EasyChair Preprint

№ 12766

Gas-Based Deterministic Concurrent Transaction Processing in Blockchain

Xinyuan Wang, Yun Peng, Xingchen Li and Hejiao Huang

March 27, 2024

Gas-Based Deterministic Concurrent Transaction Processing in Blockchain

Xinyuan Wang[1][0000−0001−6359−7869], Yun Peng[2], Xingchen Li[1], and Hejiao Huang[1][0000−0002−2030−957X]∗

[1] Harbin Institute of Technology
huanghejiao@hit.edu.cn
[2] Guangzhou University

Abstract. Blockchain enforces transactions to be processed deterministically in each node to achieve consistency. Previous research explores concurrency control algorithms from deterministic databases to improve parallelism for blockchain. However, they do not take advantage of an inherent property of blockchain transactions - the gas fee, which is paid for the energy consumed during transaction execution. Ideally, the gas fee is proportional to the execution time.

In this paper, we propose a deterministic transaction processing algorithm, Gus. Gus leverages the knowledge of gas fees to reduce blocking. The main idea is that we use the gas fees to determine the ideal non-blocking start and commit order of transactions. Then, the transactions deterministically read the latest data committed before it starts. This way, we can achieve determinism with little blocking. Furthermore, we propose two mechanisms to reduce transaction aborts. The reordering mechanism deterministically adjusts the commit order of the conflicted transactions to avoid aborts. The speculative mechanism optimistically assumes that the write set for a re-executed transaction remains unchanged from its latest execution to reduce abort. Evaluation results show that Gus outperforms state-of-the-art deterministic transaction processing algorithms by up to 5x.

Keywords: Transaction processing · Deterministic · Gas · Blockchain.

1 Introduction

Blockchain is a decentralized transaction processing system. Each node potentially becomes a proposer. A proposer packages the received transactions into a block and broadcasts them to other nodes (attestors). Each node will execute transactions within the block and apply them to its ledger independently since the peer nodes are untrusted. To ensure the consistency of the distributed ledgers, all nodes must process transactions deterministically.

Conventional blockchain processes transactions with a single thread, naturally achieving determinism. To fully utilize the power of multi-core processors, previous researches adopt and optimize the concurrency control algorithms from

deterministic databases. We divide the algorithms into two classes. One is single-batch algorithms [3, 6], which read the latest data but validate and commit transactions in a pre-determined order (TID). Thus, they suffer from much blocking due to sequential validation and commit. The other is multi-batch algorithms [9, 13], where the transactions read the snapshot of the previous batch and are validated and committed concurrently. These algorithms suffer from many aborts since the data read is too old. Besides, none of the existing algorithms make use of an intrinsic property of the blockchain transactions - gas fee.

Gas fee is designed to motivate servers to participate in decentralized networks and protect them from attacks. The gas fee covers the energy consumption of transaction execution and is set when the transaction is initialized. Users can employ the gas evaluation methods to establish the gas fees. The latest dynamic method [8] provides gas estimates with an accuracy of less than 5% error. Ideally, the gas fee is proportional to the transaction execution time.

In this paper, we propose a novel deterministic transaction processing algorithm, Gus. Gus adopts the single-batch scheme and uses knowledge of gas fees to reduce blocking. The main idea is that we assign a start and commit times to the transaction before it is executed. The start time is determined by the gas fee of previously executed transactions. Commit time is determined based on the start time and the gas fee of the transaction to be executed. In the execution, the transaction deterministically reads the data committed before its start time and commits the data written with its commit time. The transaction will be aborted if visible data is committed between its start and commit time. The aborted transaction will be assigned a new start and commit time for re-execution. Fig. 1 (c) shows an ideal example where the gas fee is proportional to the execution time. The commit order of transactions is determined by commit times allocated through gas fees. T3 and T7 are aborted since T1 commits visible data between their start and commit time.

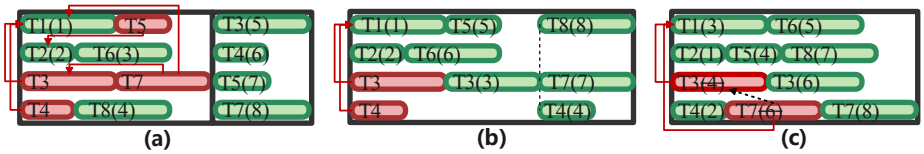


Fig. 1: **(a)**: multi-batch algorithms [9, 13]; **(b)**: single-batch algorithms [3, 6]; **(c)**: our gas-based single-batch algorithm. ⬭: transaction execution and the validation decision (commit/abort), and we mark the TID and the commit order; ⟵: conflict.

