

Research Report

Highly Parallel Reconstruction of Wallet History in Cardano Blockchain

Alex Sierkov
alex.sierkov@gmail.com

January 12, 2023

Contents

Introduction	2
Method	3
2.1 Design objectives and major components	3
2.2 Overview of the wallet-history reconstruction process	3
2.3 Overview of the indexing process	4
2.4 Structure of on-disk indices	5
2.5 Efficient references to blockchain elements	6
2.6 Fully-parallel hybrid sort algorithm	7
2.7 Optimized parallel scheduler	8
2.8 LZ4 compression of blockchain data	8
Results	9
3.1 Data processing throughput	9
3.2 Parallel efficiency	10
3.3 Wallet-history-reconstruction time	10
3.4 Effect of LZ4 compression on performance	10
Discussion	11
4.1 Can this method support all types of Cardano addresses?	11
4.2 Can blockchain data be validated quicker?	12
4.3 A technical vision for integrating this method into Daedalus	13
Conclusion	16
References	17
Appendices	18
A.1 Setup of the benchmarking environment	18

Introduction

As of January 12, 2023, Cardano [1] is the number eight blockchain in the world based on its market capitalization [2]. It operates millions of wallets with transaction volumes reaching billions of U.S. dollars daily [3]. Cardano sees an every-day user as the typical consumer of its services, so ease-of-use and convenience are of primary importance.

However, there is a **problem** that slows down its adoption: its primary fully-decentralized wallet, Daedalus [4], has a deficiency that goes against the objectives of ease-of-use and convenience: it is very slow to synchronize:

- after an installation, it takes a day to be ready for use;
- after a few months of not being used, it can take more than an hour to process the updates.

As a consequence, many users are forced to use "light" wallets, online wallets operated by centralized organizations. That reduces their security and breaks the ideal of full decentralization, a method for providing services that doesn't force users to depend on a specific organization for a specific service.

The research question of this report is: **can the synchronization performance be drastically improved?** If the answer is yes, that can have a major impact on the adoption of Cardano.

The process of synchronization has three parts: network delivery of blockchain data, data validation, and computation of derived data. Since there are existing solutions for a quicker delivery of validated blockchain data, such as Mithril [5], this report focuses on optimizing the third component: wallet-history reconstruction from pre-validated blockchain data.

The **contributions** of this report are:

- A highly-parallel method for reconstruction of wallet history, achieving **210 times quicker reconstruction time** than the status quo implementation of **Cardano Wallet** [6] when supported by sufficiently powerful hardware.
- A parallel-sort algorithm optimized for cryptographic hashes.
- Evidence that LZ4 compression can halve the on-disk size of blockchain data without noticeably affecting reconstruction performance.
- A vision for the parallel computation of per-epoch stake distribution, the key component of Cardano's consensus mechanism.
- A vision for the technical architecture integrating the proposed method into official Daedalus builds.

Method

2.1 Design objectives and major components

At the end of 2022, the size of the Cardano blockchain is 92GB and to reconstruct a wallet’s transaction history all the data has to be read at least once. Thus, the available storage bandwidth is a likely bottleneck, and the design objective of the proposed method is to make the best possible use of it.

Given the design objective, the performance of the proposed method is analyzed as the effective blockchain read throughput measured in **megabytes per second**. This evaluation approach makes it easy to compare its performance with the sequential read speed of local storage hardware and to determine if storage bandwidth is in-fact the bottleneck.

To achieve the objective, the following tactics are repeatedly used:

- leverage all available CPU cores as well as possible;
- process data sequentially to make efficient use of CPU caches and prefetching [7];
- keep data structures small for efficient utilization I/O bandwidth and CPU caches [7].

In addition, the implementation is done in C++ language, which provides direct control over in-memory layout of data structures and supports high-performance simultaneous multi-threading.

Furthermore, to make the transfer of improvements into Deadalus (written in JavaScript and Haskell) simpler, the use of third-party libraries is minimized and easier-to-implement algorithms are chosen when that does not noticeably affect performance.

The major components of the proposed method that contribute to its performance are:

- a space-and-access-efficient format of references to items stored on the blockchain;
- a compact structure of on-disk indices;
- a parallel sort algorithm leveraging the uniform distribution of cryptographic hashes;
- a parallel task scheduler optimized for multiple heterogeneous tasks.

2.2 Overview of the wallet-history reconstruction process

The wallet-history reconstruction process is organized into the following steps:

- Create three on-disk indices (described in section 2.4) that make the reconstruction of the history of an individual wallet quick.

- Use **address_use** index to find transaction outputs referencing a given wallet;
- Use **transaction_output_use** index to identify spent transaction outputs.
- Use **block_metadata** index to find time and other metadata for each transaction.

