ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

MICROTRANSACTIONAL BLOCKCHAIN VIRTUAL MACHINE

MASTER THESIS REPORT

Yanick Paulo-Amaro

Supervised by Gauthier Voron and Prof. Rachid Guerraoui

30th June 2023

ABSTRACT

For years blockchain systems have promised to change the world by providing a universal platform able to host any online application in a decentralized, transparent and safe way. In practice, their scale has been limited by the low throughput of the consensus abstraction and virtual machines they are built upon. With recent research improving consensus performance by multiple orders of magnitude, the virtual machine has become the last obstacle in making blockchains capable of supporting truly internet scale applications.

In this context, we present a virtual machine model with relaxed atomicity and isolation guarantees, in which transactions are represented as graphs of micro-transactions, a restricted form of transaction which declares its memory accesses before execution. We compare two scheduling-based implementations of this model using different scheduling algorithms and evaluate them on three applications: a parallel hashmap, a banking app and a synthetic application. Compared to a sequential baseline, we show that using a basic algorithm performs well on all applications but make the VM quite sensitive to contention. On the flip side, using a more advanced algorithm makes the VM more resilient to contention but provide poor performance on applications with random memory accesses. Both implementations achieve close to linear improvement with up to 16 cores on the synthetic workload.

CONTENTS

List of Figures

List of Algorithms

List of Tables

1 INTRODUCTION

Large scale online service are the backbone of the modern internet. They enable people to communicate seamlessly from across the globe, find relevant information and entertainment, enable multiple parties to do business without requiring trust and so much more. These services often use a distributed model to ensure scalability and availability all over the globe. However, building distributed systems is a difficult task as the application logic must now take into consideration all the different parts of the system. Considerable effort is required to make the system scalable while also mitigating the effects of server and network failures. Fortunately, it is possible to build a distributed application in a more straight forward way by using state machine replication (SMR). SMR is a technique to build a decentralized system that provide scalability and fault-tolerance by running multiple copies of an application at the same time. Using a consensus algorithm, the copies are able to agree on the order in which to process external requests, therefore ensuring that their state stays consistent.

Until recently, state-of-the-art consensus algorithms achieved maximum throughputs of 50 to 100 thousand transactions per second. Because of this, most SMR-based applications like cryptocurrencies and smart contracts managed to process these transactions in a timely manner using a single thread. However, new consensus implementations are now able to achieve throughputs of tens of million of transactions per second [1], enough to support even internet scale services. In order to benefit from this new performance, it is clear that single-threaded execution will not suffice.

Just like for distributed applications, making a multi-threaded application by hand is quite difficult and error prone, notably because of conflicting updates from concurrent transactions. Common techniques to ensure correct parallel execution include pessimistic concurrency control, where locks are used to prevent concurrent execution of conflicting transactions and optimistic concurrency control, where transactions work on private copies of the memory and resolve conflicts in an atomic commit phase. Unfortunately, these techniques perform poorly when contention increases and have some drawbacks that make them unsuitable for many applications.

Realizing the intrinsic difficulty of serializing atomic transactions, the goal of this master project is to explore a new model for parallel execution. We propose to relax the atomicity guarantee by splitting transactions into smaller atomic units called micro-transactions. Each individual micro-transaction declares its memory accesses before executing, making it possible to use scheduling to prevent conflicts without runtime checks. Compared to other execution models like Solana's Sealevel [2], this approach supports dynamic memory accesses and does not require transactions to share code in order to execute them in parallel. Additionally, because of the atomicity of micro-transactions, it is possible to recover transaction atomicity by implementing synchronization primitives at the application level. We present two prototype implementations of smart contract virtual machines using this new approach, both using scheduling for concurrency control. We evaluate them against a sequential baseline on three different applications: a parallel hashmap, a banking application and a synthetic, computation heavy application. We show that using the right scheduling algorithm, we can achieve speedups of about 2.8 to 3.6 on the hashmap application, 1.3 to 2.7 on the banking application and 14.8 on an ideal synthetic application, showing the potential of this approach.

After discussing the challenges of parallel transaction processing, we will introduce our proposed solution, micro-transactions, and discuss their limitations. We then describe an architecture for a smart contract virtual machine based on micro-transactions and using scheduling for concurrency control. We present two scheduling algorithms using micro-transactions as the unit of execution and discuss their pros and cons. We then present our experimental results including a microbenchmark of scheduling quality and provide some observations we made while implementing our prototypes. Finally, we present some related work to our project before summarizing our findings and highlighting some opportunities for future work.

2 BACKGROUND

Transforming a single-threaded application into a multi-threaded one comes with lots of challenges. If nothing is done to coordinate the different threads, concurrent memory accesses will cause them to override each other's work leading to unpredictable results. To avoid these issues, it is possible to use locks to protect important memory location and achieve a tight control on the execution of the application. However, if used too eagerly, locks can lead to both deadlocks and livelocks, bringing the whole application to a stop. While using locks more conservatively reduces the risks of these issues, it also leads to poor performance as threads have to wait to access critical sections.

Thankfully, all of this can be avoided by wrapping the sensitive code into a database transaction (or by using a software transactional memory). Database systems guarantee the ACID properties for transactions: Atomicity, Consistency, Isolation and Durability. This ensure that each transaction appears to execute on their own, as single operation. This is achieved by interleaving the operations of each transaction in a way that is serializable, i.e. which is equivalent to some sequential execution of the transactions. Two common techniques to achieves serializability include pessimistic and optimistic concurrency control.
In pessimistic concurrency control, the system assumes that transactions are likely to conflict with each other. Two transactions are conflicting if they executing concurrently and both access the same memory location with one of the accesses being a write. This would mean that their operations might interleave in a way that is not serializable. To avoid this possibility, pessimistic concurrency control requires transactions to acquire locks to (all) memory locations they want to access. By acquiring locks before a transaction start executing and only releasing them after it has completed, this ensures that conflicting transactions are will not be executing concurrently.
In optimistic concurrency control, the system assumes that the transactions are unlikely to conflict with each-other and therefore does not prevent them from executing in parallel. To guarantee serializability despite this, optimistic concurrency control execute each transaction on a separate copy of the memory. Once it is done executing, a transaction attempts to commit its local changes to the main copy of the memory by atomically checking if a conflicting transaction successfully committed while it was running. If that is the case, the transactions is aborted, it discard its local copy and retries executing with a fresh copy of the memory. Otherwise the transaction update the main memory and returns successfully.

While these techniques can guarantee a serializable execution, not all implementations guarantee a *deterministic* execution, which is required if they are to be used with state machine replication. Additionally, both techniques come with serious disadvantages that make them poor choices for a blockchain virtual machine. Pessimistic concurrency control tends to be relatively slow because of all the locks that are acquired. Additionally, if transactions are allowed to make indirect memory accesses, it is possible for deadlocks to occur. In that case, systems using pessimistic concurrency control need additional mechanisms to detect those deadlocks and abort the involved transactions, potentially creating cascading aborts. As for optimistic concurrency control, it requires a massive amount of extra memory to store all local copies of the transactions. This is highly impractical for applications that access lots of memory. Additionally, while it performs quite well when there are few conflicting transactions, its performance degrades catastrophically when contention increases because of transactions repeatedly failing to commit and being restarted.

3 PROPOSED SOLUTION

In this section we present the concept of micro-transactions, a restricted form of transaction and discuss their advantages. We then present the architecture of a parallel smart contract virtual machine using scheduling and micro-transactions. Finally we present two scheduling algorithms and discuss the different trade-offs necessary to make them work.

(A) Original transaction (B) Equivalent micro-transactions

FIGURE 1.1
Example of indirect memory access using two micro-transactions (go to text)

3.1 MICRO-TRANSACTIONS

Most transaction processing systems guarantee that transactions are executed atomically and in isolation, often using some form of concurrency control to ensure a serializable execution, i.e an execution that is equivalent to a sequential one. However, serializability and atomicity are difficult to achieve when transactions are long and have unpredictable memory accesses.

Our solution is to use a restricted form of transaction and to relax the execution guarantees. Similar to how compilers reduce code into basic blocks to simplify analysis and optimization, we propose to compile transactions into smaller units called micro-transactions or transaction pieces. A micro-transaction is an atomic piece of code whose memory accesses are known before it is executed. They can contain branches and loops and can call functions as long as they only access memory that was declared in the micro-transaction's signature. A micro-transaction signature is similar to a function signature but with additional parameters for the memory addresses they access. Just like basic blocks, a micro-transactions can have zero or more successors they can call as part of their last instruction. Using these micro-transactions as building blocks, a transaction becomes a simple pointer to a micro-transaction and some arguments. This restricted form of transaction makes analysis easier and should help achieve higher performance regardless of the concurrency control being used.

In order to achieve even more performance, we also relax the atomicity and isolation properties of transactions. Instead, we guarantee the ACID properties only for micro-transactions. Conceptually, this is like splitting a program into small critical sections at each indirect memory access. This means that this approach should not reduce transaction expressiveness but simply provide a different level of abstraction. For example, a dynamic memory access can be implemented by having one piece resolve the indirection and passing the address to a second piece which will be able to access it (Fig. 1.1). As for transaction atomicity, it can be achieved at the application level by using the atomicity of individual micro-transactions to implement synchronization primitives such as locks and semaphores. Importantly, to avoid deadlocks and ensure the system always make progress, those primitives should "yield" to a new pieces when necessary, for example when waiting on a lock to be released. Of course, it is also possible to implement concurrency control using micro-transactions as the unit of execution to enforce isolation at the system level.

