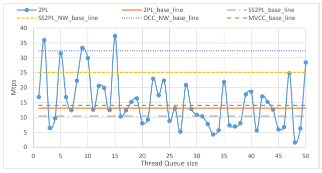


(c) raw RR and ML count

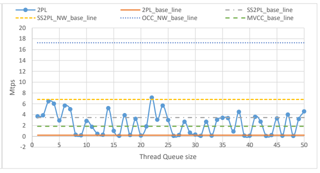


(d) raw AG count

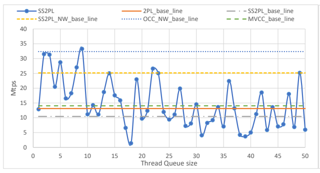
Figure 10: non-deterministic concurrency protocols throughput and conflict count in **high contention**



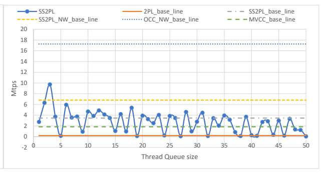
(a) 2PL optimized in 3 thread



(b) 2PL optimized in 8 thread



(c) SS2PL optimized in 3 thread



(d) SS2PL optimized in 8 thread

Figure 11: non-deterministic wound-wait concurrency protocols throughput vs queue_size in **high contention**

4.1.2. low contention

. The performance of raw nondeterministic concurrency control in low contention is shown in Figure 12a. In this scenario, the MVCC showed good performance, reaching a throughput of 202 MTPS at 16 threads. From Figure 12c and Figure 12d, we can also see that the MVCC had only a few reading conflicts, which happened when the reading version in uncommitted. The second one is 2PL which with an isolation level of uncommitted reads, so the read/write conflicts are relatively rare. The third is SS2PL with 137 MTPS in 16 threads and in 24 threads got a performance crash with conflict increase. 2PL and SS2PL have an exponential increase in conflict after 16 threads. The poor performance of OCC_NW and SS2PL_NW in low contention workload since the no-wait mode transactions have more conflict costs by re-executing the entire transaction. From Figure 12c and Figure 12d, we can see that OCC_NW and SS2PL_NW have conflict accretes in 12 threads, while SS2PL and 2PL are in16 threads.

The low contention coroutine optimization effect is shown in Figure 13a Figure 13b and Figure 14c. The first graph shows the TPS with the Queue size increase. We can see that the improved 2PL is up to MVCC performance, and the throughput fluctuation is mainly concentrated between 202 MTPS of MVCC and 137 MTPS of SS2PL. But there are also a few cases of performance degradation. At Queue size 27, the 2PL decreased to 22 MTPS of SS2PL_NW and approached OCC's 35 MTPS at size 41. And from the Figure 14c, we can find that 2PL has a low optimization rate when its throughput is high, therefore the SGPM even brings the negative effect in most cases.

When the thread reach 24 as Figure 13b and Figure 14c 24t line. SGPM can achieve a high optimization rate, which is mainly distributed between 500% and 1000%. At Thread Queue size 5, the throughput of 2PL reaching 185 MTPS exceeds that of the first baseline MVCC.

The optimization performance of SS2PL with low contention is shown in Figure 13c, Figure 13d and Figure 14d. SS2PL has a similar performance curve to 2PL. SS2PL's best performance is close to MVCC at 16 threads, and there is also a small negative performance effect exists. 90% of the throughput is concentrated between 100 and 200 MTPS. SS2PL showed a good improvement in 24 threads with the optimization rate basically greater than 50%, while the optimization rate of 16 threads was mainly concentrated between 0 and 0.5.

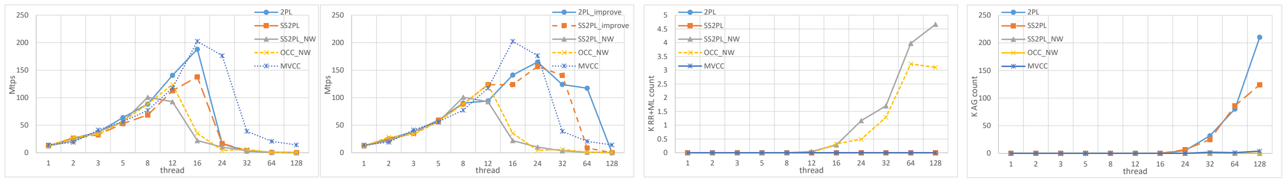
The optimized comparison of 2PL and SS2PL under low contention is shown in Figure 12b which with Thread Queue sizes of 38 and 16. By comparing with Figure 12a, we can see that the peaks value of 2PL and SS2PL are basically not increasing while the peak condition changed from 16 thread to 24 thread. The crash point increased from 24 and 32 threads to 64 and 128 threads. Thus the SPGM can mitigate the negative of low concurrency and high conflict.

