Anthemius: Efficient & Modular Block Assembly for Concurrent Execution

Ray Neiheiser[1] and Eleftherios Kokoris-Kogias[2]

[1] ISTA, Klosterneuburg, Austria
[2] Mysten Labs, Athens, Greece

Abstract. Many blockchains such as Ethereum execute all incoming transactions sequentially significantly limiting the potential throughput. A common approach to scale execution is parallel execution engines that fully utilize modern multi-core architectures. Parallel execution is then either done optimistically, by executing transactions in parallel and detecting conflicts on the fly, or guided, by requiring exhaustive client transaction hints and scheduling transactions accordingly.
However, recent studies have shown that the performance of parallel execution engines depends on the nature of the underlying workload. In fact, in some cases, only a 60% speed-up compared to sequential execution could be obtained. This is the case, as transactions that access the same resources must be executed sequentially. For example, if 10% of the transactions in a block access the same resource, the execution cannot meaningfully scale beyond 10 cores. Therefore, a single popular application can bottleneck the execution and limit the potential throughput.
In this paper, we introduce Anthemius, a block construction algorithm that optimizes parallel transaction execution throughput. We evaluate Anthemius exhaustively under a range of workloads, and show that Anthemius enables the underlying parallel execution engine to process over twice as many transactions.

Keywords: Blockchain · Parallel Execution · Smart Contracts · Distributed Ledger Technology.

1 Introduction

The growing interest in blockchain and distributed ledger technology has resulted in many research advances in the field, ranging from improvements on the consensus layer [15, 4] to sharding [11] and parallel transaction execution [8, 14]. As most blockchains still execute transactions sequentially, parallel smart contract execution engines that take advantage of modern multi-core architectures are considered a crucial building block to scale blockchain transaction throughput [8].

Existing approaches to parallel execution can be roughly divided into two categories: optimistic and guided. Optimistic approaches, such as Block-STM [8], are designed to execute transactions in parallel, detect conflicts as they arise, and

re-execute affected transactions. However, in blockchain environments characterized by highly contended workloads [16, 14], conflicts arise more often, requiring more frequent re-executions of transactions.

In contrast to optimistic approaches, guided approaches strictly limit read/write access by requiring transactions to pre-declare an exhaustive list of resources (i.e., addresses) that will be accessed during execution. This allows the scheduler to identify independent transactions and execute them concurrently. Examples of this approach include FuelVM, Solana, or Sui [6, 21, 19]. While this avoids the re-execution overhead in settings with high contention, it puts additional load on the application developers. Furthermore, in some cases, it may not be possible to precisely predict at transaction creation time which resources will be accessed during execution, as the application state might change in the meantime. Then, an overly pessimistic approach is required, locking a wider range of resources, and potentially resulting in the sequential execution of transactions that otherwise could have been executed concurrently.

Combining both approaches, Polygon recently introduced an update [18] that extracts transaction dependencies during block creation and includes this dependency tree as metadata in the block. This approach allows to optimize scheduling during the execution phase, avoiding unnecessary re-executions and pessimistic locking [18]. Nonetheless, this approach requires executing transactions on the critical path of consensus during block creation, crippling the potential throughput. A similar approach is Chiron [14] which leverages execution hints to speed up execution on struggling validators and full nodes. Chiron guarantees safety in the presence of invalid hints by utilizing the validation step of Block-STM, which identifies conflicting resource accesses and reschedules transactions that potentially accessed shared resources in parallel for re-execution [8].

However, as outlined in [14], due to the characteristics of blockchain workloads, transaction execution remains a significant bottleneck. This is the case, as transactions that access the same resources must be executed sequentially and, as several recent studies have shown, in practice a significant portion of the transactions access the same resources, resulting in a long sequential path of transactions slowing down the system [7, 14]. As such, the performance is currently limited by the workload.

Due to the nature of the problem, a single popular application can bottleneck the execution engine and cripple the throughput of the system [14]. This could be a newly launched NFT, a popularly traded token, or even the on/off-boarding of a popular layer-2 smart contract. This is further aggravated by the fact that most existing blockchains that support parallel execution currently have no pricing mechanisms to charge clients for accessing popular resources causing system bottlenecks.

Most blockchains such as Ethereum [3] prevent extensive execution times by limiting the combined execution complexity in gas of each given block. However, a single parameter is insufficient in the context of parallel execution, as it does not take transaction dependencies and potential parallelization into account. Therefore, a novel approach is necessary to make block assembly sensitive to

transaction dependencies and execution complexity, charging clients for accessing popular resources and delaying transactions that would otherwise bottleneck the execution.

In this paper, we propose ANTHEMIUS, a novel approach to construct blocks that takes both the execution complexity in gas and the distribution of resource accesses into account to construct "Good Blocks" that can be executed efficiently in parallel. We evaluate ANTHEMIUS extensively under a series of realistic workloads, showing a consistent speed-up up to 240% compared to native parallel execution. ANTHEMIUS not only vastly improves the execution performance but also prevents popular or malicious applications from bottlenecking the system, eliminating a performance attack scenario. ANTHEMIUS provides different latency paths between transactions accessing congested and not congested resources. Transactions can still be fast-tracked by paying higher transaction fees, resulting in a price that more closely reflects its resource consumption. We discuss this further in Section 6.