Gus can reduce blocking. This is because transactions are committed in a non-blocking order by their estimated execution times (gas fees). For example, we estimate that T4 ends execution earlier than T1 (with less gas fee), so T4 commits before T1 rather than validating and committing after T1 commits. In

this way, Gus has less blocking than single-batch algorithms and fewer aborts than multi-batch algorithms. However, Gus can not eliminate blocking since gas fees may be inaccurate. For example, if T1 has a shorter execution time but a higher gas fee than T4, T1 must wait for T4 to commit before validating.

Furthermore, we propose reordering and speculative mechanisms to reduce aborts. The reordering mechanism reduces aborts by changing the commit order of the conflicted transactions. For example, in Fig. 1 (c), T3 can avoid the conflict when it is committed before T1. The speculative mechanism provides an option of utilizing the multi-batch scheme, where aborted transactions are moved to the next batch for re-execution. It optimistically assumes that the write set of an aborted transaction is unchanged in the next execution. Thus in the next batch, transactions can avoid conflicts with the re-executed transactions by their previous write set. For example, in Fig. 1 (c), if the speculative mechanism is enabled, T3 and T7 will be moved to the next batch for re-execution. T7 can wait for T3 to commit before reading to avoid aborts, as T3's write set is known before re-execution.

Our evaluation on two popular benchmarks shows that Gus outperforms state-of-the-art deterministic transaction processing algorithms by up to 4.7x. The reordering and speculative mechanisms can achieve 3.3x and 2.6x improvements.

2 Background and Relative Works

Transaction processing in blockchain. Shi et al. [11] divided blockchain concurrent transaction processing models into two main classes. The first is the attestor-attestor concurrency model. The proposer utilizes conventional concurrency control algorithms to execute transactions and generate transaction dependency graphs. Afterward, the proposer broadcasts both the transactions and the graph to the attestors. The attestors concurrently execute transactions in a deterministic manner based on the graph. This model requires synchronization between the proposer and the attestor and suffers from time-consuming dependency analysis. The second is the proposer-attestor concurrency model. The proposer determines the order of transactions, and then the proposer and attestors deterministically execute the transactions based on the pre-determined order. Gus and the state-of-the-art approaches [13, 6] belong to the second model.

Deterministic database. Our work ensures that the transaction processing results are deterministic, so we can borrow the idea from the deterministic databases. The early deterministic databases [2, 12, 4, 5] rely on the prior knowledge of read-write sets, which allow transactions to execute in a predetermined order without aborts. However, it is impractical to obtain read-write sets in the blockchain since blockchain transactions are Turing complete. Aria [9] and DOCC [3] break this limitation.

Aria [9] adopts a multi-batch transaction processing algorithm. As shown in Fig. 1 (a), transactions in a batch are executed concurrently by reading the

snapshot of the previous batch. Transactions that violate serializability will be moved to the next batch for re-execution. A recent work, DVC [13], is a variant of Aria. It utilizes the read-write set obtained in the execution to distribute the aborted transactions to a conflict-less batch to reduce aborts.

DOCC [3] is a single-batch transaction processing algorithm. As shown in Fig. 1 (b), transactions are executed concurrently by reading the latest committed data while validated and committed serially in TID order. The transaction will be aborted and re-executed if the data read is overwritten before validation. Block-STM [6] is a multi-version single-batch algorithm, in which the versions written by the aborted transactions are kept in the version chain. In this way, a transaction can wait for its prior known dependent versions to commit before reading to avoid abort.

3 Gas-based Deterministic Concurrency Model

The proposer assigns **TID**s to transactions in their received order. Once the total gas fees of the transactions reach a threshold (100 K units of gas by default), the transactions will be packaged into a block and sent to the attestors. We adopt the inter-node concurrency model, so the proposer and the attestors process a block of transactions asynchronously in a deterministic manner.

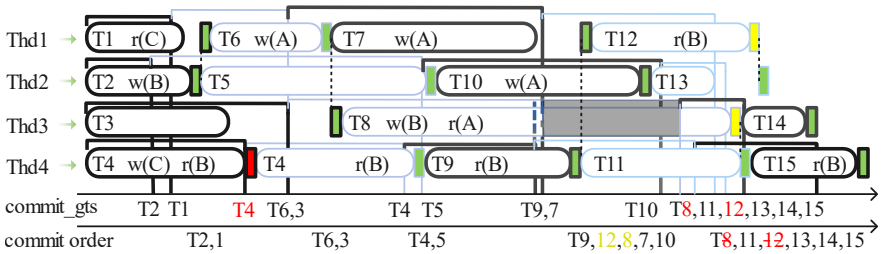


Fig. 2: An example of Gus. ⌐¬: the left boundary is $start\_gts$, the right boundary is $commit\_gts$, and the width is gas_fee; ⬭: transaction execution; ▮: transaction validation and commit/commit after reordering/abort; **r/w(X)**: read/write the record X.