4.1.3. Conclusion

From the above experiment in non-deterministic concurrency control, we can figure out that the SGPM coroutine model has the ability to reduce conflict cost. In the high contention workload, it can improve the upper limit of the throughput. In the low contention scenario, it can maintain performance stability.

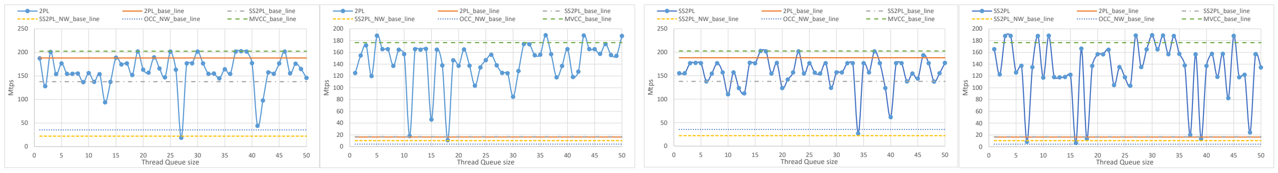
4.2. Deterministic concurrency control

Deterministic concurrency control include Calvin [33], BOHM [15], PWV [16] and Aria [27]. Except for Aria we implement the no-wait and wound-wait mode for each. Because the PWV and BOHM both use MVCC and have similar performance, we just improve the Calvin and PWV in this evaluation.



(a) raw throughput(Mtps) (b) 2PL, SS2PL optimized in 38 and 16 q_size (c) raw RR and ML count (d) raw AG count

Figure 12: non-deterministic concurrency protocols throughput and conflict count in **low contention**



(a) 2PL optimized in 16 thread (b) 2PL optimized in 24 thread (c) SS2PL optimized in 16 thread (d) SS2PL optimized in 24 thread

Figure 13: non-deterministic wound-wait concurrency protocols throughput vs queue_size in **low contention**

4.2.1. high contention

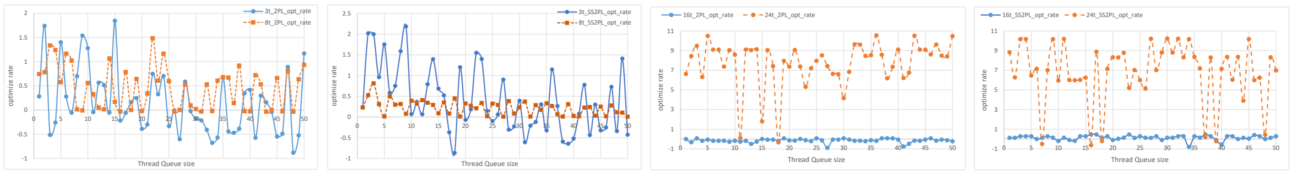
The raw performance of deterministic transactions under high contention is shown in Figure 15a. From which we can figure out that the Calvin got a high throughput with seldom of conflict happened by AG signal as Figure 15d which peaks at 16 thread reaching 30 MTPS. The second one is PWV_NW and PWV both of which got a similar throughput because in this workload the wound-wait circle is close to the redo circle. The next is Aria, which peaks at 16 thread with 20 MTPS. Aria has the least effect by thread increase since the conflict count is deterministic. BOHM with reading commit is slightly worse than PWV. We can also find that BOHM and PWV are more likely to conflict than Calvin and Aria.

As show in Figure 15c, PWV_NW conflict starts to increase exponentially at 8 threads, in 128 threads is up to 13000 M which is not shown in the graph. BOHM_NW conflict start from 5 threads, which leveled off between 8 and 32 threads and skyrocketed after 32 threads. Calvin_NW's overall throughput is not high, but the conflicts count remains steady and surpasses BOHM and PWV after 16 threads. Ranging from 1 to 128 threads, Aria has a stable amount of conflict which between 0.7M and 1.2M. Unlike other transaction protocols, Aria has a fixed conflict count, so it can maintain great performance in high-contention environments.