Moreover, thanks to its modular design, ANTHEMIUS can be integrated into any state-of-the-art blockchain seamlessly, without the need for a hard fork or modifications to the execution engine or consensus mechanism. ANTHEMIUS operates stateless and only requires execution hints such as the resources that will be accessed during execution. In blockchains such as Sui and Solana [19, 21] these hints are already present during block construction, while in blockchains such as Aptos or Ethereum [5, 3] these hints could either be simulated in a pre-execution step or generated at the full nodes.

In summary, we provide the following contributions:

- We propose ANTHEMIUS, a novel and modular block construction algorithm and approach to speed up parallel execution without security tradeoffs.
- We evaluate ANTHEMIUS integrated with both an optimistic and a guided execution engine under the Chiron benchmarks resulting in a significant speed-up in almost all settings.

In Section 2, we present the System Model of ANTHEMIUS, followed by a detailed overview of ANTHEMIUS in Section 3. Next, we describe the implementation and evaluation in Section 4. Related work is reviewed in Section 5, and potential drawbacks, along with their solutions, are discussed in Section 6. Finally, we conclude the paper in Section 7.

2 System Model

We assume a blockchain environment consisting of N server processes $p_1, p_2, .., p_N$ and I client processes $c_1, c_2, .., c_I$. Clients send signed transactions to the server processes to be included in a future block. The blockchain functions as the Public Key Infrastructure where the identifier of a client is its public key, and clients use their private keys to sign their transactions.

We assume a consensus abstraction as a blackbox, where one or more processes construct blocks of transactions and propose them to the consensus mechanism. As a result, the consensus abstraction outputs an ordered sequence of blocks $b_1, b_2, \ldots, b_n$, which is then processed by the execution engine. Additionally, we assume an execution engine abstraction as a blackbox that receives this ordered sequence of blocks from the consensus abstraction and executes them deterministically.

In the context of this work, we make no assumptions regarding the coupling between the consensus and execution layers. The interaction between consensus and execution may either follow a modular, decoupled approach, as in Sui and Aptos [19, 5], or operate in a tightly coupled, sequential manner, as in Ethereum [3].

Client transactions might range from simple peer-to-peer transactions to complex application logic with the help of smart contracts. As applications might access arbitrary resources (i.e., addresses) that can not easily be deduced, we assume the existence of a system that provides hints about the resources a transaction will access during execution to the block producer. This can either be in the form of client hints as in Solana or Sui [21, 19], or in the form of an optimistic pre-execution step that determines these hints locally as in Polygon [18]. However, we do not assume the list of hints to be exhaustive or correct. Transactions with incomplete or incorrect hints might trigger re-executions if the execution engine is Block-STM or a derivative [8, 14], or aborted in Solana or Sui [21, 19].

3 Anthemius

The primary objective of Anthemius is to redesign the block-assembly approach in blockchains that offer parallel execution to improve the overall system throughput and prevent popular applications from creating bottlenecks by factoring in transaction dependencies and execution time.

At the time of writing, most blockchains that support parallel transaction execution use a single parameter such as the computational complexity in gas, the raw block size in bytes, or the number of transactions to limit the block size [5, 19, 21]. However, in the context of parallel transaction execution, a single parameter does not reflect the execution complexity of a block. If all transactions in the block access the same resource, the execution time is the sum of the runtime of all transactions. In contrast, if none of the transactions access conflicting resources, the runtime depends on the number of cores.

Therefore, as a first step to begin constructing "Good Blocks", we need parameters that allow us to quantify this. We deploy two parameters to address this. First a transaction complexity parameter in Gas, similar to Ethereum, and second a concurrency parameter c describing the system's ability to execute transactions in parallel (i.e. number of cores). As a result, the total maximum capacity of each block is $c * Gas$.

In the next sections, we first discuss where Anthemius fits into existing blockchain architectures. Following that, we outline the design of the block con-

struction algorithm that considers both parameters and constructs blocks sensitive to transaction dependencies and their execution time to speed up the parallel execution of the block.

3.1 Architecture

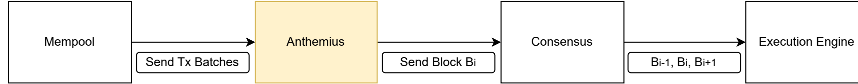


Fig. 1: ANTHEMIUS is inserted between the Mempool and Consensus

Figure 1 shows where ANTHEMIUS fits into the existing protocol stack of a blockchain. ANTHEMIUS is a modular layer that can be inserted between the consensus layer and the mempool where client transactions are stored and handled. In ANTHEMIUS, instead of fetching transactions directly from the mempool, the consensus layer fetches blocks of transactions through ANTHEMIUS. In turn, ANTHEMIUS obtains its transactions from the mempool, divides transactions into batches, and constructs the block to return to consensus. Following that, the block is proposed in consensus which outputs an ordered list of blocks to the execution engine.

ANTHEMIUS requires the read and write sets of transactions, as well as an estimation of their execution time, to assess dependencies between transactions and construct blocks that can be executed efficiently in parallel. This information is already available in blockchains such as Solana [21] and Sui [19], where transactions must declare all resource addresses they access during execution. In other blockchains, such as Ethereum [3], this information can be obtained, for example, by executing the transactions.