Gus deterministically processes a block of transactions in a single batch. Each worker thread maintains a **gas timestamp** (gts). gts is a tuple variable. The structure and the necessary operations are as follows.

gts: $\{1 : uint\ (\sum gas\_fee),\ 2 : uint\ (thd\_id)\}$
$a\ (gts) + gas\_fee\ (uint)$: **return** $gts\{a.1 + gas\_fee, a.2\}$
$a\ (gts) < b\ (gts)$: **if** $a.1 = b.1$ **then return** $a.2 < b.2$ **else return** $a.1 < b.1$
$a\ (gts) = b\ (gts)$: **return** $a.1 = b.1$ && $a.2 = b.2$

The running logic of the worker thread is shown in Algorithm 1. The worker thread assigns *start_gts* and *commit_gts* to the transaction to be executed and updates the *gts* of the thread accordingly, as shown in lines 13 to 16. During execution, a transaction deterministically reads the latest data committed before its *start_gts*. For example, in Fig. 2, T8 reads the latest data committed by T1-T3 and T6.

After a transaction is executed, the worker thread can validate the transaction only when its *gts* is minimum. Otherwise, the worker thread will be blocked. For example, in Fig. 2, Thd1 completes the execution of T1, but it needs to wait until Thd2 commits T2 before validating T1. This is because T2 has a longer execution time but has a lower gas fee than T1. There are no identical *gts* between worker threads. This is because the *gts* is determined by the gas fees and a unique worker thread ID. For example, the combined gas fees of T1 and T6 equal the gas fee of T3. However, T6 is validated before T3 since the worker thread executing T6 has a smaller *thd_id* than the worker thread executing T3. This ensures that ① the transactions are validated in a deterministic order.

If the transaction passes the validation (which will be detailed in Sec. 4 and Sec. 5), the data written will be committed with the *commit_gts*. Then, the worker thread will pop the transaction from the global transaction queue for execution. If the transaction violates the serializability of the *commit_gts* order, e.g., T4 conflicts with T2, it will be aborted and re-executed. ② The validation result is deterministic since the transaction reads the deterministic data in execution. By ① and ②, the transactions are committed/aborted deterministically in the *commit_gts* order.

Algorithm 1: Worker Thread Runing Model

```
1  Function (thd *Thread) Run():
2      thd.wait()
3      for txn ← the unassigned transaction with the smallest TID do
4          thd.update_gts(txn)
5          thd.execute(txn) // read the data committed before start_gts.
6          thd.wait() // wait until the thd.gts is minimum.
7          if thd.validate(txn) then
8              thd.commit(txn) // validate and commit with commit_gts.
9          else
10             thd.abort(txn)
11             goto line4
12     thd.Exit() // thd.gts is set to +∞.
13 Function (thd *Thread) update_gts(txn *Txn):
14     txn.started_gts ← thd.gts
15     txn.committed_gts ← thd.gts + txn.gas_fee
16     thd.gts ← txn.committed_gts
```

4 Multi-version Scheme

We use a multi-version scheme to manage transactional read and write. A version has two variables *commit_gts* and *order_gts*. Each record maintains a singly linked list to store the committed versions in the order of *order_gts* which is equal to *commit_gts* if there is no reordering. We define dependencies between transaction and version as follows.

Definition 1. *There is a version V in record R, and a transaction T reads R. If V.commit_gts* $\leq$ *T.start_gts, then V is* ***readable*** *for T.*

Definition 2. *There is a version V in record R, and a transaction T reads R. If V.order_gts (= V.commit_gts if no reordering)* $<$ *T.commit_gts, then V is* ***visible*** *for T.*

In the execution, a transaction will read the *latest readable versions*, and the data written is stored in the local cache. In the validation, the transaction gets the next versions of the versions read in the version chain. If the next version is *visible* to the transaction (a.k.a., conflicted version), the transaction suffers a conflict. If the validation passes, the transaction inserts the new versions written into the version chain with *commit_gts*. Only the latest version in the version chain is available for transactions in the next block. Thus, we will reclaim versions other than the latest version (i.e., garbage collection) after the block processing is completed.

5 Reordering Mechanism

In the validation, we can use the reordering mechanism to reduce transaction aborts. The ① principle of the reordering mechanism is to commit the conflicted transaction with an earlier order to make the conflicted versions invisible. To ensure ② serializability, we must ensure that the transactions that have been committed do not depend on the reordered transaction, and the versions read are also visible after reordering. Besides, we must ensure that ③ the versions are organized in the order of *order_gts* in the version chain to ensure the correctness of the validation.