Figure 15d shows the AG signal count. The BOHM and PWV are the highest, which increased exponentially beginning at 8 threads and 16 threads respectively, and the number of conflicts is 200M at 128 threads. PWV_NW not only has the SF signal but also has the AG signal, which happened at verifying reading version commit failure in the last phase. The AG conflict amount of PWV_NW began to logarithmically grow at 16 threads. Calvin's conflicts count maintained less than 20 until 64 threads, and at 128 threads the conflict soared to 24M which is level off the peak of PWV_NW.

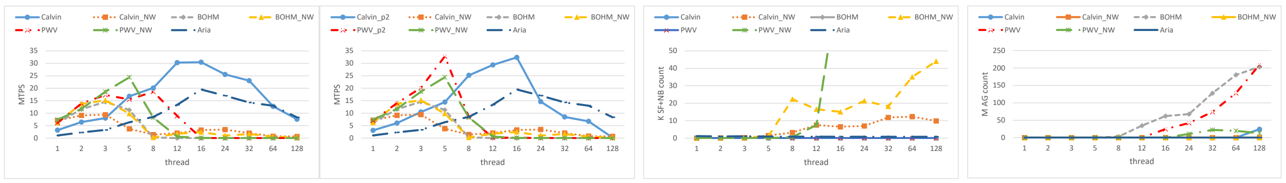
Calvin's optimization effect by SGPM is poor, as shown in Figure 19a. Both 8 threads and 128 threads under high contention have a negative impact. The highest throughput is the no coroutine mode when the Queue size is 1. The reason is that Calvin is executed sequentially in concurrency. Running subsequent transactions in Thread Queue while the previous one is not committed will get a higher conflict rate. So we get worse performance.

The PWV improve performance is shown in Figure 19b. The optimization effect of PWV at 8 threads is bad as Calvin's, but the optimization rate peaked at 24% when Queue size is 2. At this point, throughput reached 24 MTPS, surpassing Calvin, which ranked first as Figure 16c. In the 16 thread experiment as Figure 16d and Figure 19b 16t, PWV showed an ideal performance improvement. The



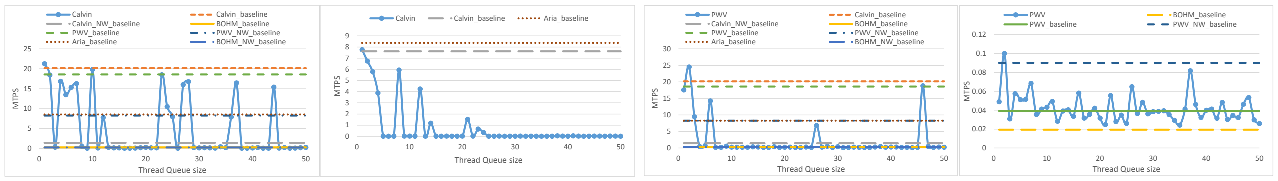
(a) 2PL in **high contention** (b) SS2PL in **high contention** (c) 2PL in **low contention** (d) SS2PL **low contention**

Figure 14: optimize rate of wound-wait **non-deterministic** concurrency protocols



(a) raw throughput (Mtps) (b) Calvin and PWV optimize with q_size 2 (c) raw SF and NB count (d) raw AG count

Figure 15: deterministic concurrency protocols throughput and conflict count in **high contention**



(a) Calvin optimized in 8 thread (b) Calvin optimized in 128 thread (c) PWV optimized in 8 Thread (d) PWV optimized in 16 Thread

Figure 16: deterministic wound-wait concurrency protocols throughput vs queue_size in **high contention**

optimization rate is mainly between -20% and 100%. When Thread Queue is 2, that reaches the peak at 150% and the performance exceeds PWV_NW. Because Aria and Calvin still reach 20 and 30 MTPS with a low conflict level even in 16 threads, they are not compared in the figure.