This design allows ANTHEMIUS to be seamlessly integrated into any existing blockchain stack with minimal architectural and system changes, and without changing the block structure. Furthermore, since ANTHEMIUS operates solely on the set of transactions, their read and write sets, and their gas footprints, it remains essentially stateless. This makes ANTHEMIUS particularly suitable for deployment in modular architectures, such as Narwhal, where only the execution layer is stateful [4].

3.2 Block Construction

An important problem that has to be tackled when constructing good blocks is the absence of information regarding the structure of the current workload. If all transactions in the mempool access the same resources, attempting to schedule them efficiently can further slow down an already bottlenecked system.

Algorithm 1 *Batch Handler*

```
1: procedure CreateGoodBlock(block, maxgas, c)
2:     seqlimit = maxgas/c                                  ▷ Limit on the sequential path
3:     resmap ← ∅                                ▷ Map to track transaction dependencies
4:     skippedclients ← ∅                  ▷ Set to track clients with skipped transactions
5:     numrelax ← 0                         ▷ Number of times inclusion rate was relaxed
6:     for all batch ∈ mempool do
7:         incrate ← SCHEDULE(block, batch, seqlimit, c, resmap, skippedclients)
8:         if incrate < TARGETINCRATE
9:             if numrelax ≥ MAXRELAXNUM ∨ (incrate = 0 ∧ batch.isfull)
10:                return
11:            seqlimit = maxgas/c * MIN(MAXRELAXRATE, incrate * TARGETINCRATE)
12:            numrelax + +
```

Similarly, if the algorithm is too strict in situations where a large percentage of transactions access the same resources, the synergetic effects of executing larger batches of transactions are lost. This is the case, as, for each block, the system has to instantiate the executor and worker threads, set up the virtual machine, extract the execution results, etc.

Therefore, as a first step, we divide Anthemius into two modular elements. First, the *batch handler*, which polls batches of transactions from the mempool and hands the batches to the *batch scheduler* in a batch-by-batch fashion. Second, the *batch scheduler*, that attempts to include a given batch into the current block and provides feedback to the *batch handler* about the success rate. Subsequently, based on the feedback, the batch handler can adjust the inclusion policy to prevent too small blocks and also avoid wasting scheduling time on difficult-to-schedule workloads.

Batch handler. The functionality of the Batch Handler is outlined in Algorithm 1. The batch handler receives a *block* to fill, the global concurrency parameter c, and the maximum gas limit. It then calculates a limit on the sequential path *seqlimit* and initiates a map to track the transaction dependencies *resmap* as well as a set of clients with skipped transactions *skippedclients*.

Next, the batch handler retrieves transaction batches from the mempool and hands them to the batch scheduler alongside the block, the limit on the gas, the number of cores, the transaction resource dependencies *resmap*, and *skippedclients* set in Line 7. The batch scheduler responds with the transaction inclusion rate *incrate*.

Depending on the workload, as mentioned, the *seqlimit* may be very strict which can result in very few transactions being included in a block. Therefore, if the inclusion rate *incrate* is smaller than some TARGETINCRATE, we relax the gas limit relative to the inclusion rate, up to some MAXRELAXRATE (Line 11).

However, if the inclusion rate was too small for several consecutive attempts (i.e. $numrelax \geq$ MAXRELAXNUM), we exit scheduling to avoid building a heavily sequential block again. Furthermore, if there was an attempt to schedule a full

batch and no transaction of this batch was successfully included in the current block ($incrate = 0$) we also stop scheduling (Line 9) as this indicates that at this point transactions are only included at a high cost to execution performance and scheduling latency. The rest of the transactions are then only included in a later block.

Batch Scheduler. Scheduling transactions with interdependencies and varying runtimes is a known NP-complete problem [2] where approximate solutions can construct near optimal schedules in polynomial time. However, polynomial runtime, particularly when executed on the critical path of consensus, may lead to a construction time that outweighs the performance gains achieved from producing "Good Blocks."

Fortunately, our first insight is that a near-optimal schedule for block construction is unnecessary. Instead, our main objective is to prevent popular resources and applications from creating a bottleneck while maximizing the parallel execution. We can achieve this by iterating over the set of resources each transaction accesses, recording the cost of the sequential path leading up to the transaction, and deciding if the transaction should be included in the current block by comparing the cost of the path with the gas per core parameter. Furthermore, we also want to delay transactions that access multiple hot resources as they make it harder to schedule subsequent transactions.

As a result, the complexity of the block construction is of $O(N * k)$ where N is the number of transactions and k is the average number of resource accesses per transaction.

Algorithm 2 shows how we achieve this. The algorithm starts with the call of the SCHEDULE method, which receives the block to include the transactions in, the batch of transactions to schedule, the maximum gas per core *seqlimit*, the concurrency parameter c, the map of resources and the skipped clients. Following that, it starts iterating over all transactions in the batch (line 2). First, to maintain the order clients specified (e.g. through sequence numbers), after a client had a transaction skipped, the client is added to the *skippedclients* set and no further transaction from this client will be included in this block.(Line 4). Following that, we iterate over all reads in the transaction read-set and attempt to calculate the read with the longest path in gas leading up to this transaction (Line 7). In parallel, we count the number of *hot* reads. A hot read is a read on a resource that is accessed significantly more often than other resources.