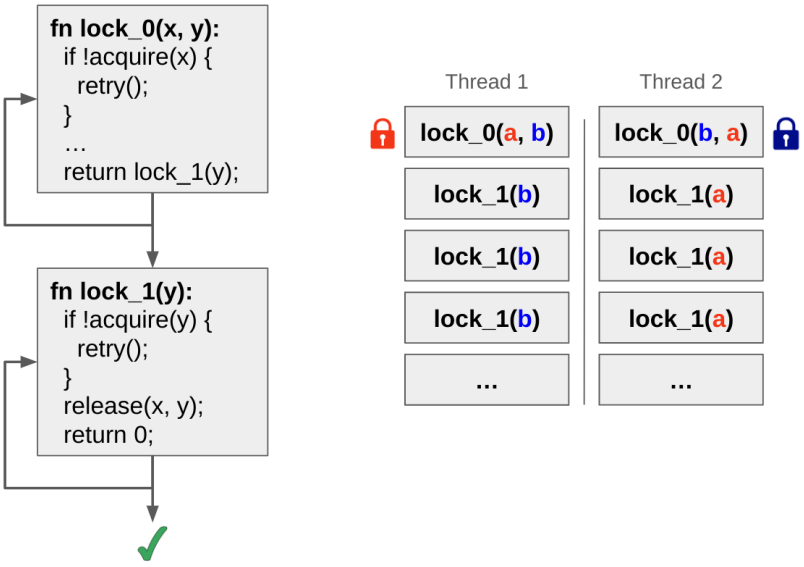


FIGURE 1.2
Example of livelock between two instances of the same micro-transaction (go to text)

LIMITATIONS

This approach does have some limitations that need to be taken into considerations. The first limitation is that systems built using micro-transactions are susceptible to livelocks. Since transaction-level synchronization is handled by the application, an ill-formed program could have concurrent pieces acquire two locks opposite order, causing each piece to wait for each other (Fig. 1.2). In systems where the creation of applications is controlled, this issue can be prevented by analysing the graph of micro-transactions and detecting potential livelocks. When the addition of new transactions cannot be controlled, it is still possible to mitigate this issue by limiting the depth of transactions, for example using gas fees in smart contract systems.

This brings us to the second issue. What happens if a transaction fails after some of its micro-transactions have been executed? This issue has already appeared in database systems when discussing (open) nested transactions and multi-level transactions [3–6]. One possible solution is to use compensating transactions to return the application to a consistent state. However, this only pushes the problem further as compensating transactions themselves could fail. Additionally, because we don't ensure isolation between transactions, a compensating transaction could interfere with concurrent transactions. Another option is to restore the isolation property of transactions by having transactions operate on their own private copy of the memory. This would ensure that failed transactions do not need to be rolled back and could simply be retried. It might even be possible to only retry part of the transaction, similar to using savepoints in between micro-transactions.

3.2 ARCHITECTURE

We decided to use scheduling instead of concurrency control for our implementations because micro-transactions give us a lot of information about conflict prior to execution. On a high level, our system has three parts: a scheduling module, an execution module and a coordination module. The scheduling module takes a block of transactions as input and produces a stream of schedules as output. Each schedule contains a subset of the block as well as information on how to execute it in parallel. The execution module executes each schedule one by one using multiple threads, synchronizing between each schedule.

Once a thread has executed a micro-transaction, it loads the following piece of that transaction and send it to the coordinator so that it can be scheduled. If this was the last micro-transaction, it will send the result of the transaction instead. Finally, the coordination module orders the transaction results and collects the generated micro-transactions to send them to the scheduler. The coordinator is also responsible for sending the initial block to the scheduler and forwarding the schedules to the executor.

Let's look at the system in more details. The scheduling module is composed of S schedulers indexed from 1 to S, running in parallel, each of them outputting its own stream of schedules. When a new block enters the system, the coordinator splits it into even chunks and sends one chunk to each scheduler. It then uses round robin to select the schedule to execute from the incoming streams, waiting on the scheduler until it produces one. If a scheduler has run out of transactions to schedule, the coordinator will simply skip it. Concretely, the coordinator will wait for scheduler 1 to output its first schedule before forwarding it to the execution module. Once the schedule has been executed, it will then wait for the first schedule of scheduler 2 and do the same. Once the first schedule of each scheduler has been executed, the coordinator starts over, selecting the second schedule of each scheduler, then the third one and so on. This has the advantage of letting scheduling and execution progress in parallel, reducing the overall latency of the system. Additionally, streaming schedules from multiple schedulers ensure that the execution almost always has work to do, at least once the first few schedules have been generated.
It is possible for a scheduler to produce less schedules than the others, for example if all transactions in its backlog can be executed in a single schedule. In this case the coordinator will simply skip this scheduler until it has new transactions to schedule.
To ensure the schedulers do not stay idle too long, the coordinator collects the pieces generated by the execution module and regularly send them to be scheduled, splitting them between the schedulers just like the initial block. An important thing to note is that the schedulers do not add the new pieces directly into their current backlog. This would make scheduling non-deterministic since the timing with which new pieces are generated can vary. Instead, the schedulers keep a queue of backlogs which they process one after the other.
The frequency with which the coordinator sends new pieces to the schedulers has a large impact on performance. Every time the coordinator wants to send a set of micro-transactions to be scheduled, it must split it between the schedulers. If it sends new pieces as soon as they are generated then the blocks, and therefore the schedules, will become smaller and smaller and their number will increase exponentially. Since there is a synchronization point after executing each schedule, this will also increase synchronization overhead drastically, resulting in poor performance. To avoid this, we decided to send new pieces at the end of each round robin iteration. That is, we will collect the new pieces until the first schedule of each scheduler has been executed and then split the new pieces among the schedulers. This ensures that the size of the blocks stays roughly the same over time. Another possible approach would be to only split the set of generated micro-transactions if it is large enough. If its size is below a certain threshold it can be given to a single scheduler instead. Finally, once all schedulers have exhausted their pending backlogs, the coordinator knows that all transactions have been executed to completion and can return the results.

3.3 SCHEDULING

In order for the system to perform well, it is important to have a scheduling algorithm that unlocks as much parallelism as possible while introducing as little latency as possible. We will now present two scheduling algorithms, each making different trade-offs.

BASIC SCHEDULING

The first algorithm is very simple. Its objective is to select a subset of the micro-transactions which have no memory conflict to ensure that they can be executed in parallel. We call such a subset a schedule.

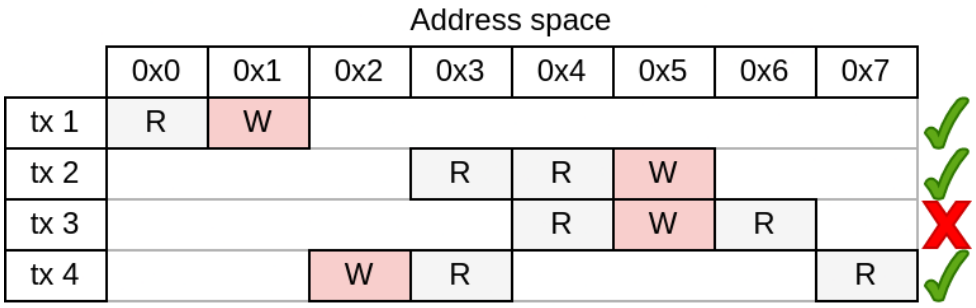


FIGURE 1.3
Example of micro-transactions forming a schedule (go to text)

Schedules are produced by iterating through a block of micro-transactions and selecting the first one that does not conflict with the current selection. A candidate piece is conflicting if it writes to a memory location that has already been read or written to by an already selected micro-transaction or if it reads an address that has been written to. This is illustrated in figure 1.3 which shows four micro-transactions operating in an address space of 8 addresses. The first, second and fourth micro-transactions can form a schedule since they can be executed in parallel while the third one cannot because it conflicts with the second micro-transaction. To avoid comparing the candidate to each selected piece individually, the algorithm aggregates the memory accesses of the current selection and checks if they overlap with the candidate's. Once the whole block has been scanned, the selected micro-transactions are removed from the block and the schedule is sent to the coordinator to be executed. The scheduler then starts over with the remaining micro-transactions until they have all been added to a schedule.

Let's go through the pseudo code of algorithm 1 in more details. The $schedule$ variable contains the micro-transactions currently selected by the algorithm. To keep track of the accesses of the selected pieces, the $accesses$ variable will map each address to its access type. When multiple addresses are passed as argument, it returns their access types as a set. Reads and writes will be marked as R and W respectively, while addresses that haven't been accessed yet are marked as $\perp$. Each candidate micro-transaction tx has a read set $tx.reads$ containing the addresses that it wants to read and a write set $tx.writes$ containing the addresses it wants to writes. Addresses that are both read and written by tx only appear in its write set. When considering a candidate, the algorithm will check the validity of its read and write sets to ensure that it does not conflict with the current selection. The read set is valid as long as it does not contain any addresses that will be modified by an already selected micro-transaction. Similarly, the write set is valid if it does not contain any address that will be read or modified by an already selected piece. If both sets are valid, the mapping is updated with the candidate's read and write sets and the micro-transaction is moved from the backlog to the schedule. Once all micro-transactions have been considered once, the schedule is sent to the coordinator and the algorithm starts preparing a new schedule with the remaining pieces.

One advantage of this algorithm is that the micro-transactions in a given schedule can be executed in any order and still produce a deterministic result because none of them conflict with each other. This means that it is possible to dynamically assign them to threads using work stealing and achieve a load balanced execution.