Based on the above experiments, as Figure 15b we set Queue size to 2 to observe the optimization effect of Calvin and PWV. Calvin gets a slight improvement from 5 threads to 16 threads, with a maximum of 30 MTPS in 12 threads going up to 32 MTPS in 16 threads. But, performance degrades after 24 threads and drops below 1 MTPS at 128 threads. The optimization effect of PWV is obvious. The max throughput rises from 18 MTPS in 8 threads to 33 MTPS in 5 threads, surpassing Calvin in first place. But, it also brings the higher conflict cost. The half of performance decrease in 8 thread where is the peak in raw PWV. And a performance crash happened in 12 threads earlier than the raw one in 16 threads.

4.2.2. *low contention*

The deterministic transaction origin performance in low contention is shown in Figure 17a. Aria remains a performance leader in this scenarios, with throughput up to 128 MTPS at 16 threads, and the number of conflicts always stays low. BOHM_NW and PWV_NW tied for second place, followed by Calvin_NW. We can figure out that the performance of no-wait mode is better than that of wound-wait mode under low contention. This is because no-wait can reduce the probability of conflict but increase the conflict cost. In low contention, the conflict amount is little and hardly any negative effect. Calvin, BOHM, and PWV have lower performance, with maximum throughput in 24,51 and 35 MTPS respectively. From Figure 17d, we can see that BOHM and PWV's conflict count grow slowly at 8 threads and exponentially increase at 24 threads while Calvin has always maintained a low level of conflict with low throughput.

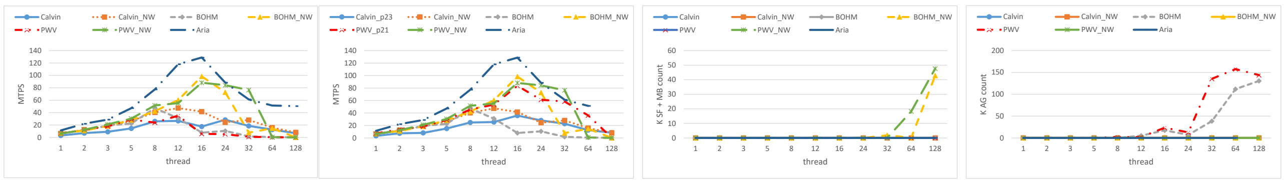
From Figure 18a and Figure 19c, we can see the throughput of Calvin in 12 thread relation with the Queue size. Aria_baseline is greater than 90 MTPS and is not shown in the graph. Calvin's optimization rate is concentrated between 0% and 35 %, exceeding the BOHM_baseline and approaching the PWV_NW_baseline.

The optimization of Calvin under 128 threads is shown in Figure 18b and Figure 19c. The overall improvement is slightly better than which in 12 threads. The optimization rate is concentrated between 0% and 50%. In most cases, it exceeded the second-ranked Calvi_NW_baseline. Calvin peaks 23 MTPS at the Queue size of 23. Since BOHM and PWV are less than 1 MTPS at 128 threads and Aria is 58 MTPS, these outliers are not added in the comparison.

As Figure 18c and below of Figure 19d, PWV optimization rates are concentrated between 50% and 150% at 12 threads. PWV's optimized performance is better in most cases than the second-ranked BOHM_NW. There is a peak of the optimized rate at 130% with the 100 MTPS throughput after improvement.

At 32 threads which are shown in Figure 18d and the top of Figure 19d, the PWV can achieve excellent improved. In most cases, the optimization rate exceeded 500%. Throughput peaked at 100 MTPS when Queue size is 26, that performance exceeds the number one PWV_NW, while the Aria drops to the second place.

Figure 17b shows the optimization of Calvin and PWV. The Thread Queue sizes are configured by 23 and 21 respectively. From the graph, we can see that Calvin does not have much improvement. The peak value rose from 21 MTPS with 12 threads to 41 MTPS with 16 threads. The improvement of PWV is more obvious. At 16 threads, it reaches a peak of 84 MTPS, which is close to PWV_NW and BOHM_NW. And it can still maintain a high level of throughput when the thread is 64. This shows that SGPM has an anti-conflict ability.