After this, we check whether the number of hot reads exceeds a predefined threshold, MAXHOTR. If this condition is met and the transaction is not within the first or last LIM(i.e. 10%) transactions, we skip the transaction (Line 13). We delay transactions with too many hot reads as they unify several critical paths of transactions which can severely bottleneck the execution. However, we initially allow any transactions to be included up to some threshold LIM to accumulate sufficient data to assess the complexity of reads and to guarantee that transactions that access several hot resources are eventually included. Furthermore, we also allow including transactions with multiple hot reads towards the end of

Algorithm 2 *Batch Scheduler - Called in Line 7 of Algorithm 1*

```
1: procedure SCHEDULE(block, batch, seqlimit, c, resmap, skippedclients)
2:     for all tx ∈ batch do                                        ▷ Iterate over transactions
3:         if tx.sender in skippedclients
4:             continue                                             ▷ Skip transaction inclusion
5:         chaincost ← 0                                            ▷ Longest chain length
6:         hotresources ← 0
7:         for all readres ∈ tx.readset do                          ▷ Iterate over readset
8:             if readres ∈ resmap                                  ▷ Find longest chain
9:                 if resmap[readres] > chaincost                   ▷ Find read with largest cost
10:                    chaincost ← resmap[readres]
11:                if resmap[readres] > block.gas / c               ▷ Check if read exceeds limit
12:                    hotresources + +
13:        if hotresources ≥ MAXHOTR ∧ (|block| > LIM ∨ |block| < MAXLEN − LIM)
14:            skippedclients ← skippedclients ∪ tx.sender
15:            continue                                             ▷ Skip transaction inclusion
16:        if chaincost + tx.gas > seqlimit ∨ block.gas + tx.gas > seqlimit ∗ c
17:            skippedclients ← skippedclients ∪ tx.sender
18:            continue                                             ▷ Skip transaction inclusion
19:        block ← block ∪ tx                                       ▷ Add tx to Block
20:        for all writeres ∈ tx.writeset do                        ▷ Iterate over writeset
21:            if writeres ∉ resmap ∨ resmap[writeres] < chaincost
22:                resmap[writeres] ← chaincost                     ▷ Note new chain length
23:    return(numscheduled / |batch|)
```

the block as the block is almost full already and they are less likely to cause scheduling problems at this point.

Following that, we check if the transaction cost itself is larger than the max gas per core *seqlimit* or if the current transaction exceeds the total gas limit of the block. If so, we also skip the transaction (Line 16).

Finally, we include the transaction in the block, iterate over its write set, and record the transaction path cost in the resource map *resmap* if its writes increase the critical path. This results in an algorithm that is linear to the number of transactions per block, as the map accesses are $O(1)$ and we check each transaction at most once per block.

4 Evaluation

We implemented Anthemius on top of Block-STM [8] and Chiron [14] in Rust to evaluate its performance impact on both an optimistic execution engine and a guided execution engine, covering two of the most widely adopted approaches to parallel execution in the blockchain space. The implementation is publicly available on Github [3]. As Chiron is built on top of Block-STM, this simplifies the

[3] https://github.com/ISTA-SPiDerS/Anthemius

implementation and allows for an easier comparison of the results. Furthermore, we use the parallel execution benchmarks proposed in Chiron [14].

Finally, we implemented the batch handler ($\sim$ 70 lines of code) and the batch scheduler ($\sim$ 120 lines of code) to assemble blocks and then forward these blocks to the respective execution engines.

4.1 Benchmark

The experiments were executed on a Debian GNU/Linux 12 server with two AMD EPYC 7763 64-Core Processors and 1024 GB of RAM. We generated batches of transactions with different distributions of read/write-accesses and different user distributions with the help of Chiron [14] for all five proposed workloads. Namely, one peer-to-peer workload (P2PTX), two Decentralized Exchange Workloads (DEXAVG and DEXBURSTY), one NFT workload (NFT), and one mixed workload (MIXED). These workloads are derived from real-world data from Ethereum and Solana and are designed to evaluate parallel transaction execution engines under realistic levels of contention. Each workload has a unique and realistic resource access pattern, along with a varying count of read and write operations per transaction.

Each experiment was executed a total of 10 times and the results we outline in this section present the average of all 10 runs. Furthermore, in each workload, we vary the number of worker threads from 4 to 32 in increments of 4. Finally, we are interested in two key metrics: throughput, to assess the performance improvement introduced by ANTHEMIUS, and latency, to determine the average delay introduced by ANTHEMIUS.

We set the following parameters for the batch handler and batch scheduler: First, we evaluate the execution engines using blocks of up to MAXLEN = 10,000 transactions, as this block size represents a sweet spot for both engines, where the execution setup overhead (e.g., virtual machine initialization) becomes negligible. Accordingly, we configured the batch size to match the target block size, as smaller batch sizes increase block construction overhead, while larger batch sizes reduce the batch handler's flexibility to adapt to the workload's characteristics.

Next, to minimize tail latency for transactions accessing hot resources, we allow the first and last LIM = 1,000 transactions to be included freely without restrictions. Furthermore, we permit up to MAXRELAXNUM = 2 relaxations of the inclusion rate as we observed diminishing returns from additional relaxations and large scheduling costs beyond this point. We set the relaxation rate to a maximum of MAXRELAXRATE = 100, targeting an inclusion rate of TARGETINCRATE $= 2\frac{\text{maxlen}}{c}$. This accounts for the higher returns from a more aggressive target inclusion rate as the concurrency potential increases. Finally, we configure MAXHOTR = 4 to avoid uniting too many critical paths of transactions, ensuring manageable contention levels.