However, it also has a few disadvantages. First of all, it is a greedy implementation of graph coloring, with vertices being micro-transactions, edges representing conflicts and colors representing schedules. Coloring a vertex corresponds to selecting a micro-transaction and adding it to a schedule. One small difference is that we only compute one color at a time in order to send schedules to the coordinator as soon as possible. Being a greedy makes this algorithm quite fast but also means that it can produce arbitrarily bad results depending on the ordering of the input. Figure 1.4 shows an example of such a

sub-optimal output. The input consists of 11 micro-transactions working within an address space of 8 addresses. The memory accesses of each micro-transaction are displayed below the address space on the left. For example, the first micro-transaction writes to address 0x0 and reads addresses 0x3 and 0x5, the second one writes to 0x0 and reads 0x1 and so on. The output of the scheduling is presented on the right side, each color representing one schedule, meaning all micro-transactions of that color will execute concurrently. Because the basic algorithm greedily includes tx 3 in the first schedule, it ends up having to produce 5 schedules instead of the optimal 3. Figure 1.5 shows the impact of the sub-optimal scheduling on the execution when using 4 cores. Remember that the threads must be synchronized in between each schedule to ensure they do not execute concurrently with one another. In this simple example this means two additional synchronization points for the basic scheduling. This makes execution especially inefficient since the last thread is idle throughout the execution. Notice however that switching tx 3 and tx 4 in the input would make the basic algorithm output the optimal solution. While there is always an ordering that produces an optimal coloring, finding such an ordering for a given graph is NP-Hard (since graph coloring is NP-Complete). Thankfully, many heuristic exist to achieve good approximations in general and for some sub-categories of graphs such as interval graphs, the optimal ordering is known. Additionally, because greedy graph coloring tends to perform well on sparse graphs, applications with low contention should generally be scheduled efficiently and effectively.

The second disadvantage of this algorithm is that it is lazy. When it encounters a conflict it immediately gives up and postpone the conflicting micro-transaction, forcing it to be added to a later schedule. This is detrimental to applications which frequently access a hot section of memory because it will cause each piece to be added to its own schedule. Having many small schedules increases synchronization overhead drastically and has a negative effect on performance. Finally, because this algorithm keeps track of the memory accesses of each micro-transaction, it will perform poorly for memory intensive applications, especially if they access many disjoint memory locations.

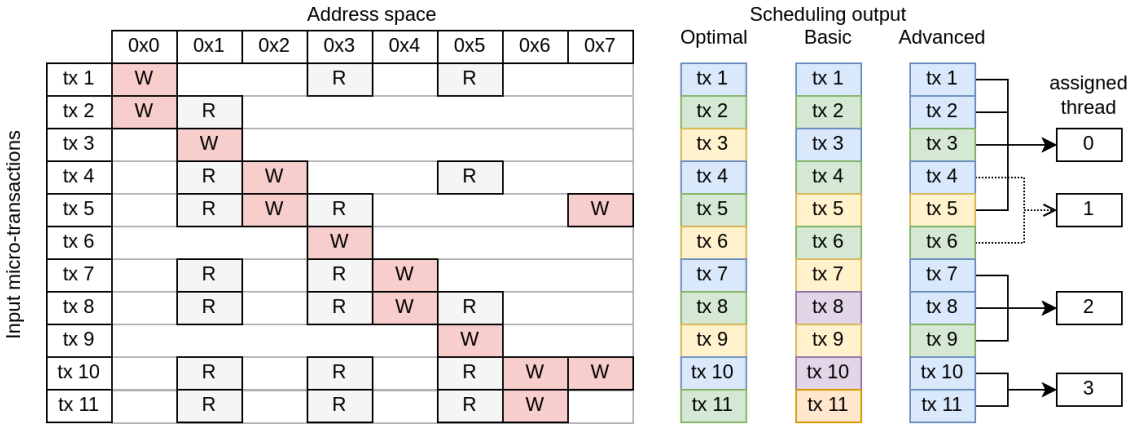


FIGURE 1.4
Scheduling output using basic and advanced algorithms (go to text)

ADVANCED SCHEDULING

The second algorithm improves on the first one by taking into account which worker thread will execute which micro-transaction. Instead of being added to a set, selected micro-transactions will be added to the work queue of a particular thread. Just like before, the algorithm will check if a candidate conflicts with already selected pieces. However, when a candidate conflicts with only a single queue, it is selected and added to that queue. The idea is that since a work queue is executed sequentially by a single thread, all micro-transactions in that queue will naturally be serialized, ensuring that those conflicts to not cause memory inconsistencies. This is very beneficial for applications with very segmented memory accesses.

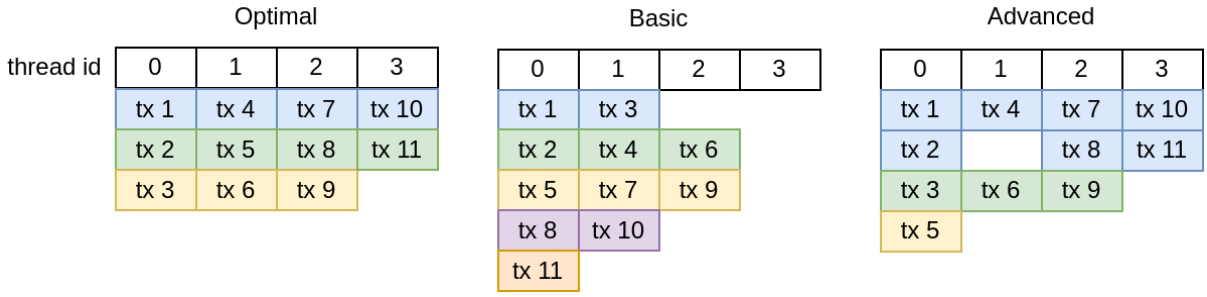


FIGURE 1.5
Execution of micro-transactions following basic and advanced scheduling (go to text)

The pseudo code of this more advanced scheduling is presented in algorithm 2. The $schedule$ variable is replaced by n queues, one for each execution thread. Like in the basic algorithm, the $accesses$ variable is used to keep track of the memory accesses of the selected pieces. Reads are still marked as R but this time writes are marked as W_i, where i is the index of the thread that will access that address. When an address is marked as W_i, we say that it was claimed by thread i. Checking the validity of a candidate's read set is the same as before but checking the write set is a little bit more complicated. If all the writes of the candidate have been claimed by a single thread, we know that it will only conflict with micro-transactions in that thread's queue and we can therefore assign it to that thread. On the other hand, if none of the writes have been claimed, we know that the candidate will not conflict with any selected piece and we can assign it to any queue. We chose to select the queue using round robin to have some form of load balancing. However, depending on the distribution of the memory accesses, some queues might be longer than others despite using round robin. This could be mitigated by having more queues than available threads and letting threads pick a work queue dynamically at runtime. This would still be deterministic because the queues do not conflict with each other and can therefore be executed in order. Going back to the algorithm, if neither of these conditions hold, the candidate is postponed. This can happen when it tries to write an address that is read-only or if one address was claimed by two different threads i and j. This can also happen when only subset of the addresses have been claimed by a single thread i. This is not quite a memory conflict and the candidate could be assigned to that thread without causing memory corruption. However, such a candidate should not be selected as it could create the worst schedule possible on some inputs, as shown in figure 1.6. In the incorrect scheduling, candidates are selected even if they extend the memory region assigned to a thread. Because each micro-transaction in the input overlap slightly, the incorrect scheduler will assign all of them to thread 0 in a single schedule, leading to a completely sequential execution.

The advantage of this algorithm is that it produces larger schedules when there is contention on some memory region compared to the basic version. This reduces synchronization overhead because less schedules are generated and it reduces scheduling latency because the backlog shrinks faster. Looking at figures 1.4 and 1.5 again, we can see that the advanced algorithm only produces 3 schedules and offers a more even execution compared to the basic scheduling.

One disadvantage of this algorithm is that reads are never assigned to threads, they always make the address read-only. This means that even if an address is read by only one thread, no candidate will be able to write to that address in the current schedule, even if it could be assigned to that thread. Similarly, if a write was claimed by a thread, no candidate will be allowed read that address, even if they do not make conflicting accesses with other threads. Additionally, this is still a greedy algorithm and therefore shares the associated disadvantages.

Algorithm 1 Basic scheduling algorithm (go to text)

Input: A list of micro-transactions B
Output: A stream of schedules

while $B \neq \varnothing$ **do**
  $schedule \leftarrow \varnothing$
  $accesses \leftarrow$ empty mapping (default $\perp$)
  for all tx *in* B **do**
    if $W \in accesses(tx.reads)$ **then**
      continue
    end if
    if $R \in accesses(tx.writes) \vee W \in accesses(tx.writes)$ **then**
      continue
    end if

    $accesses(tx.reads) \leftarrow R$
    $accesses(tx.writes) \leftarrow W$
    remove tx from B
    $schedule \leftarrow schedule \cup tx$
  end for
  Send $schedule$
end while

Algorithm 2 Advanced scheduling algorithm (go to text)

Input: A list of micro-transactions B
Output: A stream of schedules

$n \leftarrow$ number of execution core
while $B \neq \varnothing$ **do**
  $queues[i] \leftarrow \varnothing, i = 1, ..., n$
  $accesses \leftarrow$ empty mapping (default $\perp$)
  for all tx *in* B **do**
    $k \leftarrow \perp$
    if $W\_ \in accesses(tx.reads) \vee R \in accesses(tx.writes)$ **then**
      continue
    end if
    if $accesses(tx.writes) = \{W_i\}$ **then** ▷ All writes are claimed by thread i
      $k \leftarrow i$
    else if $accesses(tx.writes) = \{\perp\}$ **then** ▷ None of the writes have been claimed yet
      $k \leftarrow$ round robin
    else
      continue
    end if

    $accesses(tx.reads) \leftarrow R$
    $accesses(tx.writes) \leftarrow W_k$
    remove tx from B
    $queues[k] \leftarrow queues[k] \cup tx$
  end for
  Send $queues$
end while