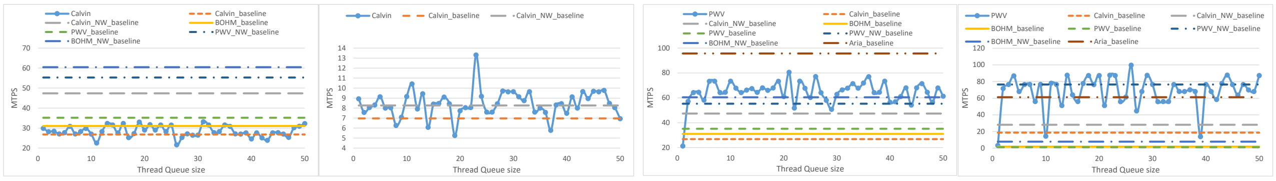
(a) raw throughput (Mtps)

(b) Calvin, PWV optimize with 23 and 21 q_size

(c) raw NB and SF count

(d) raw AG count

Figure 17: deterministic concurrency protocols throughput and conflict count in **low contention**



(a) Calvin optimized in 12 thread

(b) Calvin optimized in 128 thread

(c) PWV optimized in 12 thread

(d) PWV optimized in 32 thread

Figure 18: deterministic wound-wait concurrency protocols throughput vs queue_size in **low contention**

4.2.3. Conclusion

Calvin reaches peak performance with low conflict cost, so it has less room for optimization. PWV shows a significant throughput increase by adopting SPGM. The maximum throughput by 80% during high contention and increases by 150% in low contention, and the performance remains stable when the conflict increases.

5. Related work

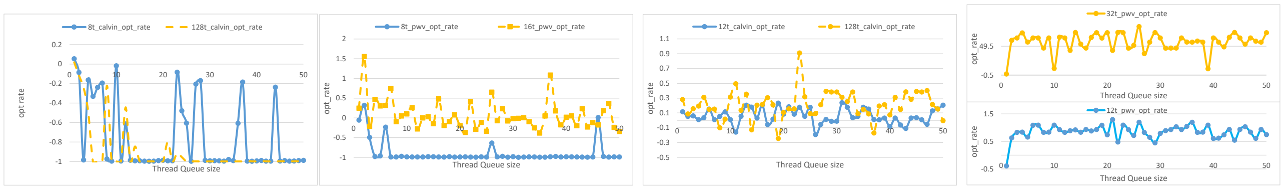
The coroutine optimization in this paper is based on previous studies. Areas covered the coroutines implement and apply, nondeterministic and deterministic concurrency control.

The **coroutine** is designed for the C10K problem [26]. If using the client-to-thread model, the server 10K clients overpass the limit thread count and can only solve by IO multiplexing [6]. With coroutines, the developer can just focus on the application instead of thread scheduling. According to the coroutine implementation, we can divide them into stacked coroutines and stackless coroutines.

The **stacked coroutine** is similar to the thread which holds the call information on the stack. As a coroutine is created, stack memory is allocated to hold the context of the coroutine, and if a child coroutine is created, it continues to be pushed to the stack. The feature of the stacked coroutine is that users do not need to worry about the scheduling. The disadvantage is that it is difficult to allocate a reasonable stack size. The state-of-art languages that use stacked coroutines are Lua [11] and Golang [19].

Stackless coroutines are used like functions. which store coroutine local variable in heap memory. Since there is no stack, subroutines can be woken up anywhere in the program. The stackless coroutine is characterized by the low overhead of coroutine switching. The disadvantage is that coroutine scheduling is unbalanced and users need to intervene in the scheduling. Popular languages such as C++ [2], Rust [18], Kolint [13] and Python [34] use stackless coroutines. The SGPM model in this paper is also stackless.

The latest research employ coroutines for DBMS since which can efficiently utilize the CPU resources in thread waiting. Corobase [20] uses the transaction-to-coroutine model to take full advantage of data prefetch waiting times.The Interactive Transaction Process [44] introduces SQL-to-Coroutine, which separates transactions into SQL, and each SQL is executed by a single Coroutine. The goal is to reduce the waiting overhead of transactions and networks, with the same point as this paper, but it uses a C++-based stackless coroutine and applies only to Hekaton [12]. FaSST [21] and



(a) Calvin in high contention

(b) PWV in high contention

(c) Calvin in low contention

(d) PWV in low contention

Figure 19: optimize rate of wound-wait **deterministic** concurrency protoco

Grappa [28] use C++ coroutines to hide the RDMA network latency of distributed transactions.