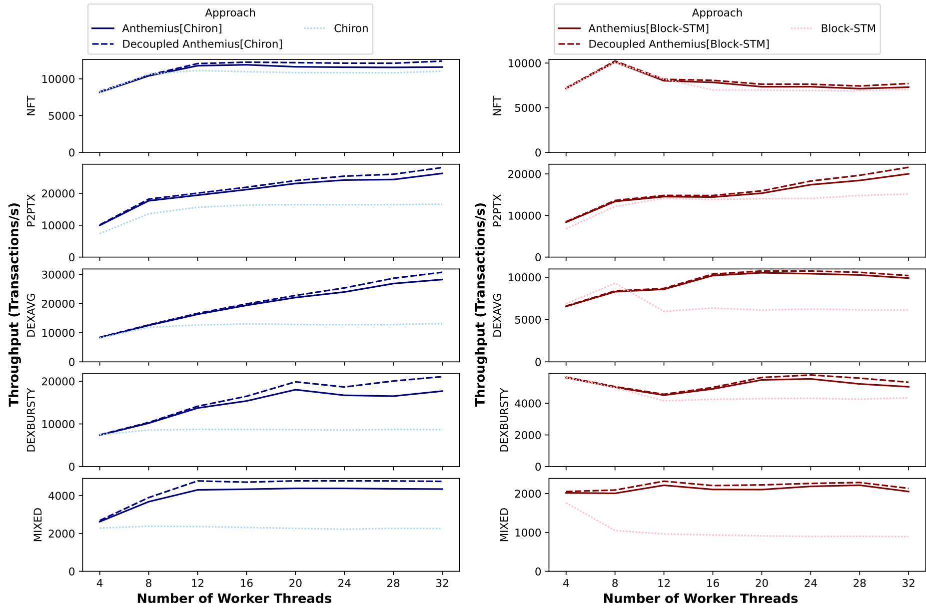
4.2 Throughput

As Anthemius delays the inclusion of some transactions in favor of others to enhance system performance, we provide the batch handler with several batches of 10,000 transactions to saturate the system and measure the maximum throughput. Each batch is generated with the same distribution of resource accesses, both within and across batches. We then evaluate Anthemius by passing all batches to the batch handler and run Anthemius until all transactions from the first batch are successfully executed. Consequently, the evaluation for Anthemius spans multiple blocks, where the reported throughput represents the average throughput over the entire runtime and accounts for scheduling and execution time. For the baseline versions of Block-STM and Chiron, we use a single block containing 10,000 transactions that also fully saturates the system, with runtime variations dependent solely on the specific workload.

As blockchains such as Aptos or Sui decouple consensus from execution, block scheduling could be moved outside of the critical path of consensus. This can significantly reduce the overhead, as scheduling requires only a single thread and only has to be done at the proposer node. Due to this, we display two lines for Anthemius. First, one that serves as a ceiling on performance, where we assume that there is an idle thread that can be used for scheduling outside of the critical path of consensus, denoted *Decoupled Anthemius*. Second, one that serves as a floor on performance where we count the full scheduling overhead on the critical path of consensus, referred to as *Anthemius*.

The results for Anthemius with Chiron are shown in Figure 2a, with the throughput in transactions per second on the y-axis and the number of worker threads on the x-axis. With the NFT workload, we only see a small speedup from creating good blocks. This is due to the account distribution in this workload, where transactions from users appear very frequently in several batches. Due to this, once a transaction of a given user is skipped, the following transactions also have to be skipped, resulting in long scheduling times and leaving very few transactions behind that can be included in the block. In comparison, in the peer-to-peer workload there is already a significant improvement, where with an increasing number of worker threads, we can reach almost twice the initial throughput. Following that, with increasing contention and less repetitive users, the decentralized exchange workloads reach over 240% performance boost compared to vanilla Chiron. While in the average DEX workload, the scheduling overhead is very small, with increasing contention and increasing number of worker threads we can also see an increased scheduling overhead. Finally, in the mixed workload, we also see a large performance advantage. This is also due to the much higher overall execution complexity compared to the scheduling overhead. Due to the complexity of the workload, the overhead is constant after 12 cores, but Anthemius under this workload shows over 200% performance advantage compared to vanilla Chiron.

The throughput results for Anthemius with Block-STM are shown in Figure 2b, with the throughput in transactions per second on the y-axis and the number of worker threads on the x-axis. Compared to the results with Chiron,



(a) Throughput per Second - Chiron (b) Throughput per Second - Block-STM

Fig. 2: Throughput per Second

the results for Block-STM vary more as the high contention within each block results in a large re-execution overhead. As such, even when we build better blocks with ANTHEMIUS, the contention in the block is still so high, that Block-STM struggles to take advantage of that. We can still see the largest disadvantage in the NFT workload, due to the user distribution preventing us from building better blocks. Furthermore, we can see that in the peer-to-peer workload, once we reach 20 threads, ANTHEMIUS is starting to be able to compensate for the re-execution overhead of Block-STM and reach a speed-up of up to 25%. Similarly, for the DEX workloads, there is an initial performance drop due to the re-execution overhead, which is only compensated with more worker threads later. Finally, in the MIXED workload, ANTHEMIUS shows a constant speed up compared to vanilla Block-STM up to 200% the original performance.