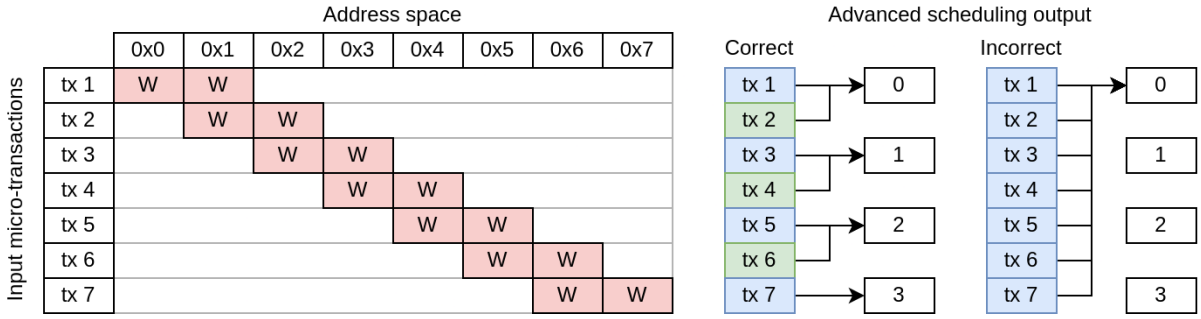


FIGURE 1.6
Example of incorrect advanced scheduling (go to text)

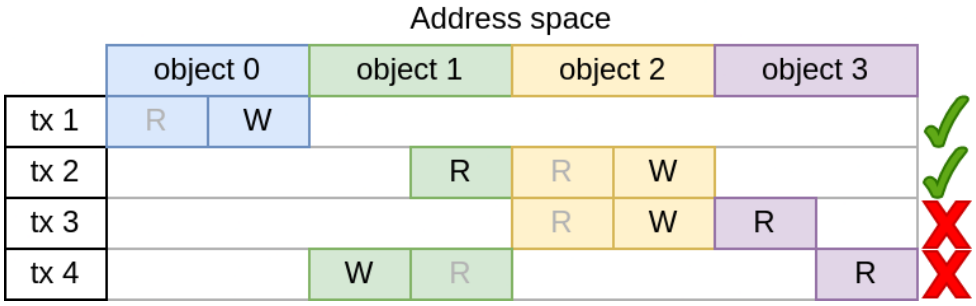


FIGURE 1.7
Example of coarse grained scheduling (c.f. figure 1.3) (go to text)

SCHEDULING GRANULARITY

Because both of these algorithms keep track of every address accessed by each micro-transaction, they can be quite slow. To make them more efficient, we changed the granularity of the scheduling to keep track of memory objects instead of memory addresses. For example, when selecting tx 1 we would only track a single write to $object$ 1 instead of a read to address 0x0 and a write to 0x1. By having a more coarse grained tracking, we are able to make scheduling much faster in exchange for a lower accuracy. Figure 1.7 illustrates how this approaches reduces the accuracy of the algorithm. In this coarse grained version, tx 4 is not scheduled together with the first two micro-transactions because it writes to an *object* that tx 2 is reading. In contrast the fine-grained scheduling presented in figure 1.3 was able to schedule them together. We believe this trade-off is worth it because the loss in accuracy should be small for most applications.

4 EVALUATION

4.1 SETUP

To validate the usefulness of our approach, we implemented two prototypes, one using the basic scheduling algorithm and one using the advanced algorithm. The two implementations are available on our github repository [7] and are simply called *BasicPrototype* and *AdvancedPrototype*. Both of them follow the architecture presented above but the second one is still work in progress and is a little bit different to accommodate the different schedule structure. We evaluated their performance on three different applications summarized in table 1.1 and compared the results to a sequential baseline. For each application, we ran the benchmark with varying number of schedulers and execution threads. All scheduling threads and execution threads were pinned to a distinct vCPU core. For each measurement, we process 1000 batches of 65536 transactions to measure the mean latency and compute the throughput.

Application	Micro-transaction	Size	Computation	Memory accesses	Potential parallelism
Hashmap	ComputeHash	72 B	Medium	Low	Very high
	Get	72 B	Low	Medium	Medium
	Insert	72 B	Low	Medium	Medium
	Remove	72 B	Low	Medium	Medium
	Resize	72 B	High	High	None
Banking	Transfer	32 B	Low	Low	High
Compute	Fibonacci	24 B	High	None	Very high

TABLE 1.1
Summary of applications (go to text)

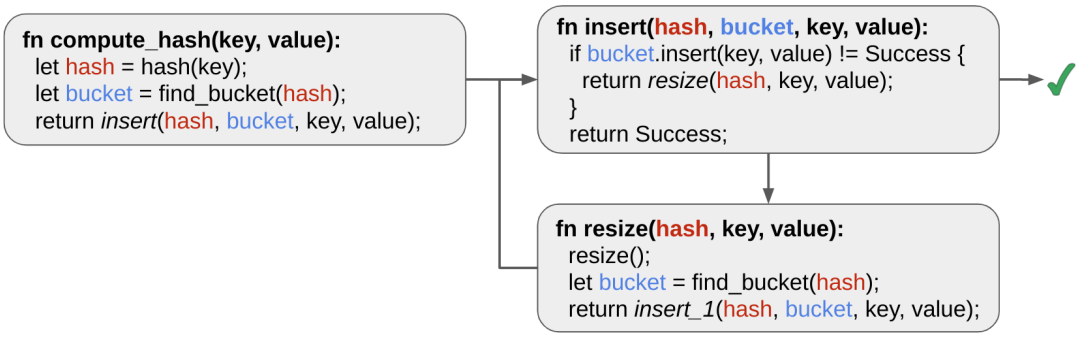


FIGURE 1.8
Decomposition of hashmap insertion (go to text)

Transactions contain the address of the sender, a "pointer" to the first micro-transaction to execute, the index of the transaction in the batch, parameters for the micro-transaction being called and the addresses that it is going to access. Since each application has a different number of addresses and parameters, their transactions have different sizes. We benchmark our prototype on an AWS c6i.8xlarge instance with 32 vCPU and 64 GiB of RAM running Ubuntu 22.04 with kernel version 5.19.0-1027-aws. We implemented our prototypes in rust 1.70.0 using standard library threads and $crossbeam$ 0.8.2 for channel communication. We used $ahash$ 0.8.3 for a fast hashing during scheduling and $blake3$ 1.3.3 for the cryptographic hashes in the hashmap application.

4.2 HASHMAP

The first application is a hashmap using blake3 cryptographic hash function. It supports search, insertion and deletion operations and each one is decomposed into multiple pieces. Figure 1.8 shows the decomposition of the insert operation, the other decompositions are analogous. The first piece computes the hash of the key for the operation and find the corresponding bucket while the second piece performs the operation on that bucket. The insertion operation has an optional third piece which takes care of increasing the hashmap capacity once a bucket is full. The hashmap is implemented as a continuous chunk of memory, starting with the hash table and followed by the buckets. Each bucket is an array of fixed length with a sentinel value marking the bottom of the bucket. When searching for an entry, the given bucket is scanned sequentially until the requested key is found or the end of the bucket is reached. When an entry is removed, is it simply marked as empty with another sentinel value. In terms of scheduling, each bucket is considered as a single memory object. The size of map entries, the initial number of buckets, the capacity of buckets as well as the proportion of operations can be configured to create different workloads.

Low contention (1024 buckets)

For the first workload, the hashmap application was configured to use 64 byte entries and to start with 1024 buckets of 64 entries. The proportion of operations is 90% searches, 5% insertions and 5% deletions.

Figure 1.9 shows the throughput of our prototype in million of transactions per second against the number of cores used for parallel execution. Starting with the basic scheduling algorithm (Fig. 1.9a), our prototype achieves only 7.3 million transactions per second with a single scheduler and 2 cores, compared to 6.15 million for the sequential baseline. As we increase the number of execution cores, the performance increases up to about 13 million transactions per second with 16 and 20 cores. With 2 schedulers we see a similar scaling and even better performance, as our prototype goes from 9.3 million tx/s with 2 cores to 17.9 million with 12 cores. This scaling does not continue with 4 and 8 schedulers as performance goes down significantly after 8 cores. Overall a peak throughput of 18 million tx/s is achieved with 4 schedulers and 4 cores which is a 2.9 times speedup over the sequential version.

Switching to the advanced scheduling algorithm (Fig. 1.9b), we see a more homogeneous performance across the board. With 1 and 8 schedulers the performance is slightly better, both reaching a maximum throughput of 15 million tx/s with 16 cores. Using 2 and 4 schedulers leads to very similar scaling with performance increasing up to a maximum of 16.1 and 17.1 million tx/s respectively. This is a little bit less than with the basic scheduling but the robust scaling is a good sign for future optimized implementations.

Low contention (64 buckets)

In the second workload, presented in figure 1.10, the distribution of operations is the same but the hashmap is configured to start with only 64 buckets, increasing the probability of the map resizing due to buckets becoming full.

The additional work needed to resize the map is reflected in the throughput of the sequential version going down to about 5.5 million tx/s, compared to 6.15 previously (-11%). When using the basic scheduling algorithm our prototype is even more impacted, with performance going down by 56% with 1 scheduler and by 36%, 34% and 37% with 2, 4 and 8 schedulers respectively (Fig. 1.10a). One reason for this drop in performance is that resizing the map requires exclusive access to all buckets simultaneously, preventing other transactions from making progress. Additionally, because the number of buckets is initially lower the contention is higher, leading to lower performance.