There are many benefits to this approach:

- It allows to process the raw blockchain data only once.
- It reduces the amount of RAM needed for the processing.
- It allows to quickly reconstruct the history of any stake-key-identifiable wallet.
- It allows the indices to be shared via trustful mechanisms, such as Mithril [5], to further accelerate the bootstrapping time of Daedalus after a fresh install.

The downside of this approach is that it requires up to 10% more of storage space.

2.3 Overview of the indexing process

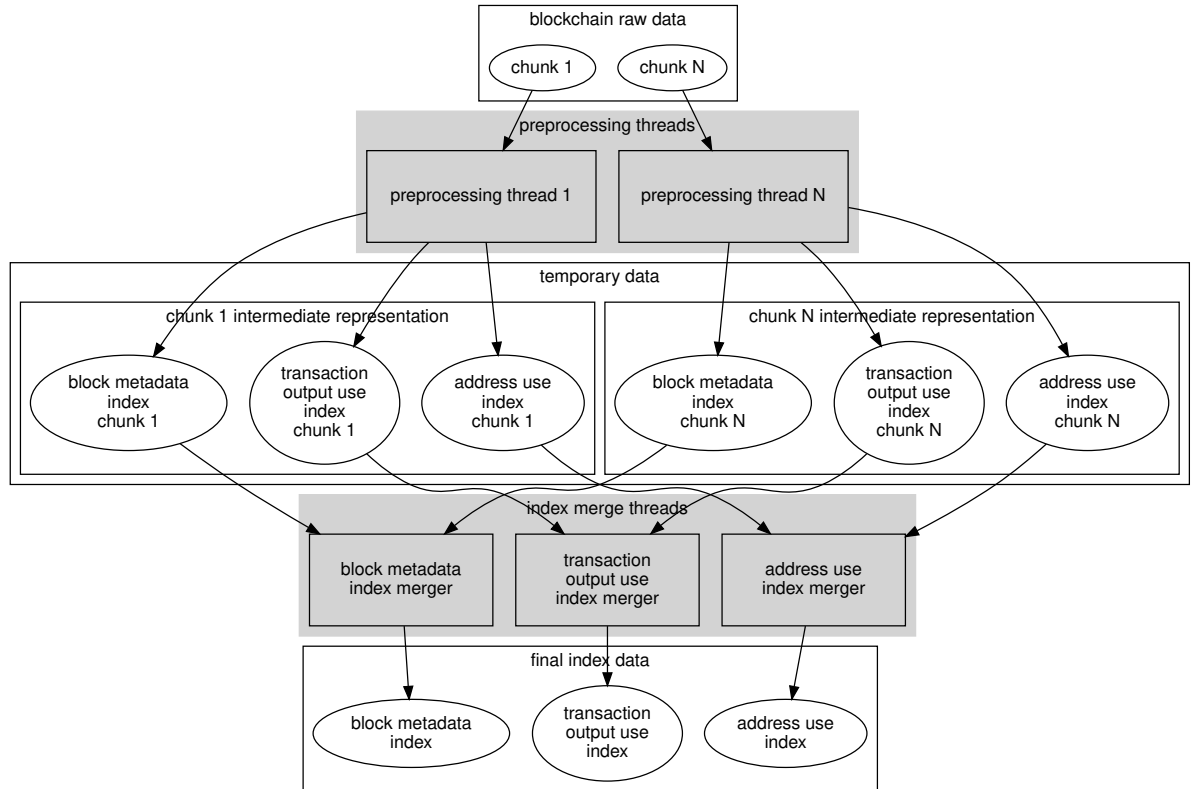


Figure 2.1: A high-level overview of the indexing process

Figure 2.1 presents a high-level overview of the indexing process. The indexing process generates three on-disk indices described in section 2.4 and consists of the following steps:

- Raw blockchain data in the format of **ImmutableDB** produced by Cardano Node [8] is scanned for available blockchain chunks with each chunk's data mapped into the unified

address space (described in section 2.5) for easy referencing.

- Parallel-processing tasks for each chunk are scheduled, generating per-chunk indices (described in section 2.4).
- Index merge threads (described in section 2.6) combine the per-chunk indices into final indices.

2.4 Structure of on-disk indices

To determine the minimum set of indices, the information provided by Daedalus about transactions was analyzed. Table 2.1 presents the list of data items shown in the Daedalus’s UI to users along with the respective information source.