4.3 Latency

As we are delaying the inclusion of some transactions that access hot resources, we expect a latency overhead increase at the tail. Similarly to the throughput evaluation, we send several batches of transactions to the batch handler. To fully assess the effect of ANTHEMIUS, we evaluate how the tail latency develops when awaiting the finished execution of up to five batches for all workloads with a fixed number of 16 cores. The results of this evaluation are shown in Figure 3,

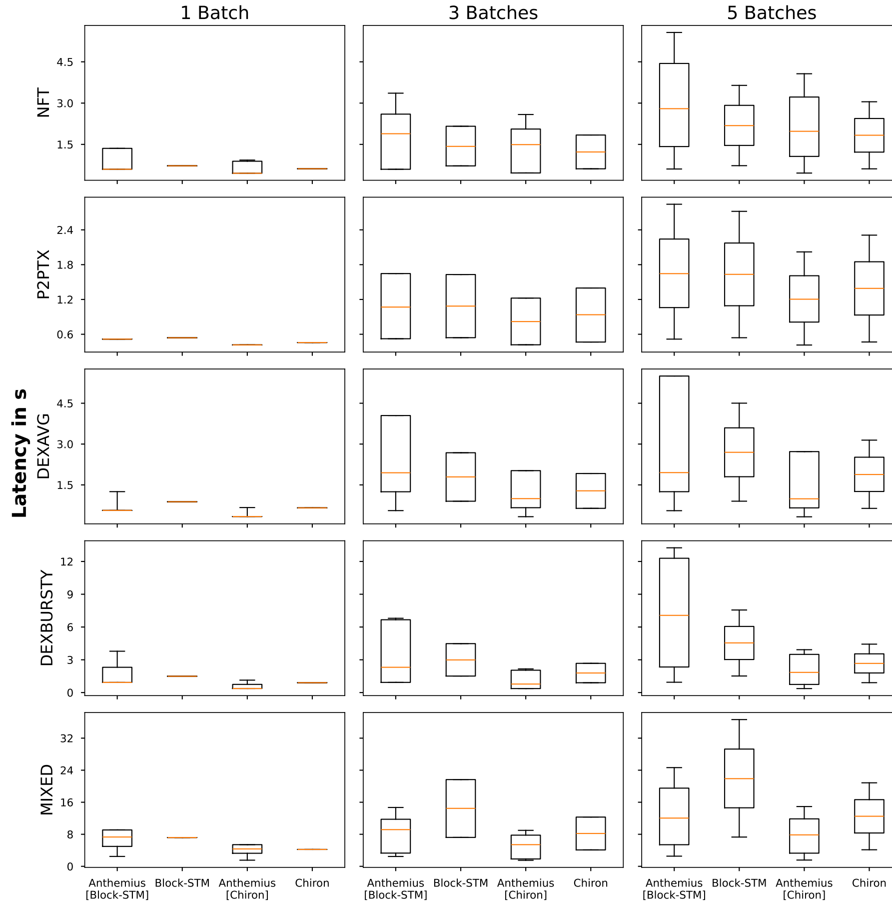


Fig. 3: Tail Latency for Chiron and Block-STM

where the yellow line indicates the 50th percentile (median), the box represents the 25th and 75th percentiles (interquartile range), and the whiskers denote the 10th and 90th percentiles.

The results mirror what we saw in the throughput evaluation where in almost all workloads and configurations where ANTHEMIUS shows a significant speedup the average transaction latency is significantly lower. Furthermore, thanks to the large throughput advantage in these settings, especially when paired with *Chiron*, ANTHEMIUS has a latency advantage for up to the 90% percentile of transactions.

On the other hand, as expected, ANTHEMIUS shows a growing tail latency with an increasing number of batches. This is expected since the congestion caused by the highly contended workloads results in different scheduling decisions. Nevertheless, we can see that the growing tail latency affects not only

ANTHEMIUS but also the reference systems, although for certain workloads the effects of ANTHEMIUS are more prominent at the p90 percentile.

This is a tradeoff the blockchain needs to take into account based on their expected workload and tune ANTHEMIUS parameters to better match the chracteristics of the transactions expected.

4.4 Summary

In this section, we evaluated the throughput improvement ANTHEMIUS can provide across different execution engines. Our findings demonstrate that while ANTHEMIUS improves throughput for both types of execution engines under several of the workloads, its impact is significantly larger when combined with guided execution engines. In this case, ANTHEMIUS provides a large throughput improvement across all but one of the workloads. The only exception is the NFT workload, where many high-frequency users appear across multiple blocks, preventing ANTHEMIUS from effectively rescheduling their transactions.

When it comes to latency, we analyzed the tail latency percentiles of delayed transactions. Our results show that for most workloads the majority of transactions (over 75%) have lower or similar latency compared to the vanilla execution, while only the slowest 25% of transactions sustain a latency overhead. This indicates that ANTHEMIUS can be a valuable addition to any blockchain with a parallel execution engine where the workload does not primarily stem from a very small set of users.