In addition to the **nondeterministic transactions** involved in the experiment, there are some commonly used variants of optimistic concurrency protocols. Silo [36] and Tictoc [41] generate usable CommitID at the validation phase based on the pre-loaded version and current index version, which are also known as 1VCC [25]. ERIMA [22] flexible controls the isolation level to accommodate different contention transaction requests. Cicadas [25] and Hekaton [12] both use multi-version concurrency control. Cicadas leverages optimistic and multiple synchronized clocks to reduce the overhead of different contention. Hekaton proposed a latch-free data structure to avoid physical interference.

Deterministic transactions are attracted attention by the low cost of distributed communication. Besides the deterministic concurrency protocols mentioned above, Lynx [42], a geographically distributed database that splits transactions into hops, each hop executed deterministically on the server to reduce distributed latency. VoltDB [8] is a commercial deterministic database based on H-Store [31] which divided the database into partitions, and each partition performs transactions without interruption. Hyder [3] sends logs consistently to all partitions, which decide commit or abort based on the last committed snapshot and logs to achieve deterministic commit, which is similar to Aria [27].

6. Conclusion

In this paper, we propose a coroutine model SGPM to optimize wound-wait concurrency controls. The optimization targets include pessimistic concurrency 2PL, SS2PL, and deterministic concurrency Calvin and PWV. SGPM's scheduling strategy is controlled by the signal returned by specific transaction operations. We adapted SGPM to the commonly used concurrency control protocols for perfecting schedules. The final experimental show that SGPM can optimize the performance of 2PL and SS2PL to exceed OCC and MVCC in different contention scenarios. It also improved PWV performance by 170% to surpass other state-of-the-art deterministic protocols in high contention. And we also find that SPGM has the ability to mitigate performance crashes with high conflict.

A. Supplementary information

The source code is available in `github.com/WastonYuan/sgpm`.

References

[1] AB, M., 1995. Mysql. URL: `github.com/mysql/mysql-server`.

[2] Belson, B., Xiang, W., Holdsworth, J., Philippa, B., 2020. C++ 20 coroutines on microcontrollers—what we learned. IEEE Embedded Systems Letters 13, 9–12.

[3] Bernstein, P.A., Reid, C.W., Das, S., 2011. Hyder-a transactional record manager for shared flash., in: CIDR, pp. 9–20.

[4] Bernstein, P.A., Shipman, D.W., Rothnie Jr, J.B., 1980. Concurrency control in a system for distributed databases (sdd-1). ACM Transactions on Database Systems (TODS) 5, 18–51.

[5] Bernstein, P.A., Shipman, D.W., Wong, W.S., 1979. Formal aspects of serializability in database concurrency control. IEEE Transactions on Software Engineering , 203–216.

[6] Brackett, C.A., 1990. Dense wavelength division multiplexing networks: Principles and applications. IEEE Journal on Selected areas in Communications 8, 948–964.

[7] Carlson, J., 2013. Redis in action. Simon and Schuster.

[8] Colton, A., 2009. Volddb. URL: `https://www.voltdb.com/`.

[9] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R., 2010. Benchmarking cloud serving systems with ycsb, in: Proceedings of the 1st ACM symposium on Cloud computing, pp. 143–154.

[10] Damien Katz, J.L., 2005. Apache couchdb. URL: `github.com/apache/couchdb`.

[11] De Moura, A.L., Rodriguez, N., Ierusalimschy, R., 2004. Coroutines in lua. Journal of Universal Computer Science 10, 910–925.

[12] Diaconu, C., Freedman, C., Ismert, E., Larson, P.A., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M., 2013. Hekaton: Sql server's memory-optimized oltp engine, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 1243–1254.

[13] Elizarov, R., Belyaev, M., Akhin, M., Usmanov, I., 2021. Kotlin coroutines: design and implementation, in: Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, pp. 68–84.

[14] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L., 1976. The notions of consistency and predicate locks in a database system. Communications of the ACM 19, 624–633.

[15] Faleiro, J.M., Abadi, D.J., 2014. Rethinking serializable multiversion concurrency control. arXiv preprint arXiv:1412.2324 .

[16] Faleiro, J.M., Abadi, D.J., Hellerstein, J.M., 2017. High performance transactions via early write visibility. Proceedings of the VLDB Endowment 10.

[17] Franaszek, P.A., Haritsa, J.R., Robinson, J.T., Thomasian, A., 1993. Distributed concurrency control based on limited wait-depth. IEEE Transactions on Parallel and Distributed Systems 4, 1246–1264.