We use Fig. 2 as an example. T8 has read-write conflicts with T7 and T10. To satisfy ①, T8 can be committed earlier than T7, making the versions written by T7 and T10 invisible. To satisfy ②, the commit order of T8 must be later than T4 and T9 that have been committed. Otherwise, T4 and T9 will have a read-write dependency on T8 and break serializability. In addition, T8 must be committed after T6 to keep the versions read visible. Therefore, to avoid conflicts, T8 can be committed with the *order_gts* equal to T7's *commit_gts*. To satisfy ③, we do not allow version insertion between the *order_gts* and *commit_gts* of T8 in T8's write set, i.e., the shadow range of T8 in Fig. 2. Note that the versions written by T8 are not readable for T12, even if T8 is committed before (*order_gts* is less than) the *start_gts* of T12. Thus, T12 will conflict with T8. However, T12 can still be committed after T9 and before T8 by reordering.

6 Speculative Mechanism

The speculative mechanism is optional, which optimistically assumes that the write set of a re-executed transaction remains unchanged from its latest execution. If the known write set is visible but unreadable, the transaction can be executed later than the commit of the known write set. Then, we can update the *start_gts* of the transaction to the *commit_gts* of the known write set to make the unreadable version readable, avoiding conflicts. Besides, with the increase of the *start_gts*, the newer visible versions that are out of the known write set are readable.

The speculation mechanism uses a mult-batch scheme to execute a block of transactions where the aborted transaction will be re-executed in the next batch. Each batch has two phases. The first phase installs the version placeholders of the previous write set into the version chain, and the second phase executes the transaction based on the known version written. The multi-batch scheme is designed to address the issue of high data skew workload. In this workload, re-executed transactions may face further aborts. Multi-batching can help prevent these aborts by assigning transactions to a batch with known write sets.

However, if the write set is changed, the reader may be unaware of the new visible version causing a conflict. Additionally, the aborted pre-installed versions can cause transactions to be blocked in vain.

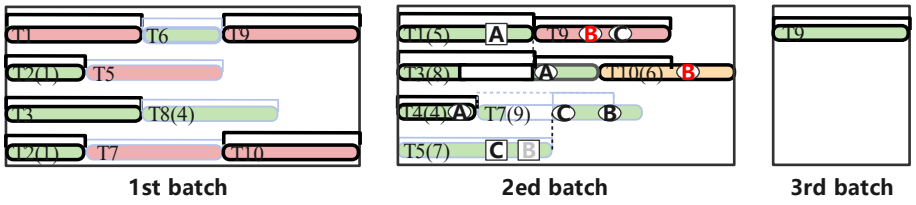


Fig. 3: An example of speculative mechanism. Ⓧ: reading X (red indicates a conflict); X: writing X (**black** indicates the writing record within the previous write set, and **gray** indicates the writing record outside of the write set).

Fig. 3 shows an example of the speculative mechanism. The first batch is run as original since there are no previous write sets, and the aborted transactions are moved to the next batch. In the second batch, because the write set of the aborted transaction is installed, T3 waits for T1 to commit version A before reading, avoiding the abort. T5 writes record B that is outside its previous write set. The new version in record B is unreadable for T9, so T9 is aborted. However, T7 sets *start_gts* to T5's *commit_gts* before reading the version in record C. Thus, the new version in record B is readable for T7, and T7 is committed. T10 conflicts with the new version in record B written by T5, but it can be committed before T5 and later than T1 through reordering.

7 Experimental Evaluation

7.1 Experimental Setup

Workloads. We generate workloads using two popular benchmarks, TPC-C and YCSB. TPC-C [7] simulates an e-commerce order processing application. We configure the proportion of NewOrder and Payment transactions to be 50%, respectively, and control the contention of the workload by adjusting the *Warehouses* (similar to Partitions). YCSB [1] is a key-value operation generator. The keys follow a Zipfian distribution. We adjust the key skew through the parameter *theta* in Zipfian. By default, a transaction comprises ten read/write operations, and the read-write ratio of the transactions is adjustable.

Gas fees. We pre-execute a transaction based on the latest snapshot before packaging it into a block, and the read-write operation count during the execution is used as the gas fee of the transaction. The similar methods are also employed by Web3 [3], V-Gas [10], and Li et al [8] for gas evaluation, which is proven to be more accurate than the static methods such as Solc [4].

Algorithms. We implement the following algorithms in our framework with Golang 1.18. ① **Gus**: the algorithm we proposed with the reordering mechanism enabled. ② **Gus_w/o_R.O.**: Gus with the reordering mechanism disabled. ③ **Gus_S.P.**: Gus with the speculative mechanism enabled. ④ **Gus_F.B.**: Gus enables a fallback strategy for the case that the gas fees are unavailable, in which all gas fees are set to the same value (e.g. 1 Wei). ⑤ **Aria** [9]:a multi-batch deterministic algorithm. ⑥ **DVC** [13]: a variant of Aria leverages the previous read-write set of the re-executing transaction to avoid abort. ⑦ **Block-STM** [6]: a single-batch multi-version deterministic algorithm that forces transactions to be committed in the pre-determined order (TID) to achieve determinism.