5 Related Work

To the best of our knowledge, there is no academic work proposing algorithms to construct blocks sensitive to parallel execution efficiency. While the problem is an NP-Complete scheduling problem which is explored in theoretical computer science [2], the greedy version of these algorithms still requires polynomial time which would present a large overhead and negate most of the positive effects. By relaxing the optimality requirement, instead, ANTHEMIUS achieves a linear complexity relative to the number of transactions per block.

In the database literature, there are numerous approaches to re-order transactions for reduced abort rates. Most of the work in this context reorders transactions after execution to increase the goodput. Examples of this approach are Aria [12], where an efficient algorithm reorders transactions after execution based on the read and write sets to reduce the number of aborted transactions. Similarly, Sharma et al. [17] focus on execute-order blockchains where transactions are reordered during block construction. While these approaches are efficient and can increase the goodput, none of them consider the parallel execution setting.

Eve [9] is the most similar approach to ANTHEMIUS. In Eve, transactions are organized into batches such that, with high probability, no two transactions within the same batch access the same resource. This allows the execution engine

Table 1: Comparison of existing Block Production Approaches.

Approaches	**Parallel Execution**	**Two-Dimensional Gas Parameter**	**Dependency Sensitive**	**Execution-Time Aware**
Ethereum [3]	✗	✗	✗	✓
Polygon [18]	✓	✗	✗	✓
Aptos [5]	✓	✗	✗	✗
Solana [21]	✓	✗	✗	✓
Anthemius	✓	✓	✓	✓

to execute the block concurrently without having to worry about concurrent accesses during execution. Although the scheduling is very efficient, this approach is unsuitable for blockchain ecosystems where we are expecting a large percentage of transactions to overlap [14] and already have execution engines that can process transactions with dependencies efficiently.

We, therefore, focus on the current state of block assembly in production blockchains. The discussion is summarized in Table 1. While Ethereum [3] does not natively support parallel execution at this moment, it constructs its blocks sensitive to the execution complexity of the smart contracts. The version of Polygon [18] with Block-STM integration supports parallel execution and takes the execution complexity into account. However, it only has a one-dimensional gas parameter and does not take dependencies into account. Aptos [5] supports parallel execution but is unaware of the execution complexity of the transactions at block construction time and only takes the number of transactions and byte size into account. In comparison, Finally, Solana [21] also offers parallel execution and takes the execution complexity into account. However, Solana does not take dependencies into account and only limits the combined computational complexity of transactions of a given client.

Therefore, to the best of our knowledge, Anthemius is the first work proposing a modular and practical algorithm for "Good Block" construction in the context of parallel smart contract execution.

6 Discussion

While Anthemius can achieve a performance boost of over 240%, there are tradeoffs. In this section, we discuss these tradeoffs and potential solutions.

6.1 Malicious Leader

While a correct leader can construct blocks that significantly speed up the system, the opposite is true for malicious leaders. In Anthemius the leader constructs the block sensitive to the number of cores and a gas per core measure, however, no mechanism in Anthemius enforces the leader to construct a block following this blueprint. Even though this might seem like an oversight, it is

impossible to distinguish between a correct leader handling a fully sequential workload and a malicious leader deliberately constructing a sequential block.

This issue is inherent to blockchains that support parallel execution, such as Solana, Sui, or Aptos [21, 19, 5]. Two potential approaches could mitigate this challenge. One approach involves an expensive combination of a fair ordering protocol [10] and a pre-execution stage on the critical path of consensus such as Pompe [22]. Alternatively, leaders could be incentivized through a game-theoretic framework to construct highly parallelizable blocks, with penalties imposed for creating overly sequential ones. However, a detailed analysis of these frameworks is beyond the scope of this work and is left for future research.

Therefore, while Anthemius extends the capabilities of correct nodes to improve the system throughput and empowers them to prevent clients from bottlenecking the system, it does not alter the role a malicious validator could play in the system compared to the state of the art. In fact, due to its modular nature, individual validators on many blockchains could already plug Anthemius into their stack without requiring a hard fork.

6.2 Censorship Resistance

A common concern for leader-based protocols is censorship resistance. In Anthemius, the leader has, as part of the protocol, the power to delay some transactions to speed up the overall system. However, as the leaders in existing protocols such as Ethereum or Aptos already have this power as there is no mechanism that controls this, Anthemius would not hand the leader stronger censorship powers compared to the state of the art.

Nonetheless, protocols focused on short-term censorship resistance such as [20] might not work out of the box with Anthemius. Thus, adjustments to the protocol would be necessary, to only allow a leader to delay a given transaction up to some bounds. This presents a direct trade-off between performance and short-term censorship resistance.

Furthermore, protocols focused on fair ordering, such as [1] often require the transaction and metadata to be encrypted which strips Anthemius of the capability to use transaction meta-data to construct "Good Blocks". Nonetheless, these approaches are also generally incompatible with hint-based execution schemes as used in Chiron, Sui, or Solana.

6.3 Transaction Fees & Client Incentives

While a malicious leader can arbitrarily delay a client transaction, in Anthemius correct nodes might also delay client transactions to improve the overall system throughput. However, in some cases, a client might want their transaction to be included with higher urgency even if it accesses very hot resources, e.g. when a bidding process is approaching the time limit.