UI data item	Source of the information
transaction timestamp	block metadata of the block containing the transaction
transaction type (incoming/outgoing)	transaction metadata and the dictionary of spent transaction outputs
transaction amount	transaction metadata
transaction status	irrelevant for wallet-history reconstruction
from addresses	transaction metadata
from rewards	transaction metadata
to addresses	transaction metadata
transaction fee	transaction metadata
certificate deposits	transaction metadata
transaction id	computed directly from the transaction metadata
cardano explorer link	derived from the transaction id, so computed from the transaction metadata

Table 2.1: Transaction-related information that Daedalus shows to users in its UI

Based on the above, one can establish that three indices are sufficient:

- **address_use** index, an index to find transactions related to a given wallet;
- **transaction_output_use** index, an index to check if a given transaction output has been spent;
- **block_metadata** index, an index to find block metadata by a transaction reference.

Figure 2.2 provides C-style definitions of index entries for all indices. Each index file is a simple sorted array of fixed-size elements, allowing to find an element in

- $\log(N)$ lookups using a simple binary search;
- $\log(\log(N))$ lookups when using precomputed data about the location of individual key ranges in the index.

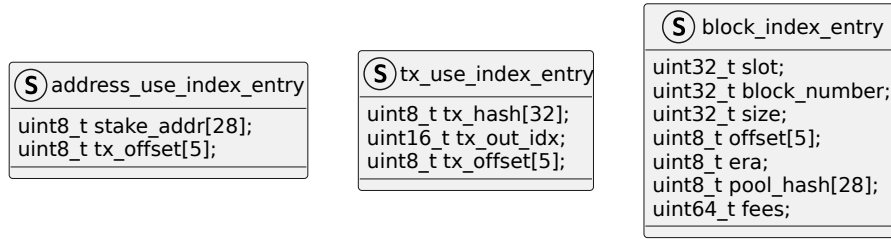


Figure 2.2: On-disk data structure of items in each index

2.5 Efficient references to blockchain elements

In the process of wallet-history reconstruction one must identify each transaction related to a given wallet. Such identification implies that there is an efficient way to access the metadata of each transaction. At the same time, the transaction identifiers used by the Cardano blockchain (Blake2b-256 hashes of the transaction data) do not directly provide for such a way. So, there is a need for a way to reference transactions with the following properties:

- be compact to save the storage space and reduce the required RAM;
- have fixed size to allow for quick on-disk and in-memory access;
- have $O(1)$ access complexity to the referenced data.

Figure 2.3 depicts a simple mapping of blockchain chunk data from the **ImmutableDB**. Any byte offset and chunk number can be uniquely mapped into a single unsigned integer. The reason for that is that new entries are added to the blockchain always at the end. Thus, the offsets of historical items are immutable, allowing for a simple and efficient mapping.

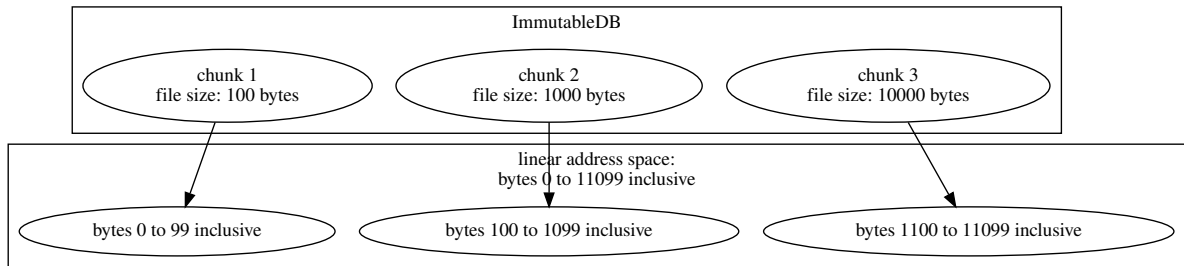


Figure 2.3: A mapping of ImmutableDB's chunks and their byte offsets into a single address space

So, an **efficient reference** is simply a byte offset into this unified address space of raw blockchain data, which can be inversely mapped to a chunk's on-disk filename and a chunk's byte offset. Such reference has all the necessary properties:

- it requires just 5 bytes to reference any transaction while supporting blockchain sizes of up to 1.1TB (92GB currently);
- it has a fixed size;

- it allows the transaction metadata to be read directly from disk with $O(1)$ complexity.