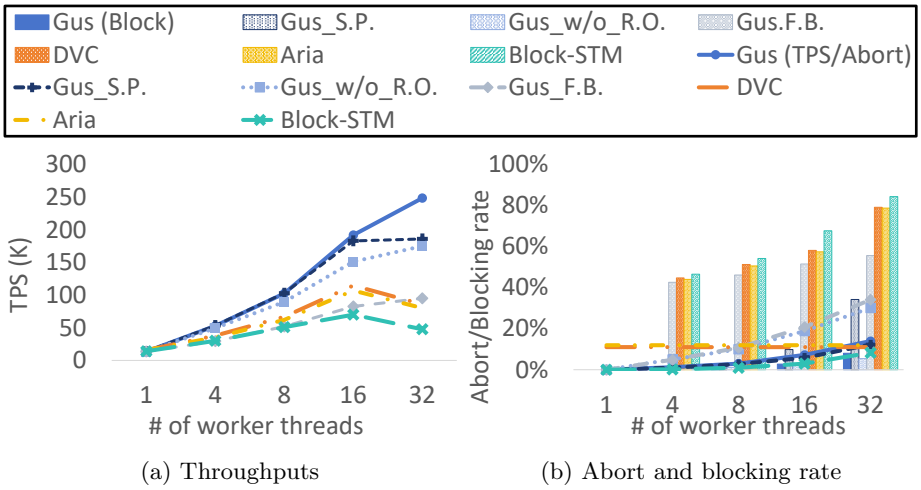
Performance metrics. For each test, we report the throughput, blocking rate ($\sum_{i=1}^{\infty} \frac{WorkerThread\_i.BlockPeriod}{WorkerThread\_i.RunningPeriod}$), and abort rate ($\frac{AbortTxnCnt}{ExecTxnCnt}$). All data presented are averages of 10 runs.

Hardware and System Software. The experiment servers are equipped with 32 cores 3.4 GHz processor and 128 GB DDR4 memory. The operating system is CentOS 6.5.

7.2 TPC-C Result

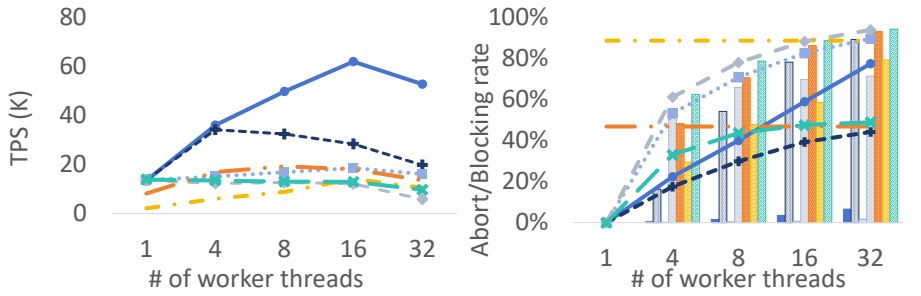
First, we measure the performance of the algorithms under TPC-C. We tuned Warehouse to 1 and 32 to generate high- and low-contention workloads. We vary the number of worker threads from 1 to 32 to study the scalability of the algorithms.

[3] https://web3js.readthedocs.io/en/v1.2.0/web3-eth.html#estimategas.
[4] https://github.com/ethereum/solidity.



(a) Throughputs (b) Abort and blocking rate

Fig. 4: Low-contention TPC-C workloads.



(a) Throughputs (b) Abort and blocking rate

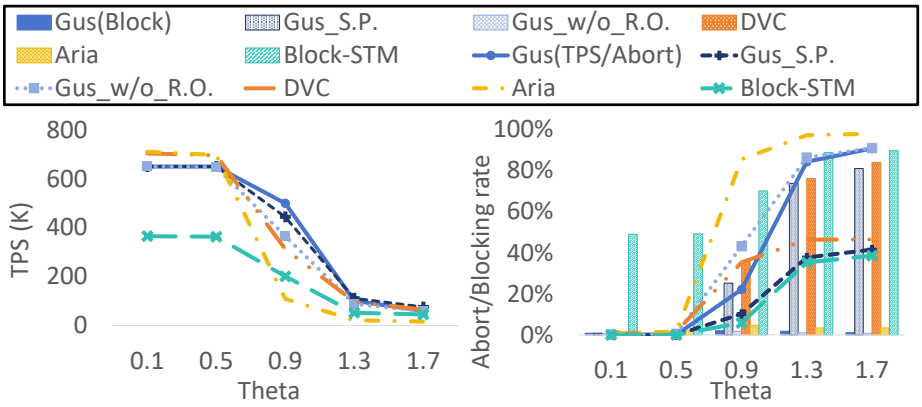
Fig. 5: High-contention TPC-C workloads.