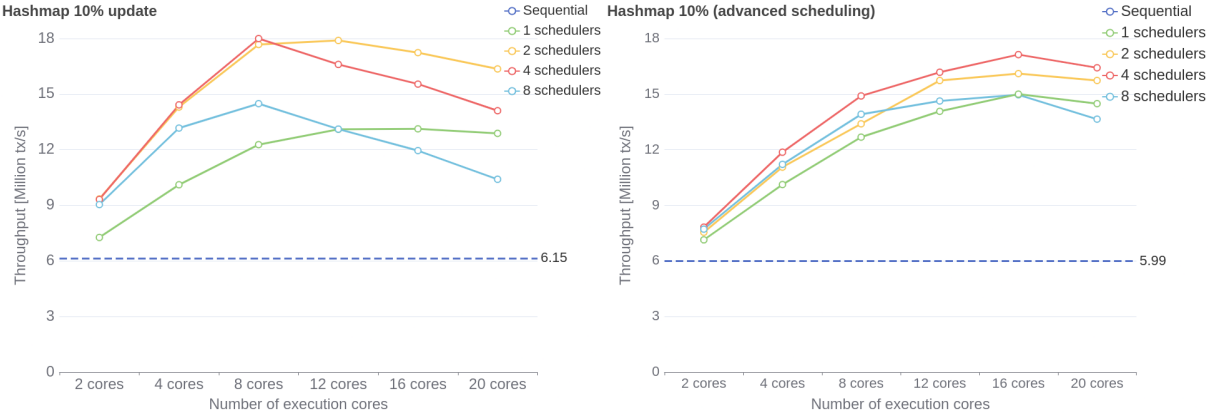
When using the advanced scheduling however, our prototype is able to maintain the same level of performance as before, achieving a maximum throughputs of 16.8 million tx/s with 4 schedulers and 16 cores (Fig. 1.10b). This is thanks to the schedulers being able to assign groups of conflicting micro-transactions to a single thread instead separating them into multiple schedules.

High contention (1024 buckets)

In this third workload the hashmap starts with 1024 buckets again but has a more challenging distribution of operations containing 50% searches, 25% insertions and 25% deletions. This means that each bucket is expected to have 32 transactions trying to modify it at the same time, creating a lot of contention.

Using basic scheduling, our prototype struggles to deal with the additional contention (Fig. 1.11a). With a single scheduler it shows no scaling at all and achieves a throughput of only 2.6 million tx/s with 12 cores, much lower that the 5.44 million tx/s of the baseline. Using 2, 4 and 8 schedulers our prototype is able to overcome the baseline with peaks of 8.8, 11.8 and 10.2 million tx/s respectively, resulting in a 1.6 to 2.2 times speedup.

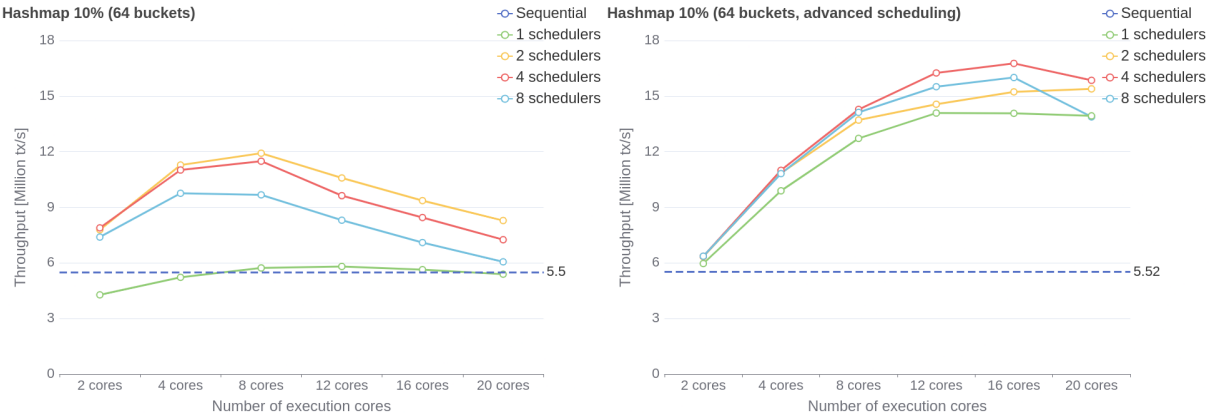
In contrast, using the advanced algorithm our prototype is able to maintain decent performance despite



(A) Basic scheduling

(B) Advanced scheduling

FIGURE 1.9
Throughput on hashmap application with 1024 buckets and 10% of updates (go to text)



(A) Basic scheduling

(B) Advanced scheduling

FIGURE 1.10
Throughput on hashmap application with 64 buckets and 10% of updates (go to text)

the high contention. With 16 cores it achieves between 11.6 million and 14.2 million tx/s, a 2.2 to 2.7 times speedup over the baseline.

To better understand where the difference in performance between the two algorithms comes from, let's look at the latency of the system. Figure 1.12 shows an approximate breakdown of the latency of our prototype on this workload with 4 schedulers. In blue we have the total scheduling time of the slowest scheduler and in green the total time spent executing schedules. In both cases, we do not include the time spent waiting for inputs. In yellow we have the difference between the total latency of the system and the previous two latencies. This is the latency of the coordinator and includes the time spent book keeping and synchronizing with the schedulers and executors. In reality the scheduling and execution latency do not contribute as much to the overall system latency because they are executing in parallel. As we can see, both systems have comparable scheduling latency and the same execution latency. However their coordination latency is quite different, it is constant when using the advanced algorithm but scales with the number of cores when using the basic one, causing its performance to drop with more than 8 cores. It is likely that the extra synchronization latency caused by the additional cores is being multiplied by the high number of schedules of the basic algorithm. On the other hand, the advanced algorithm is able to avoid this increase by producing a single schedule.

HIGH CONTENTION (64 BUCKETS)

Keeping the high contention workload, we once again reduce the number of buckets from 1024 to 64. This means that the hashmap will have an average of 512 insertion/remove operations per bucket initially.

Using basic scheduling, our prototype struggles even more to deal with such high contention (Fig. 1.13a). With 1 scheduler, it achieves a constant 1.4 million tx/s regardless of the number cores used for execution and with 2 and 8 schedulers it achieves a maximum throughput of about 7.3 million tx/s with 8 cores. With 4 schedulers, it achieves a peak throughput of 8.5 million tx/s which is a 2.3 times speedup over the baseline.

On the other hand, reducing the number of buckets has negligible impact when using the advanced scheduling (Fig. 1.13b). With 1 scheduler, it still achieves up to 10.7 million tx/s, even outperforming the basic implementation with 4 schedulers. A maximum throughput of 14.5 million tx/s is achieved with 4 schedulers and 16 cores, leading to a 3.6 times speedup over the baseline.

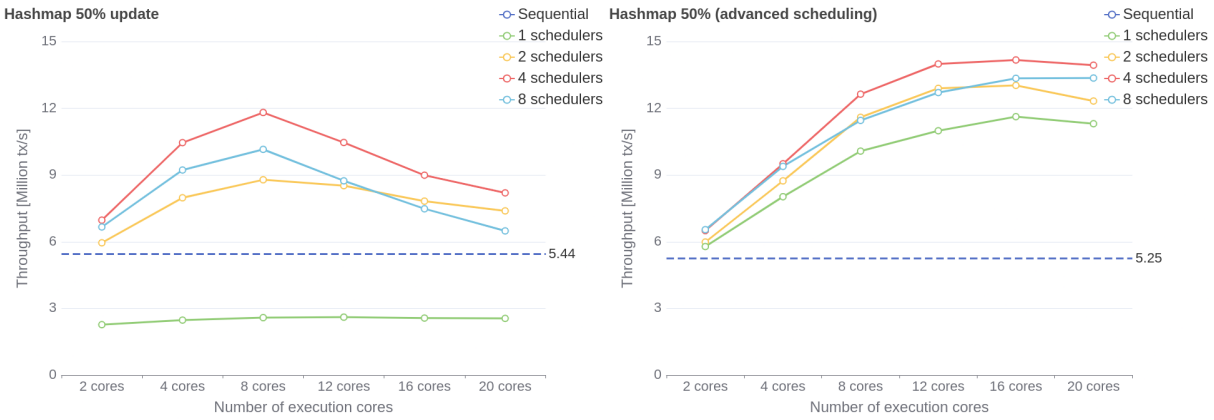
4.3 BANKING APPLICATION

The second application is a simple banking app which transfers funds between different accounts. Each account has an initial balance large enough to ensure it will not run out of funds during the benchmark. The transfers are implemented as a single micro-transaction. The difficulty of the workload can be configured by choosing the proportion of conflicting transfers. Conflicts are generated by having some transfers share the same recipient.

NO CONTENTION

For the first workload, each transfer involves a unique pair of sender and receiver, ensuring that there are no conflicts.

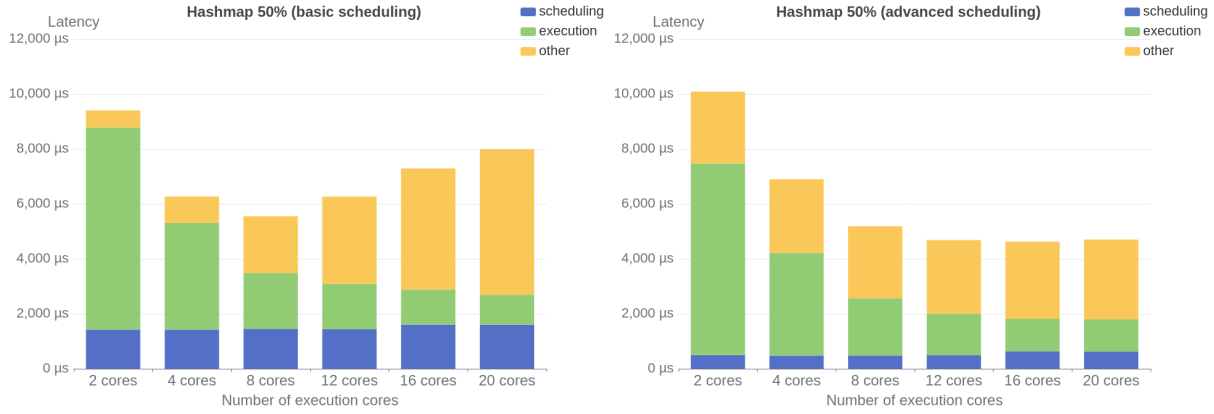
With basic scheduling, figure 1.14a shows our prototype having a modest scaling across the board with 1, 2 and 4 schedulers reaching plateaus of 25.3, 46.8 and 73.6 million tx/s respectively. Using 8 schedulers provide slightly better scaling with performance going from 51.6 million tx/s with 2 cores to 89 million with 8 cores. With a sequential baseline of about 32.6 million tx/s, this translates to a decent 2.7 times speedup.