2.6 Fully-parallel hybrid sort algorithm

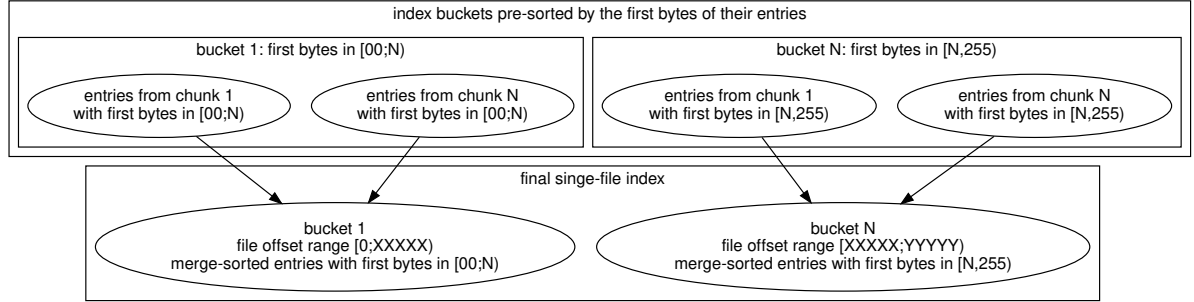


Figure 2.4: First-byte pre-sorting with within-bucket merge sort

Figure 2.4 depicts the hybrid sort algorithm, which does the following:

- Each chunk-processing task outputs pre-sorted values for each index into separate files.
- The index chunks are additionally split into N **buckets** based on the value of the first byte of each index entry. This step allows to quickly determine the final size of each bucket and to combine the buckets without comparing their sub-entries.
- Each bucket is allocated a fixed byte range in the final index file based on the size of each bucket and the relative order of the first bytes of entries contained in that bucket.
- For each bucket a merge-sort thread is started, merging the data from per-chunk files of that bucket into the pre-allocated offset range within the final index file.

In comparison to the **parallel merge sort** [9]:

- This method uses all available CPU threads during the final merge stage while a k -way-merge sort can use only one thread to ensure the correct order of elements in the output.

In comparison to the **parallel radix sort** [10]:

- This method does not require an initial bucket-splitting pass ensuring a fair distribution of items across buckets. The reason for that is that the data items in the two larger indices: **address_use** and **transaction_output_use** start with a cryptographic hash, which guarantees a uniform distribution of their bytes, and thus their first bytes as well.
- This method makes only two passes over the data in contrast to *key_length/radix_size* passes required by a radix sort. With the chosen radix size of one byte and the respective entry sizes of **address_use** and **transaction_output_use** indices of 33 and 39 bytes, that would mean **33** and **39** passes over the data respectively. Sure, that can be optimized by increasing the radix size, but still a two-pass radix sort is impossible because of its excessive memory requirements.
- This method works well only with data sets in which the first bytes of each entry are uniformly distributed.

2.7 Optimized parallel scheduler

A straightforward implementation of the indexing process would sequentially launch four parallel tasks:

- Pre-process the raw blockchain data.
- Sort the **address_use** index.
- Sort the **transaction_output_use** index.
- Sort the **block_metadata** index.

However, that would leave some performance on the table since:

- the data distribution among the worker threads can be not precisely uniform;
- the execution performance of different worker threads may be unequal;
- most worker threads wait without tasks at the end of each stage when only a few tasks are left.

To minimize the impact of the above issues, the optimized parallel scheduler:

- runs multiple heterogeneous task sets using the same worker threads;
- supports configurable priorities for each parallel task;
- schedules new tasks in an event-driven way to minimize the idling of worker threads.

2.8 LZ4 compression of blockchain data

Even though the extra storage required for the indices is just 10% of the total storage needed for a Daedalus installation, this increase can restrict the applicability of the method. To see if it's possible to make the proposed method fit into the original storage requirements of Daedalus, a compression of blockchain data with **LZ4** compression method [11] was explored. The compression was applied on the chunk level.

Results

To conduct the experiments a 24-core/48-thread server was rented with two NVMe SSDs organized into a RAID0 storage array. The setup and hardware are described in appendix A.1. Each experiment was repeated five times, and the average value across the five runs is reported. In addition, before each run, the respective Docker containers were restarted and the host’s filesystem and block-device caches were flushed with the following bash command:

```
sync && echo 3 > /proc/sys/vm/drop_caches
```

3.1 Data processing throughput

To evaluate the scalability of the proposed method with regards to the number of worker threads, data-processing throughput of the end-to-end index-creation process was measured. The results are presented in figure 3.5. The red line presents an estimate of the single-threaded sequential-read speed of the local storage evaluated using **hdparm -t** command. The proposed method scales linearly until the storage bandwidth is exhausted.