Low-contention. Fig. 4 (a) shows that Gus's throughput is 1.7x, 1.8x and 2.7x that of DVC, Aria, and Block-STM when 16 worker threads are available. The reason can be found in Fig. 4 (b). By leveraging the knowledge of gas fees, Gus' blocking rate is only 0.2% to 8%, which is far lower than that of competitors. The reordering mechanism brings a 1.4x throughput improvement as it reduces the abort rate by 61%. Gus_S.P. is slightly worse than Gus. This is because Gus_S.P. executes in a blocking manner, limiting the scalability, and the blocking will also result in an error between transaction execution time and gas fee. Gus_F.B. is slightly better than Block-STM because it utilises the reordering mechanism, while Block-STM enforces that the commit order is consistent with the TID order.

High-contention. We report the results in Fig. 5. Gus outperforms the competitors by a large margin, achieving 3.4x, 4.4x and 4.7x higher throughput than DVC, Aria, and Block-STM at 16 worker threads. This is because the blocking rates of DVC, Aria, and Block-STM are up to 86%, 58%, and 88% while Gus is only 4%. Gus achieves 3.3x higher throughput than Gus_w/o_R.O. as it reduces the abort rate by 29%. Gus_S.P.'s abort rate is 33% lower than Gus's, but its performance is worse, as its blocking rate is up to 78% at 16 worker threads. Gus_F.B. has a similar throughput to Aria and Block-STM.

7.3 YCSB Result

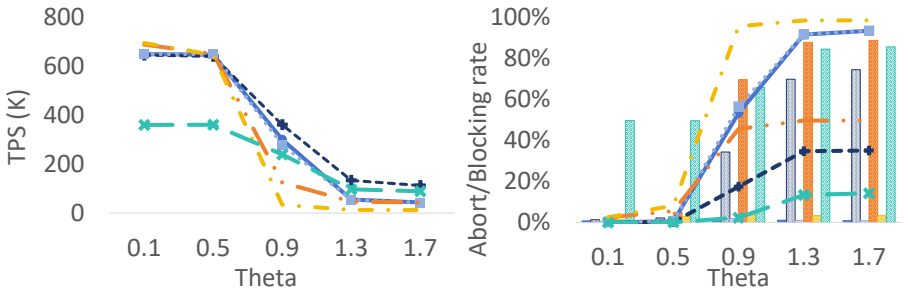
Then, we measure the performance under YCSB. We adjust the read-write ratio to 8:2 and 2:8 to generate read- and write-intensive workloads. We vary the theta in Zipfian from 0.1 to 1.7 to study the impact of data skew on performances. We omit the performance comparison of Gus_F.B. since it exhibits similar performance to Gus when the gas fees of the transactions generated by YCSB are the same.



(a) Throughputs (b) Abort and blocking rate

Fig. 6: Read-intensive YCSB workloads.

Read-intensive. We show the results in Fig. 6. At low-skew scenarios (theta between 0.1 and 0.5), Aria and DVC are the best. This is because the read-write overhead is low in the single-version scheme. Block-STM performs the worst since serial validation and committing become its bottleneck. At mid- and high-skew scenarios (theta between 0.9 and 1.7), Gus achieves 1.5x, 2.5x, and 4.6x higher throughput than DVC, Block-STM, and Aria. Reordering works best when theta is 0.9, reducing the abort rate by 49% and delivering a 1.4x performance improvement. When theta is 1.7, the throughput of Gus_S.P. is 1.3x that of Gus,



(a) Throughputs (b) Abort and blocking rate

Fig. 7: Write-intensive YCSB workloads.

in which Gus_S.P. has a 54% lower abort rate than Gus.