(A) Basic scheduling

(B) Advanced scheduling

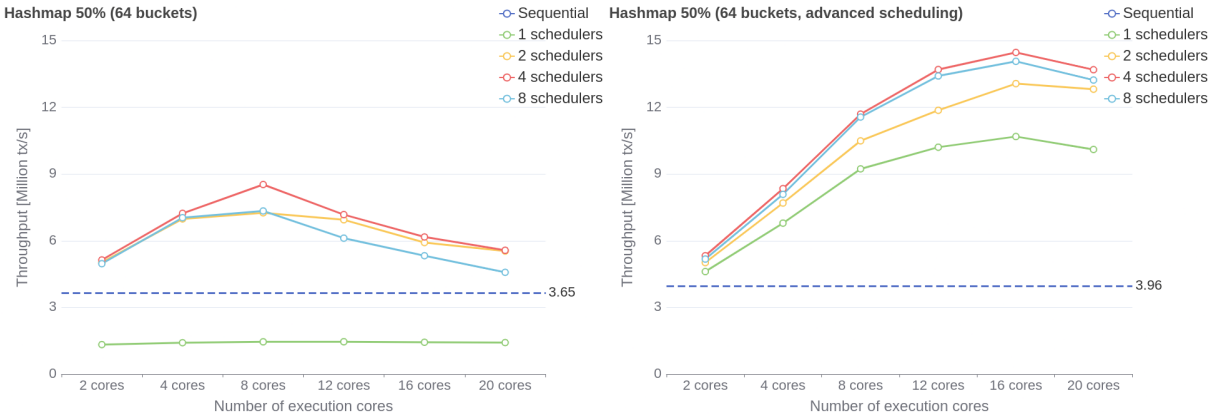
FIGURE 1.11
Throughput on hashmap application with 1024 buckets and 50% of updates (go to text)



(A) Basic scheduling (4 schedulers)

(B) Advanced scheduling (4 schedulers)

FIGURE 1.12
Latency breakdown on hashmap application with 1024 buckets and 50% updates (go to text)



(A) Basic scheduling

(B) Advanced scheduling

FIGURE 1.13
Throughput on hashmap application with 64 buckets and 50% of updates (go to text)

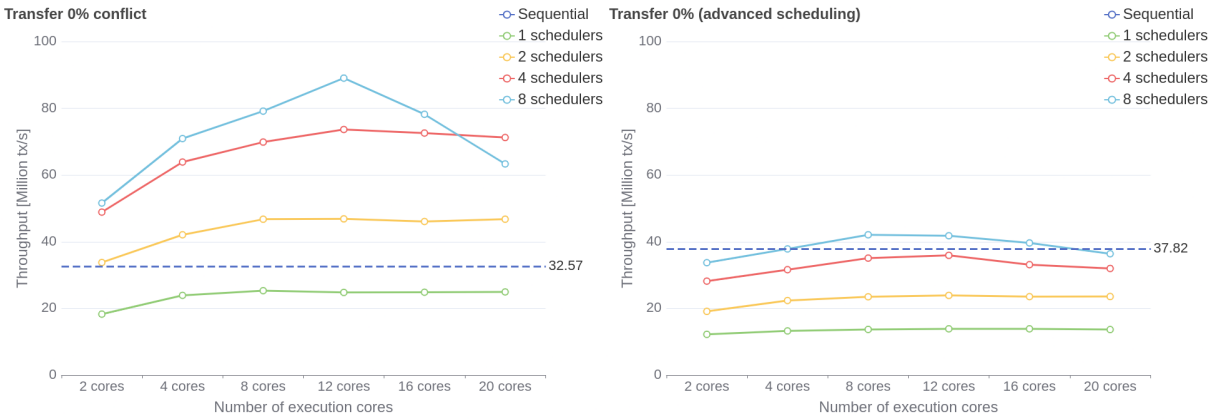
On the other hand, using the advanced scheduling has leads to very poor performance, regardless of the number of schedulers (Figure 1.14b). With 1, 2 and 4 schedulers the throughput is 46%, 49% and 52% lower than the basic scheduling and does not surpass the 37.8 million tx/s of the baseline. Even with 8 cores, it achieves a maximum of only 42.1 million tx/s which is only 1.1 times faster. The reasons for this huge disparity in performance is that the advanced algorithm is much slower than basic one. With the hashmap application it was able to compensate for this by producing better schedules but it is unable to do so here because there is no contention.

Comparing the latency breakdown in figure 1.15, we can see that while both versions have the same execution latency, the scheduling and coordination latency are much higher when using the advanced scheduling. The difference in scheduling latency is to be expected since the advanced scheduling is heavier than the basic one but such a high coordination latency is a little bit surprising. One possible cause is the difference in data structures and architecture between the two implementations. In the basic implementation the coordinator assigns work to schedulers by sending each one a reference to the current block and an index range indicating the portion it is responsible for scheduling. When they produce a schedule, the schedulers simply send a reference to the block and a list containing the indices of the scheduled micro-transactions. Then the coordinator sends each executor a copy of the schedule and an index range indicating which portion of the schedule it is responsible for executing. Overall, this means that the coordinator does not have much work to do outside of synchronization.
On the other hand the advanced implementation uses a different structure. This time, the coordinator actually splits the block into smaller chunk to assign work to schedulers. The schedules now contain multiple lists of indices for the different work queues and the read-only micro-transactions. The indices of postponed micro-transactions are also included in the schedule so that they can be sent back to the schedulers along with the new micro-transactions. This was done in an attempt to reduce allocations in the critical path of the system but has the side effect of increasing the work to be done on top of synchronization. Another difference is that instead of sending them only to the coordinator, the schedulers send copies of the schedules directly to the execution cores. Because the schedules assign micro-transactions to cores, this allows the coordinator to synchronize execution simply by sending signals to the executors instead of splitting the schedules itself. However this means that there are more inter-core communication which increases synchronization latency.

HIGH CONTENTION

In this second workload, half of the transfers have an account in common with some other transfer. As shown in figure 1.16a, the sequential baseline performs much better with this workload, achieving a throughput of 39.8 million tx/s compared to 32.6 million in the previous workload. This difference was consistent across multiple re-runs. We think this is due to this workload having fewer distinct accounts since some of them are shared among conflicting transfers. In contrast, our prototype performs worse with the additional contention. Using 1 or 2 schedulers it achieves lower throughput than the baseline with 20.3 and 36.2 million tx/s respectively. Using 4 or 8 schedulers our prototype finally outperform the baseline with a maximum throughput of 47.2 and 50.4 million tx/s respectively, leading to a 1.3 times speedup. However this also leads to worse scaling as performance drops when using more than 4 or 8 cores.

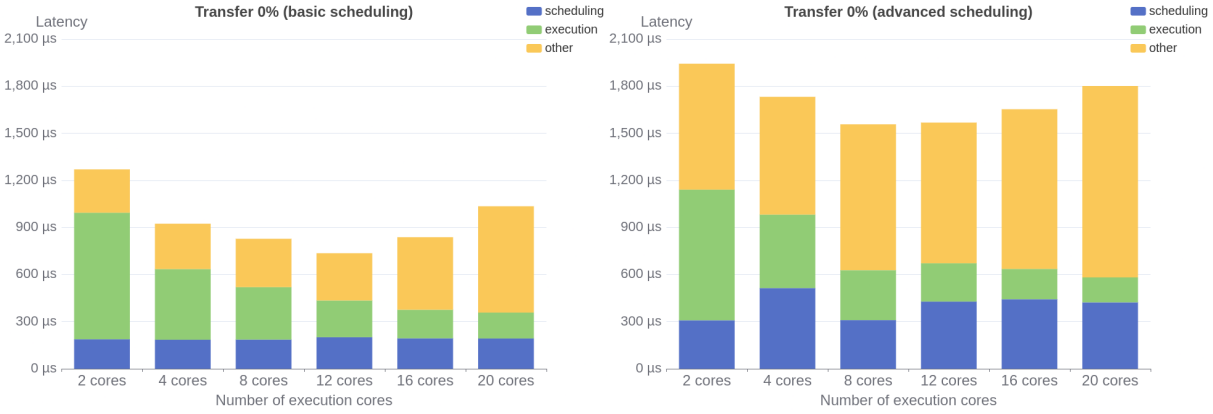
With the advanced scheduling the situation is even worse (Figure 1.16b). With a maximum throughput of only 30.8 million tx/s, our prototype is outperformed by the sequential implementation regardless of the number of schedulers. The problem is that unlike with the previous application the advanced scheduling is unable to take advantage of the high contention. This is because the memory accesses of the two applications are very different. With the hashmap, all operations accessed very specific addresses and either did not overlap or overlapped exactly, which the advanced scheduling was able to make good use of. With the banking application, transactions can access a wide range of addresses and conflicting transactions only overlap partially, preventing the scheduler from assigning them to the same thread.