[18] Graydon Hoare, M., 2010. rust. URL: `rust-lang.org`.

[19] Griesemer, R., 2009. go. URL: `https://golang.org/`.

[20] He, Y., Lu, J., Wang, T., 2020. Corobase: coroutine-oriented main-memory database engine. arXiv preprint arXiv:2010.15981 .

[21] Kalia, A., Kaminsky, M., Andersen, D.G., 2016. Fasst: Fast, scalable and simple distributed transactions with two-sided ({RDMA}) datagram rpcs, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 185–201.

[22] Kim, K., Wang, T., Johnson, R., Pandis, I., 2016. Ermia: Fast memory-optimized database system for heterogeneous workloads, in: Proceedings of the 2016 International Conference on Management of Data, pp. 1675–1687.

[23] Kung, H.T., Robinson, J.T., 1981. On optimistic methods for concurrency control. ACM Transactions on Database Systems (TODS) 6, 213–226.

[24] Larson, P.Å., Blanas, S., Diaconu, C., Freedman, C., Patel, J.M., Zwilling, M., 2011. High-performance concurrency control mechanisms for main-memory databases. arXiv preprint arXiv:1201.0228 .

[25] Lim, H., Kaminsky, M., Andersen, D.G., 2017. Cicada: Dependably fast multi-core in-memory transactions, in: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 21–35.

[26] Liu, D., Deters, R., 2008. The reverse c10k problem for server-side mashups, in: International Conference on Service-Oriented Computing, Springer. pp. 166–177.

[27] Lu, Y., Yu, X., Cao, L., Madden, S., 2020. Aria: a fast and practical deterministic oltp database. Proceedings of the VLDB Endowment 13, 2047–2060.

[28] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., Oskin, M., 2015. Latency-tolerant software distributed shared memory, in: 2015 {USENIX} Annual Technical Conference ({USENIX} {ATC} 15), pp. 291–305.

[29] Rosenkrantz, D.J., Stearns, R.E., Lewis, P.M., 1978. System level concurrency control for distributed database systems. ACM Transactions on Database Systems (TODS) 3, 178–198.

[30] Sivasubramanian, S., 2012. Amazon dynamodb: a seamlessly scalable non-relational database service, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 729–730.

[31] Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P., 2018. The end of an architectural era: It's time for a complete rewrite, in: Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker, pp. 463–489.

[32] Tay, Y.C., Goodman, N., Suri, R., 1985. Locking performance in centralized databases. ACM Transactions on Database Systems (TODS) 10, 415–462.

[33] Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J., 2012. Calvin: fast distributed transactions for partitioned database systems, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 1–12.

[34] Tismer, C., 2000. Continuations and stackless python, in: Proceedings of the 8th international python conference.

[35] Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S., 2013a. Speedy transactions in multicore in-memory databases, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 18–32.

[36] Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S., 2013b. Speedy transactions in multicore in-memory databases, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 18–32.

[37] Wang, X., 2021. Sgpm. URL: `https://github.com/WastonYuan/sgpm`.

[38] Weissman, T., 1998. Bugzilla. URL: `github.com/bugzilla/bugzilla`.

[39] Wu, Y., Arulraj, J., Lin, J., Xian, R., Pavlo, A., 2017. An empirical evaluation of in-memory multi-version concurrency control. Proceedings of the VLDB Endowment 10, 781–792.

[40] Yu, X., Bezerra, G., Pavlo, A., Devadas, S., Stonebraker, M., 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores .

[41] Yu, X., Pavlo, A., Sanchez, D., Devadas, S., 2016. Tictoc: Time traveling optimistic concurrency control, in: Proceedings of the 2016 International Conference on Management of Data, pp. 1629–1642.

[42] Zhang, Y., Power, R., Zhou, S., Sovran, Y., Aguilera, M.K., Li, J., 2013. Transaction chains: achieving serializability with low latency in geo-distributed storage systems, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 276–291.

[43] Zhou, S., Mu, S., 2021. Fault-tolerant replication with pull-based consensus in mongodb., in: NSDI, pp. 687–703.

[44] Zhu, T., Wang, D., Hu, H., Qian, W., Wang, X., Zhou, A., 2018. Interactive transaction processing for in-memory database system, in: International Conference on Database Systems for Advanced Applications, Springer. pp. 228–246.