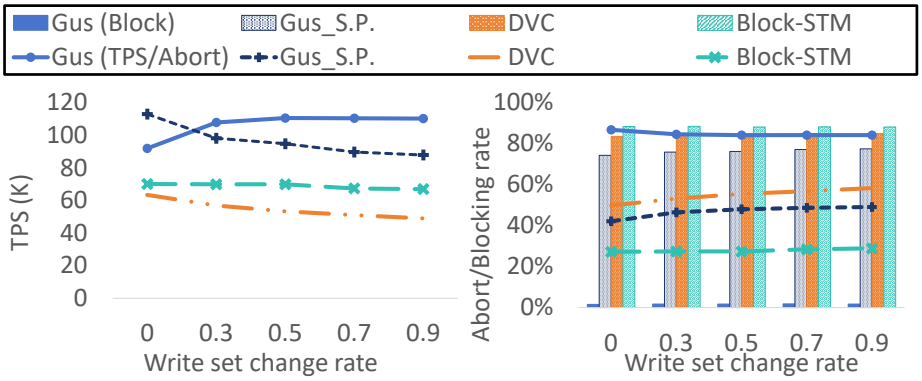
Write-intensive. We report the results in Fig. 7. At low-skew scenarios (theta between 0.1 and 0.5), the throughput rankings are similar to those observed in read-intensive workloads. At the mid-skew (theta is 0.9) scenario, Block-STM is better than DVC and Aria. This is because Block-STM uses a multi-version scheme avoiding write-write conflicts. Thus Block-STM has 73% and 86% lower abort rates than DVC and Aria. At high-skew scenarios (theta between 1.3 and 1.7), Block-STM outperforms Gus since the abort rate becomes Gus's bottleneck. The effect of reordering is weak and reduces the abort rate by only 5%. This is because reordering enforces that no version is committed between the *order_gts* and *commit_gts* in the write set. Gus_S.P. performs the best in mid- and high-skew scenarios, achieving 1.4x and 2.4x higher throughput than Block-STM and Gus, since it reduces the abort rate by 67% with speculative mechanism.

7.4 Factor Analysis

Next, we analyse three factors that affect the performance of Gus and Gus_S.P.

Change in the write set. We use YCSB to generate the write set changing workloads. We introduce a parameter -*change rate*, which indicates the percentage probability of the write set changing after a transaction is aborted. The changed write set and the previous write set obey the same distribution. We configured theta to 1.2 and the number of worker threads to 16.

The results are shown in Fig. 8. As the change rate goes from 0 to 0.9, the throughput of Gus_S.P. decreases by 22%. This is because newly inserted versions are not noticed by readers and cause conflicts. Similarly, DVC's throughput is reduced by 23% since the changes in the write set will cause the aborted transactions to be assigned to the conflicting batches. In contrast, write set changes have little impact on Block-STM since the abort rate is not the bottleneck. Changes in the write set actually make a performance improvement to Gus. This is be-

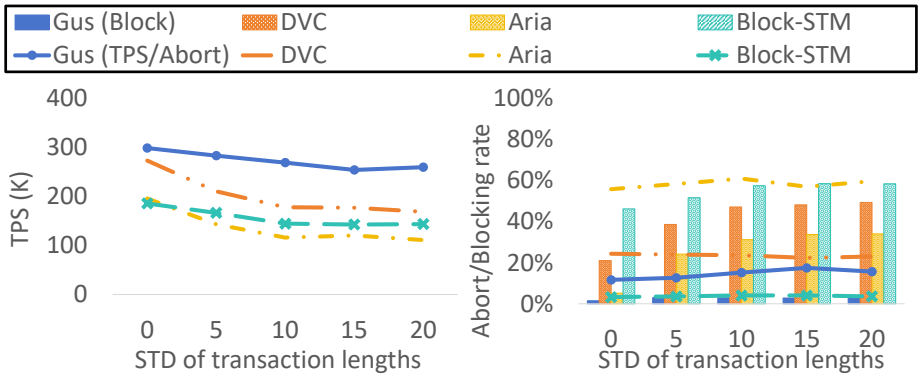


(a) Throughputs (b) Abort and blocking rate

Fig. 8: Write set changing workloads.

cause some transactions cannot be reordered due to writing the contention keys (hotkeys). However, the change in the write set may swap out the contention keys and reduce aborts.

Transaction length skew. We use YCSB to generate the workloads. We vary



(a) Throughputs (b) Abort and blocking rate

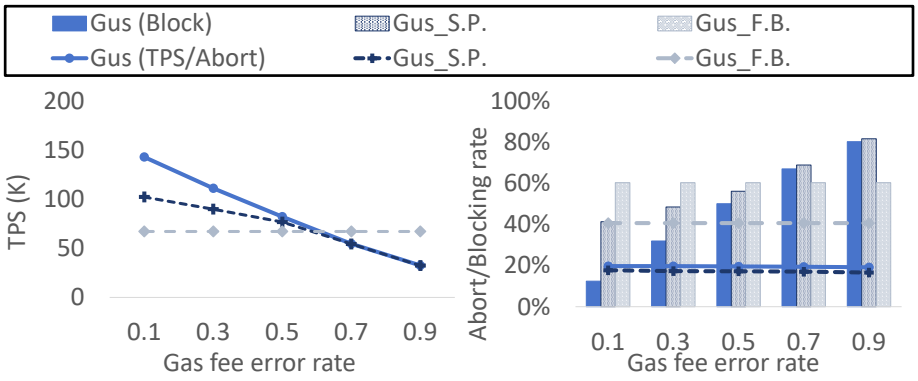
Fig. 9: Transaction length skewed workloads.

the standard deviation of the transaction length from 0 to 20. The average length of transactions is 20. Theta is 0.8. The number of worker threads is 16.