(A) Basic scheduling

(B) Advanced scheduling

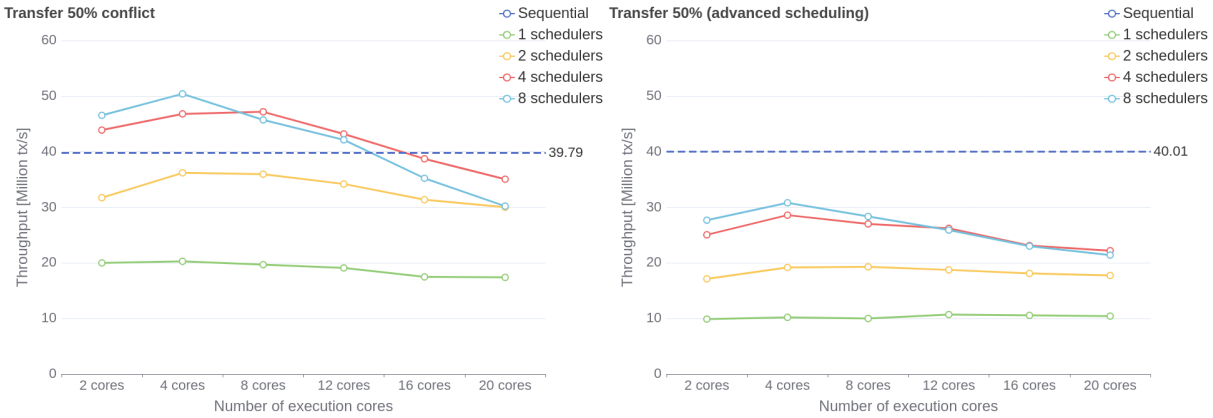
FIGURE 1.14
Throughput on banking application without conflicts (go to text)



(A) Basic scheduling (8 schedulers)

(B) Advanced scheduling (8 schedulers)

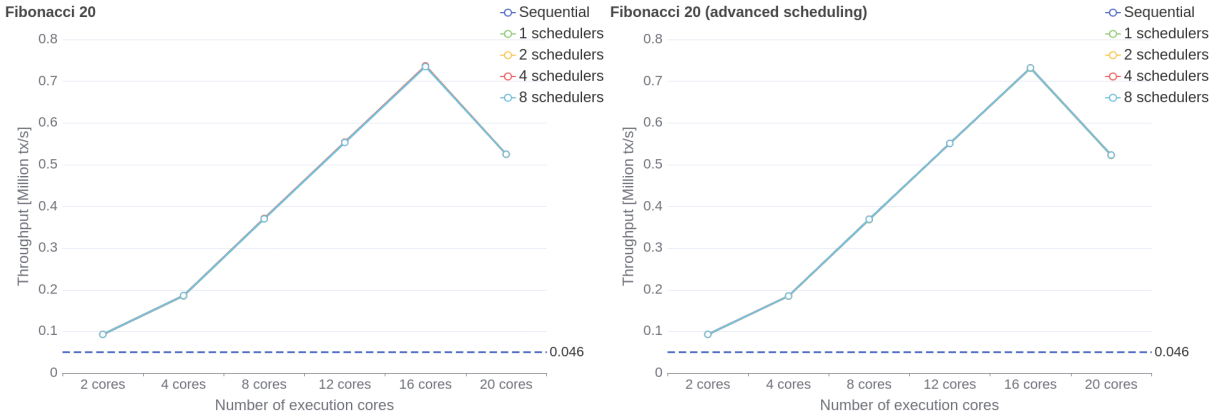
FIGURE 1.15
Latency breakdown on banking application without conflicts (go to text)



(A) Basic scheduling

(B) Advanced scheduling

FIGURE 1.16
Throughput on banking application with 50% conflicts (go to text)



(A) Basic scheduling

(B) Advanced scheduling

FIGURE 1.17
Throughput on heavy computation application (go to text)

4.4 COMPUTATION

The third application simulates a highly parallel workload containing a heavy computation. This is done by having each transaction compute the n-th Fibonacci number naively and making no memory accesses. The difficulty of the workload can be configured by choosing which Fibonacci number to compute. This is an ideal case for a parallel processing system and we expect our prototype to perform well. We chose $n = 20$ for this workload so that it would stress the sequential baseline as much as possible without taking forever to benchmark. Looking at figure 1.17, we can see that the performance of our prototype is the same regardless of the number of schedulers and the scheduling variant. This is because scheduling transactions that don't access memory is trivial in both algorithms making the scheduling latency negligible compared to the execution. With a sequential baseline of 46'000 tx/s, our prototype already achieves a 1.8 times speedup with a throughput of 90'000 tx/s with 2 cores. As we add more cores the performance increases to 190'000, 370'000 and 550'000 tx/s for 4, 8 and 12 cores respectively. With 16 cores the throughput peaks at 740'000 tx/s before going down to 530'000 tx/s with 20 cores. This is close to a linear scaling in performance with a maximum speedup of 14.8 with 16 cores.

4.5 MICROBENCHMARKS

In addition to benchmarking the whole system, we also ran a micro-benchmark to evaluate the scheduling algorithms in isolation. For each application, we scheduled a batch of 65536 micro-transactions and tracked the progress of the algorithm by measuring the latency and size of each schedule being produced. This will help us understand how efficient the algorithm is by knowing how long each schedule takes to produce and how much it contributes to the scheduling[1]. We visualize these measurements using a latency-completion graphs, where the y axis measures the latency and the x axis measures the progress towards completion. Each data point n indicate the latency of producing the first n schedules and what percentage of micro-transactions have been scheduled at that moment. For example, in figure 1.18 you can see that it took 2000 µs to produce the first schedule of the banking workload with 50% of conflicts (in green) and that it contained 72% of the micro-transactions that needed to be scheduled. Producing the second schedule took an additional 550 µs and contained 22% of the micro-transactions, bringing the completion to 94%. Finally, we can see that after producing 3 schedules, 99% of all the micro-transactions have been schedule and that it took 2600 µs in total.

[1]For the hashmap workloads we only measured the latency of the second piece since the first one is always read-only.

From the latency-completion graphs presented in figure 1.18 and 1.19 we can clearly see that the two algorithms have very different characteristics. Looking at the banking workloads, we can see that the schedules produced by the two algorithms have the same quality, i.e. same number of schedules and same progress towards completion, however the basic algorithm produces them about twice as fast as the advanced one. On the other hand the advanced algorithm has the advantage of producing much better schedules than the basic one in high contention workloads like the hashmap. This shows that scheduling can have a huge impact on performance and that it is very important to tailor the scheduling algorithm to the target application.

5 Implementation considerations

While working on our prototypes we made some observations that we thought could be useful to someone implementing a similar system. We present them in this section.

In earlier versions we noticed that a noticeable part of the time was spent on vector operations like alloc, append and push. This required us to alter the architecture to avoid unnecessarily moving data around and to reuse allocated vectors whenever possible.

In some applications, we noticed that scheduling a batch in smaller chunks was faster than doing it all at once. For example for the banking application, scheduling 1 batch of size 65536 took between 1800 and 2000 µs (with the basic algorithm) but scheduling 8 chunks of size 8192 took only 1500 µs. This was not true for the hashmap where the latency increased slightly instead. This indicates that, even without adding more schedulers, it could be possible to improve performance by giving multiple smaller chunks for schedulers to work on.

We used $rayon$ 1.7.0 and $tokio$ 1.1.0 for some of our earlier prototypes but didn't use them in the final versions for a few reasons. The main one is that they made the recorded flamegraphs more convoluted because of all the added library calls. This made it more difficult to find bottlenecks in the systems and to verify the effectiveness of optimisations. We also found that using $rayon$ made it more difficult to avoid memory allocations as its parallel iterators required us to collect into new vectors every time. Finally we found that $rayon$ added non-negligible latency when used for scheduling and execution. This is most likely because the inputs are not large enough for the benefits to overcome the overhead of using parallel iterators repeatedly.

6 Related work

This project is related to the theory of transaction chopping used in distributed databases and transaction processing systems. The purpose of transaction chopping is to split transactions as finely as possible while ensuring that any interleaving of the resulting pieces remains serializable. This makes it possible to execute them in parallel using lower isolation levels and still guarantee transaction atomicity. The core idea is to statically analyse an application's transactions to create an SC-graph in which cycles represent the possibility for unserializable schedule. By splitting transactions appropriately and taking application semantic into account, it is possible to remove these cycles and guarantee serializable execution. In some cases the finest decomposition of a transaction is to keep it as a single piece, reducing parallelism opportunities. Many systems mitigate this issue by allowing some types of SC-cycles and using runtime techniques to enforce serializability during execution. Transaction chopping has been used to build both deterministic [8, 9] and non-deterministic systems [10–12] in both distributed and concurrent settings.

Other approaches which relates to this project include assertional concurrency control [13] in which transaction semantic is used to relax the serializability requirements and QueCC [14] which uses two planning phases to achieve control-free deterministic execution and provide high throughputs.