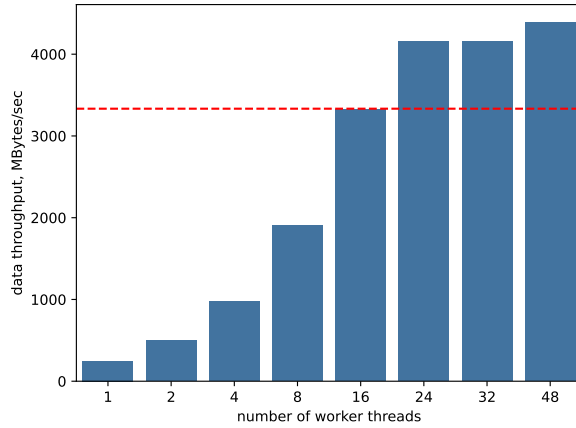


Figure 3.5: Data-processing throughput

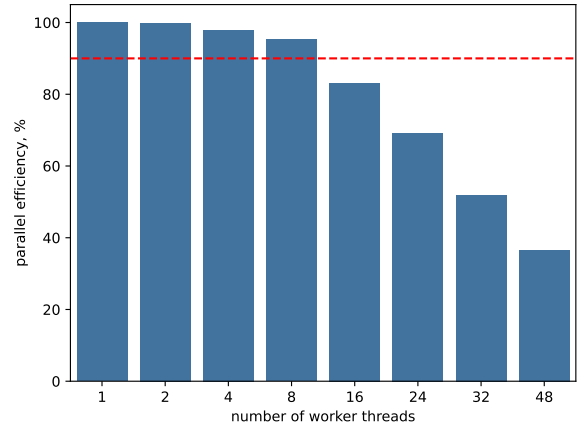


Figure 3.6: Parallel efficiency

3.2 Parallel efficiency

To understand the efficiency of the scaling, the parallel efficiency coefficient for each thread count was computed. The parallel efficiency coefficient is the throughput of each run divided by the linear scaling expectation for that thread count, which is the throughput for one thread multiplied by the number of active threads in that run. The results are depicted in figure 3.6. The parallel efficiency stays above 90% until the storage bandwidth is exhausted as described in the previous section.

3.3 Wallet-history-reconstruction time

To measure the wallet-history-reconstruction time, the time it takes **Cardano Wallet** to reconstruct the wallet history of one wallet when provided with fully validated blockchain data was compared versus the time it takes the proposed method to create the indices for wallet-history reconstruction of any wallet. Figure 3.7 presents the results. Leveraging all 48 threads, **the proposed method is 210 times faster**.

3.4 Effect of LZ4 compression on performance

Since the proposed method increases the necessary storage space, it was interesting to see the effect of a fast compression algorithm on the data-processing throughput. Figure 3.8 presents the results. LZ4 algorithm achieved a compression ratio of **2.04** while keeping the processing speed roughly the same at low thread counts and even slightly improving it at higher thread counts. Using LZ4 compression improved the throughput when the local storage bandwidth was exhausted.

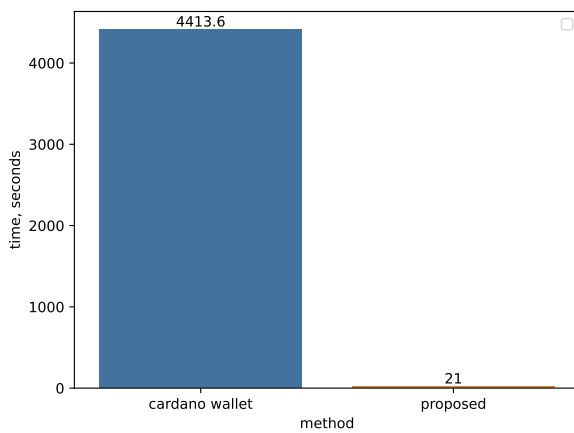


Figure 3.7: Wallet-history reconstruction time

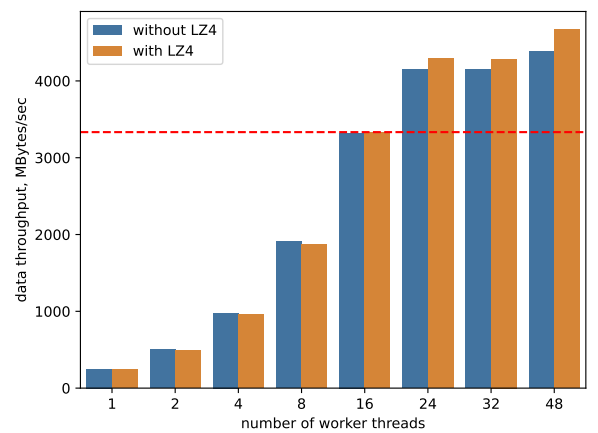


Figure 3.8: Effect of LZ4 compression

Discussion

4.1 Can this method support all types of Cardano addresses?

The proposed method finds transactions with addresses directly referencing a given stake key. Even though Daedalus forces users to reference the same stake key in all addresses, the Cardano specifications [12] allow for other options as well. Here is how the proposed method can handle the alternative cases:

- **Multiple stake keys.** An owner of a wallet uses multiple stake-key addresses during the wallet's lifespan. For this case, the solution is simple: one needs to do `N address.use` index lookups instead of just one. For personal wallets the `N` will be in low single digits, and, thus, the algorithmic complexity is practically the same.
- **Correct stake and unrelated payment keys.** A sender of funds composes a destination address from unrelated payment and stake keys. In this case, an index search by the stake key will find additional transactions. Thus, an explicit verification is needed that the matching payment addresses belong to the wallet. Since, the matching transactions are a tiny subset of all transactions, this can be done very quickly.
- **Correct payment and unrelated stake keys.** A sender of funds composes a destination address from a payment address belonging to the wallet and an unrelated stake address. This case can be solved by indexing the blockchain by payment-key hashes instead of stake-key hashes. While fully feasible, it can lead to a performance reduction in `N` times. The reason for that is that a typical personal wallet in Daedalus will use more than one payment key during the wallet's lifetime. Thus, the average number of index lookups is likely to be larger than one. Still, even in the case when `N` is approaching hundreds, the reconstruction performance will be in the range of single-digit to low-double-digit seconds. So, that doesn't look like a blocker to the adoption of the proposed method.
- **No stake key.** A sender of funds composes the destination address from a payment key only. In this case, the indexing of the blockchain must be done by payment-key hashes. This change comes with exactly the same caveats that were discussed in the previous case.

To summarize, the proposed method can be applied for all types of Cardano addresses. In some cases, the reconstruction performance can be slower by a constant but still be much faster than that of Cardano Wallet. Given that Daedalus forces users to use addresses referencing the same stake key, indexing by a stake key provides the best performance without any practical sacrifices.

4.2 Can blockchain data be validated quicker?

The proposed method assumes that the blockchain data have already been pre-validated. So, to speak of an improvement in the overall synchronization performance, it's important to take a closer look at the available options for getting the pre-validated blockchain data. They are:

- Receive the pre-validated data from a trustworthy source.
- Receive the data from an untrustworthy source and leverage a faster method for local validation.

4.2.1 A trustworthy source of pre-validated blockchain data

Even though this option may sound unrealistic, the Cardano ecosystem is already actively developing a real solution. The project is named **Mithril**, and it's an implementation of a multi-signature protocol designed to deliver verifiable snapshots of blockchain data and ledger state. More information can be found in the Mithril paper [5] and in the official GitHub repository of the project [13].

Once Mithril is production ready, the scheme for an accelerated end-to-end wallet synchronization can look as simple as get the data through Mithril and apply the wallet-history reconstruction method presented in this report.

4.2.2 A faster method for local validation of blockchain data

This research report presents a highly parallel method for reconstruction of wallet history, so, it's natural to do a quick analysis of the acceleration potential if parallel-processing methods are applied to the validation task. The results of such analysis are presented in table 4.2. It shows that one particular computation, **per-epoch stake distribution**, influences many validation steps and is likely the most expensive to compute. So, if one can find a faster way to compute it, then, other validation steps can likely be accelerated at least by the same factor. Subsection 4.2.3 presents an approach for a faster computation of per-epoch stake distribution.

4.2.3 Semi-parallel computation of per-epoch stake distribution

Figure 4.9 presents an overview of the computation. It does the following:

- Group blockchain chunks by epoch.
- Start a pool of worker threads, with each thread processing individual epochs one-by-one.
- Generate intermediate indices for each epoch. The respective data structures are presented in figure 4.10.
- Sort globally and perform a join of **transaction_output_use** and **transaction_amount** indices.
- Process in parallel the partitions of **tx_in**, **tx_out**, and **deleg** indices for each epoch. The output of each epoch are the cumulative numerical difference of each account account updated during that epoch.

Validation step	Method of parallelization
transaction witnesses	parallel loop by each witness
block body hashes	parallel loop by each block
block body signature	parallel loop by each block; a post-processing step to verify the rights for signing
operational certificate validity	parallel loop by each pool; sequentially process processes certificate updates for each pool
transaction invariants	parallel loop by each transaction
leader election	parallelizable within an epoch; across epochs must be done sequentially; depends on the computation of per-epoch stake distribution
staking rewards	parallel loop by each stake key; depends on the computation of per-epoch stake distribution
reward withdrawals	parallel loop by each stake key; depends on the computation of staking rewards
protocol parameter updates	must be processed sequentially; a very rare and computationally simple task; not a bottleneck

Table 4.2: Typical validation steps performed by the nodes of the Cardano network

- Process the per-epoch account change sets sequentially to create the final stake distributions for each epoch.

The above process has only two sequential steps:

- The global join, which operates at the sequential read speed of the local storage hardware.
- The final "apply per-epoch diffs" step, which processes aggregated data that is pre-organized for efficient sequential processing.

Based on early experiments, this method can prepare stake distributions for all epochs in mere minutes.