Integrating a mechanism with Anthemius that allows client transactions to be included with higher priority is fairly straightforward. Blockchains such as

Bitcoin and Ethereum [13, 3] already use pricing mechanisms to prioritize transaction inclusion. Therefore, a transaction with a higher fee could be transferred to the beginning of the first batch in the batch handler to guarantee its inclusion in the next block. In fact, similar to the local fee markets in Solana [21], this kind of pricing scheme, in combination with ANTHEMIUS would naturally result in a higher price for accessing hot resources, incentivizing smart contract developers to design their smart contracts with concurrency in mind and incentivizing users to avoid hot resources during system congestion times. This can help to balance the system beyond the already existing throughput advantages of ANTHEMIUS.

7 Conclusion

In this work, we presented ANTHEMIUS, a framework, and algorithm to construct highly parallelizable blocks in the context of parallel smart contract execution. We evaluated ANTHEMIUS extensively under a series of realistic workloads, demonstrating a throughput improvement of up to 240%. Furthermore, in most workloads, this approach leads to lower latency for the majority of transactions, while only delaying those that access hot resources and cause bottlenecks. Moreover, ANTHEMIUS not only improves the throughput of the underlying execution engine but also protects blockchains from being bottlenecked by popular applications. Finally, ANTHEMIUS is highly modular and can be easily integrated into any production blockchain without any security tradeoffs.

References

1. Asayag, A., Cohen, G., Grayevsky, I., Leshkowitz, M., Rottenstreich, O., Tamari, R., Yakira, D.: A Fair Consensus Protocol for Transaction Ordering. In: 2018 IEEE 26th International Conference on Network Protocols (ICNP). pp. 55–65 (2018). https://doi.org/10.1109/ICNP.2018.00016
2. Baker, B.S., Coffman, E.G.: Mutual exclusion scheduling. Theoretical Computer Science **162**(2), 225–243 (1996). https://doi.org/https://doi.org/10.1016/0304-3975(96)00031-X, https://www.sciencedirect.com/science/article/pii/030439759600031X
3. Buterin, V.: Ethereum Whitepaper
4. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In: Proceedings of the Seventeenth European Conference on Computer Systems. p. 34–50. EuroSys '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3492321.3519594, https://doi.org/10.1145/3492321.3519594
5. Foundation, A.: Aptos Whitepaper. https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf (2023), accessed on 12.04.2023
6. Fuel Labs: GitHub - FuelLabs/fuel-specs: Specifications for the Fuel protocol, https://github.com/FuelLabs/fuel-specs

7. Garamvölgyi, P., Liu, Y., Zhou, D., Long, F., Wu, M.: Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. In: Proceedings of the 44th International Conference on Software Engineering. ACM (may 2022). https://doi.org/10.1145/3510003.3510086, https://doi.org/10.1145%2F3510003.3510086
8. 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 (2022). https://doi.org/10.48550/ARXIV.2203.06871, https://arxiv.org/abs/2203.06871
9. Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., Dahlin, M.: All about Eve: Execute-Verify Replication for Multi-Core Servers. In: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). pp. 237–250. USENIX Association, Hollywood, CA (Oct 2012), https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kapritsos
10. Kelkar, M., Deb, S., Kannan, S.: Order-Fair Consensus in the Permissionless Setting. In: Proceedings of the 9th ACM on ASIA Public-Key Cryptography Workshop. p. 3–14. APKC '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3494105.3526239, https://doi.org/10.1145/3494105.3526239
11. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 583–598 (2018). https://doi.org/10.1109/SP.2018.000-5
12. Lu, Y., Yu, X., Cao, L., Madden, S.: Aria: A Fast and Practical Deterministic OLTP Database. Proc. VLDB Endow. **13**(12), 2047–2060 (jul 2020). https://doi.org/10.14778/3407790.3407808, https://doi.org/10.14778/3407790.3407808
13. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
14. Neiheiser, R., Babaei, A., Alexopoulos, G., Kogias, M., Kogias, E.K.: CHIRON: Accelerating Node Synchronization without Security Trade-offs in Distributed Ledgers (2024)
15. Neiheiser, R., Matos, M., Rodrigues, L.: Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. p. 35–48. SOSP '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3477132.3483584, https://doi.org/10.1145/3477132.3483584
16. Sergey, I., Hobor, A.: A Concurrent Perspective on Smart Contracts (2017). https://doi.org/10.48550/ARXIV.1702.05511, https://arxiv.org/abs/1702.05511
17. Sharma, A., Schuhknecht, F.M., Agrawal, D., Dittrich, J.: Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric. In: Proceedings of the 2019 International Conference on Management of Data. p. 105–122. SIGMOD '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3299869.3319883, https://doi.org/10.1145/3299869.3319883
18. Team, P.: Innovating the Main Chain: a Polygon PoS Study in Parallelization. https://polygon.technology/blog/innovating-the-main-chain-a-polygon-pos-study-in-parallelization (2022), accessed on 05.12.2022
19. Team, T.M.: The Sui Smart Contracts Platform. https://docs.sui.io/paper/sui.pdf (2023), accessed on 15.01.2024

20. Xue, B., Deb, S., Kannan, S.: BigDipper: A hyperscale BFT system with short term censorship resistance (2023)
21. Yakovenko, A.: Solana: A new architecture for a high performance blockchain v0. 8.13. Whitepaper (2018)
22. Zhang, Y., Setty, S., Chen, Q., Zhou, L., Alvisi, L.: Byzantine ordered consensus without byzantine oligarchy. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). pp. 633–649 (2020)