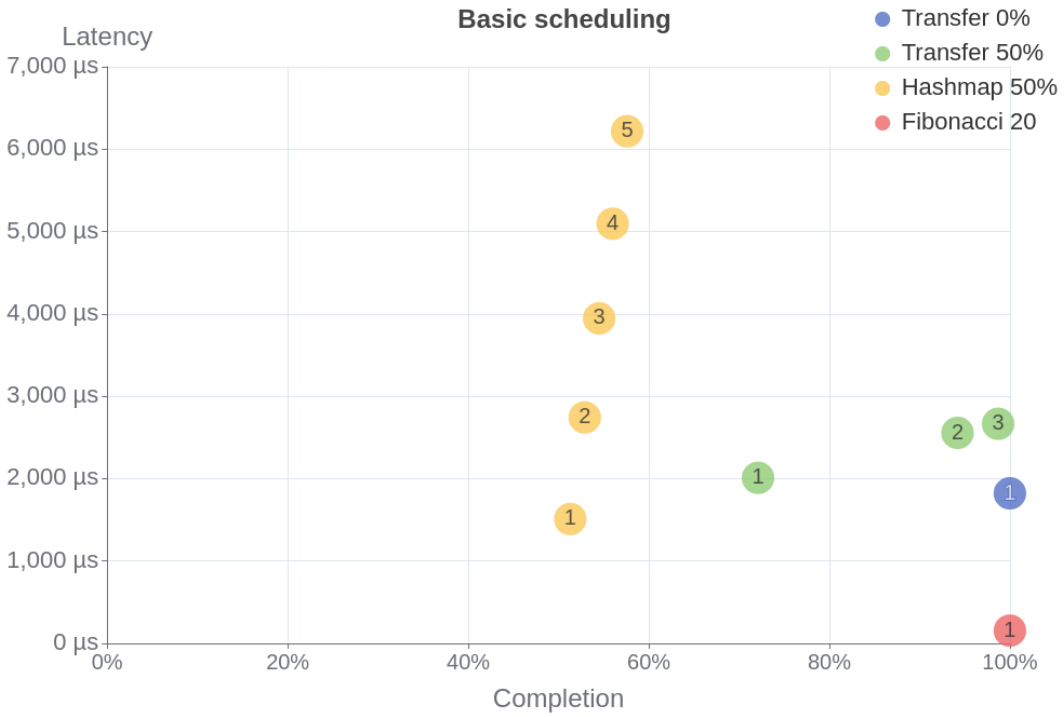


FIGURE 1.18
Performance of basic scheduling with 1 scheduler (go to text)

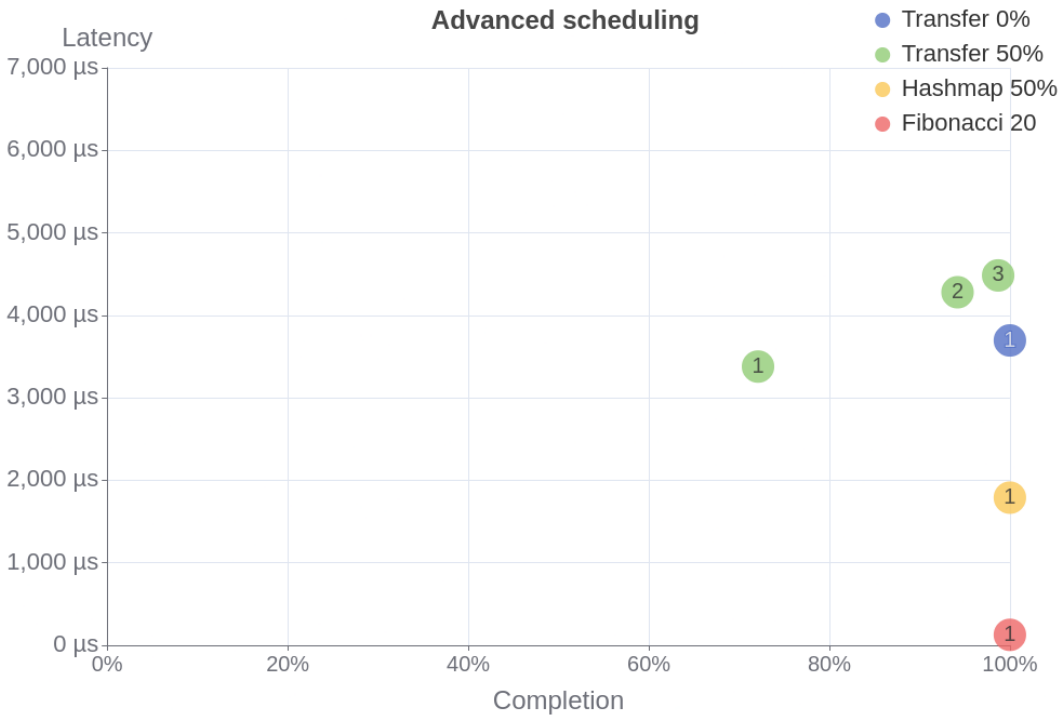


FIGURE 1.19
Performance of advanced scheduling with 1 scheduler (go to text)

Finally our project relates to open nested transactions [3, 5, 6], a variant of (closed) nested transactions where subtransactions can commit before their parent, and multi-level transactions [4], a special case of (open) nested transactions where subtransactions operate at a lower level of abstraction from their parent.

7 Conclusion

In this report we presented a new approach for parallel transaction processing based on micro-transactions. A micro-transaction is a restricted form of transaction in which memory accesses are known before execution. Using micro-transactions as building blocks, it is possible to represent arbitrary transactions as a directed graph. Just like basic blocks facilitate compiler optimizations, micro-transactions can be used to improve the performance of concurrency control implementations and make them more resilient to contention. Based on the observation that enforcing ACID properties for general transactions often comes with a performance penalty, we proposed to relax those guarantees and only enforce them for micro-transactions, leaving transaction-level synchronization to the application. To validate this approach, we implemented two scheduling-based smart contract virtual machines, each with a different scheduling algorithm, and evaluated them against a sequential baseline. We found that the prototype with the simplest algorithm was the most versatile of the two, consistently beating the sequential baseline in all applications. However, it provided poor scaling because of high synchronization overhead and was very sensitive to contention. In contrast, the prototype using the advanced scheduling was quite resilient to contention and scaled well on applications with concentrated memory accesses. On the other hand, it performed extremely poorly on applications with random memory accesses like a cryptocurrency due to the higher latency of the more complex scheduling algorithm. On an ideal synthetic workload both prototypes achieved close to a linear speedup in performance with up to 16 cores. Future work includes improving load balancing to be more resilient to heterogeneous workloads such as in those produced by mixed-application systems, reducing coordination overhead between schedulers and executors and improving scheduling so that it can scale to a higher number of memory accesses.

ACKNOWLEDGEMENTS

I would like to thank my supervisors, Prof. Rachid Guerraoui and Gauthier Voron for giving me the opportunity to work on this very interesting subject and supervising my thesis. I would especially like to thank Gauthier for his patience and support throughout the project which helped me stay motivated despite not progressing as fast as I had hoped.

BIBLIOGRAPHY

[1] Martina Camaioni et al. *Chop Chop: Byzantine Atomic Broadcast to the Network Limit*. 2023. arXiv: 2304.07081 [cs.DC] (cit. on p. 4).

[2] Anatoly Yakovenko. *Sealevel — Parallel Processing Thousands of Smart Contracts*. 2019. URL: https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192 (visited on 28th June 2023) (cit. on p. 4).

[3] Alejandro Buchmann. 'Open Nested Transaction Models'. In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 1978–1981. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_717. URL: https://doi.org/10.1007/978-0-387-39940-9_717 (cit. on pp. 7, 24).

[4] Gerhard Weikum. 'Multilevel Transactions and Object-Model Transactions'. In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 1792–1797. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_728. URL: https://doi.org/10.1007/978-0-387-39940-9_728 (cit. on pp. 7, 24).

[5] Gerhard Weikum and Hans-J. Schek. 'Concepts and Applications of Multilevel Transactions and Open Nested Transactions'. In: *Database Transaction Models for Advanced Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, pp. 515–553. ISBN: 1558602143 (cit. on pp. 7, 24).

[6] Jörg Kienzle. 'Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming'. In: (2001). URL: http://infoscience.epfl.ch/record/54686 (cit. on pp. 7, 24).

[7] Yanick Paulo-Amaro. *Master Project: Microtransactional Blockchain Virtual Machine*. URL: https://github.com/yanickpauloamaro/MasterProject (visited on 30th June 2023) (cit. on p. 13).

[8] Shuai Mu et al. 'Extracting More Concurrency from Distributed Transactions'. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 479–494. ISBN: 9781931971164 (cit. on p. 22).

[9] Alexander Thomson and Daniel J. Abadi. 'The Case for Determinism in Database Systems'. In: *Proc. VLDB Endow.* 3.1–2 (Sept. 2010), pp. 70–80. ISSN: 2150-8097. DOI: 10.14778/1920841.1920855. URL: https://doi.org/10.14778/1920841.1920855 (cit. on p. 22).

[10] Yang Zhang et al. 'Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems'. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 276–291. ISBN: 9781450323888. DOI: 10.1145/2517349.2522729. URL: https://doi.org/10.1145/2517349.2522729 (cit. on p. 22).

[11] Zhaoguo Wang et al. 'Scaling Multicore Databases via Constrained Parallel Execution'. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1643–1658. ISBN:

9781450335317. DOI: 10.1145/2882903.2882934. URL: https://doi.org/10.1145/2882903.2882934 (cit. on p. 22).

[12] Dennis Shasha, Eric Simon and Patrick Valduriez. 'Simple Rational Guidance for Chopping up Transactions'. In: *SIGMOD Rec.* 21.2 (June 1992), pp. 298–307. ISSN: 0163-5808. DOI: 10.1145/141484.130328. URL: https://doi.org/10.1145/141484.130328 (cit. on p. 22).

[13] Arthur Bernstein and Philip Lewis. 'Transaction Decomposition Using Transaction Semantics'. In: *Distributed and parallel databases* 4 (June 1998). DOI: 10.1007/BF00122147 (cit. on p. 22).

[14] Thamir Qadah and Mohammad Sadoghi. 'QueCC: A Queue-oriented, Control-free Concurrency Architecture'. In: Dec. 2018. DOI: 10.1145/3274808.3274810 (cit. on p. 22).