4.3 A technical vision for integrating this method into Daedalus

As this report shows, the potential for a drastic improvement in synchronization performance is real. To further support that, figure 4.11 depicts a high-level technical architecture for integrating the presented method into Daedalus. It's key components are:

- **Fetcher** - a technical component responsible for downloading unvalidated blockchain data over network.
- **Validator** - a technical component responsible for validating the downloaded blockchain data locally.
- **Indexer** - a technical component responsible for the creation of intermediary data.

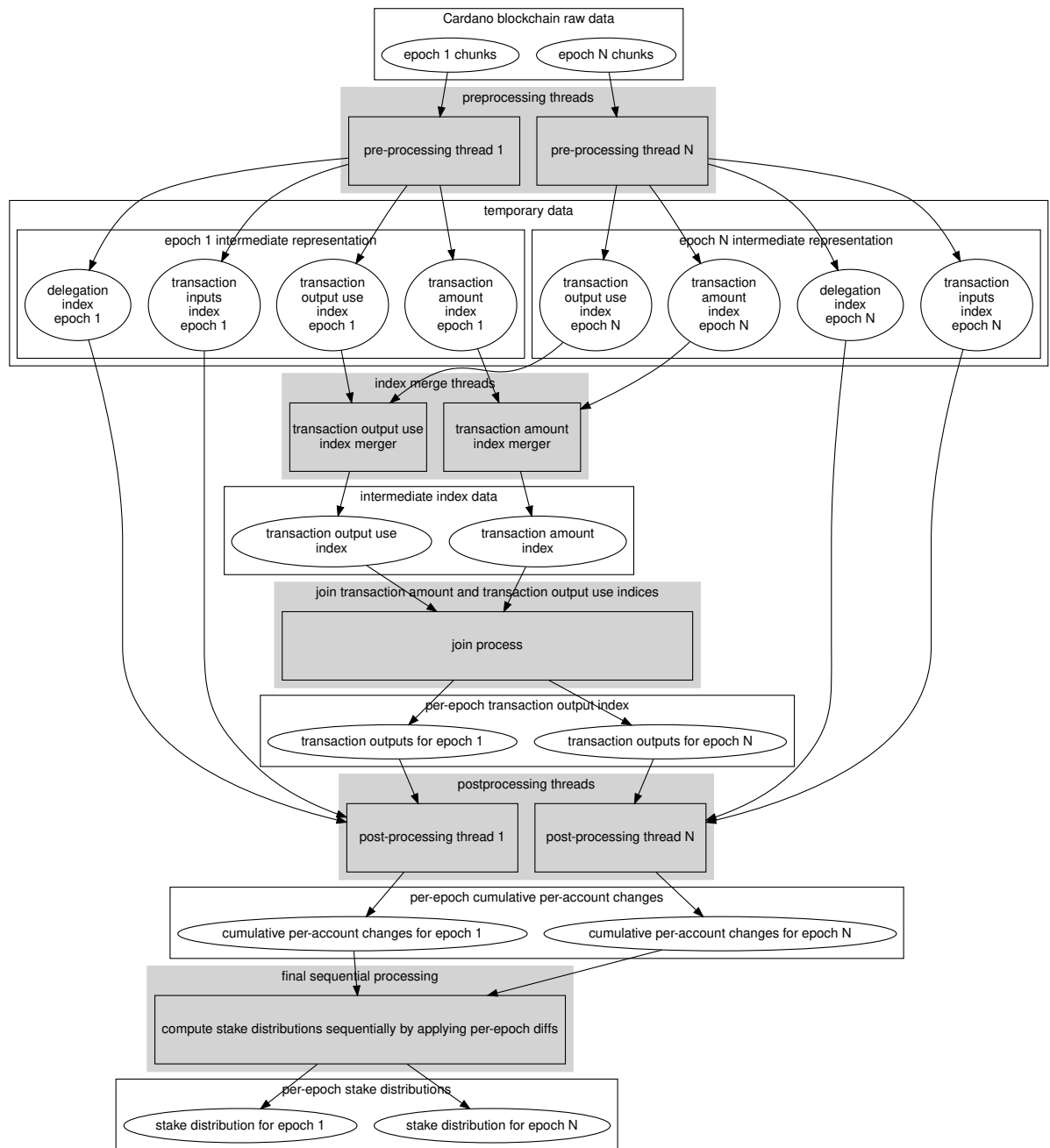


Figure 4.9: An overview of the semi-parallel way to compute per-epoch stake distribution

- **API** - a technical component that combines the Fetcher, the Validator, and the Indexer to provide a list of high-level operations with user wallets. This component provides exactly the same HTTP API as that of **Cardano Wallet**.
- **Mode Switcher** - a technical component that allows users to dynamically switch the underlying mechanism for executing wallet operations between the accelerated one, **Daedalus Turbo**, or the status quo one, **Cardano Wallet**.