We report the results in Fig. 9. The increase in transaction length skew has the greatest negative impact on the multi-batch algorithms, Aria and DVC, with the blocking rate increased by 6.6x and 2.3x and the throughput decreased by

43% and 38%. This is because the worker threads will suffer from the blocking in the batch barrier caused by their unbalanced runtimes. The throughput of Block-STM is reduced by 23% since it commits transactions in the pre-determined order, causing short transactions to wait for the previous long transactions to commit. In contrast, transaction skew has less impact on Gus. This is because we determine the commit order based on the gas fee of the transactions. The committing of the long transactions will be delayed without blocking short transactions.

Gas fee error. We use a mid-contention TPC-C workload (8 warehouses) for



(a) Throughputs (b) Abort and blocking rate

Fig. 10: Gas fee error workloads.

evaluation. We introduce a parameter - *error rate*. If the estimated gas is e, then the gas fee of the transaction is $e \times (1 \pm error\_rate)$. The number of worker threads is set to 16.

The results are shown in Fig. 10. As the error rate increases from 0.1 to 0.9, the blocking rate of Gus and Gus_S.P. increases by 6.4x and 2.0x, and the throughput decreases by 77% and 68%. When the error rate is greater than 0.7, the performance of Gus and Gus_S.P. is lower than Gus_F.B. This is because the gas fee error can result in transactions that are completed execution earlier to be committed later.

8 Conclusion

In this paper, we introduce Gus, a deterministic transaction processing algorithm for blockchain. Gus leverages the knowledge of the gas fees in blockchain transactions to reduce blocking in concurrent execution. We also propose reordering and speculative mechanisms to reduce transaction aborts. The results show that Gus

outperforms the state-of-the-art deterministic transaction processing algorithms by a large margin. Reordering and speculative mechanisms are most effective under high-contention TPC-C workload and high-skew YCSB workload, respectively. Besides, we find that write set change and transaction length skew have less negative impacts on Gus than the competitors. Gus's performance is related to the accuracy of gas fees, and we can use the fallback strategy to prevent undesirable performances when the gas fees are unavailable.

References

1. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154 (2010)
2. Cowling, J., Liskov, B.: Granola:low-overhead distributed transaction coordination. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12). pp. 223–235 (2012)
3. Dong, Z.Y., Tang, C.Z., Wang, J.C., Wang, Z.G., Chen, H.B., Zang, B.Y.: Optimistic transaction processing in deterministic database. Journal of Computer Science and Technology **35**(2), 382–394 (2020)
4. Faleiro, J.M., Abadi, D.J.: Rethinking serializable multiversion concurrency control. Proc. VLDB Endow. **8**(11), 1190–1201 (2015)
5. Faleiro, J.M., Abadi, D.J., Hellerstein, J.M.: High performance transactions via early write visibility. Proceedings of the VLDB Endowment **10**(5) (2017)
6. Gelashvili, R., Spiegelman, A., Xiang, Z., Danezis, G., Li, Z., Malkhi, D., Xia, Y., Zhou, R.: Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In: Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. pp. 232–244 (2023)
7. Leutenegger, S.T., Dias, D.: A modeling study of the tpc-c benchmark. ACM Sigmod Record **22**(2), 22–31 (1993)
8. Li, C., Nie, S., Cao, Y., Yu, Y., Hu, Z.: Dynamic gas estimation of loops using machine learning. In: Blockchain and Trustworthy Systems: Second International Conference, BlockSys 2020, Dali, China, August 6–7, 2020, Revised Selected Papers 2. pp. 428–441. Springer (2020)
9. Lu, Y., Yu, X., Cao, L., Madden, S.: Aria: a fast and practical deterministic oltp database. Proceedings of the VLDB Endowment **13**(12), 2047–2060 (2020)
10. Ma, F., Fu, Y., Ren, M., Sun, W., Liu, Z., Jiang, Y., Sun, J., Sun, J.: Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability. arXiv preprint arXiv:1910.02945 (2019)
11. Shi, J., Wu, H., Gao, H., Zhang, W.: Overview on parallel execution models of smart contract transactions in blockchains. Journal of Software **33**(11), 4084–4106 (2021)
12. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: fast distributed transactions for partitioned database systems. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. pp. 1–12 (2012)
13. Xia, H., Chen, J., Ma, N., Huang, J., Du, X.: Efficient execution of blockchain transactions through deterministic concurrency control. In: International Conference on Database Systems for Advanced Applications. pp. 509–518. Springer (2023)