(S) tx_in	(S) tx_out	(S) tx_out_amount	(S) tx_out_use	(S) deleg
uint16_t epoch; uint8_t stake_addr[28]; uint64_t amount;	uint16_t epoch; uint8_t stake_addr[28]; uint64_t amount;	uint8_t tx_hash[32]; uint16_t tx_out_idx; uint64_t amount;	uint8_t tx_hash[32]; uint16_t tx_out_idx; uint16_t epoch;	uint16_t epoch; uint8_t stake_addr[28]; uint8_t pool_hash[28];

Figure 4.10: C-style definitions of on-disk data structures used to compute per-epoch stake distribution

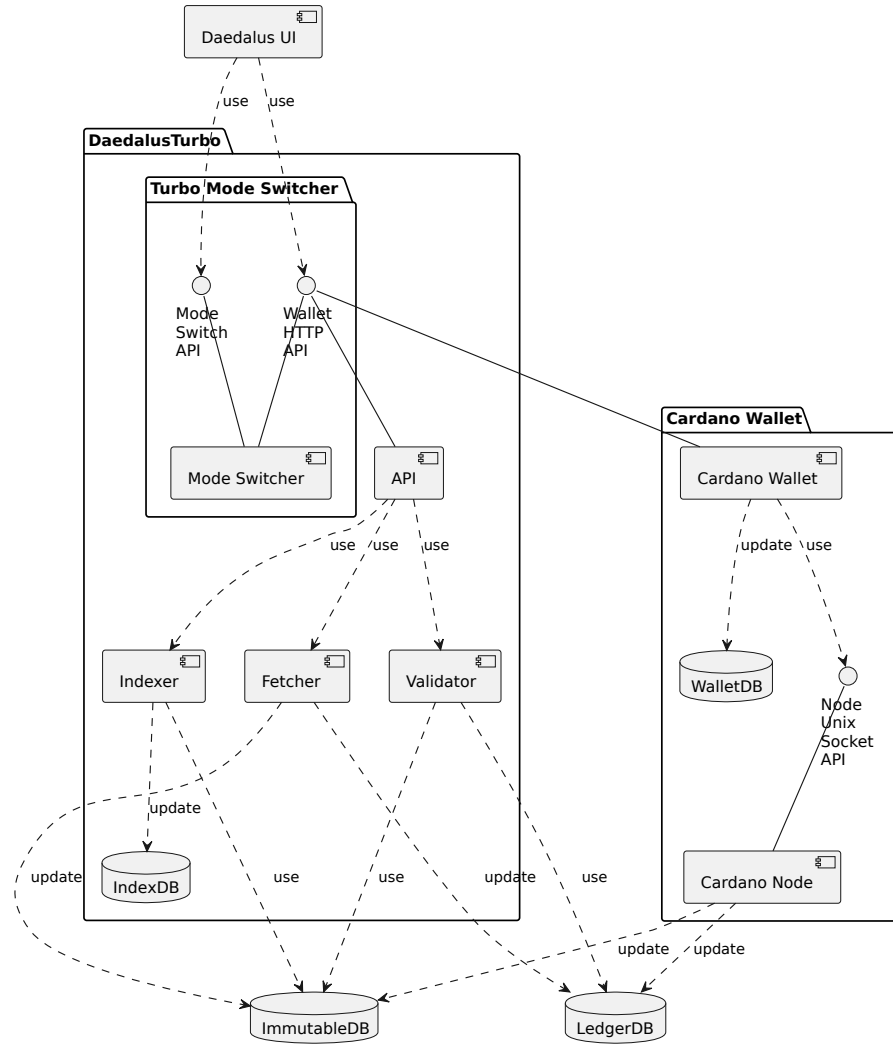


Figure 4.11: A potential technical architecture integrating the proposed method into Daedalus

The benefit of the presented technical architecture is that it allows for a gradual introduction of the accelerated operations as they are being developed and published. At the same time, users can switch between the accelerated method and the traditional method at any time, providing the additional security and flexibility necessary for a gradual rollout of a big change.

Conclusion

The evidence provided in this report shows that **the synchronization performance of Daedalus can be drastically improved**. The core of this report presents a highly-parallel method for wallet-history reconstruction. This method can reconstruct a wallet's history **210 times quicker than Cardano Wallet** when supported by sufficiently powerful hardware. Moreover, it does so in a much more scalable way: the prepared data allow reconstructing the history of any wallet identifiable by a stake key not just a single one.

Last but not least, this report discusses how the pre-validated blockchain data, this method's prerequisite, can be prepared quicker and presents a high-level technical architecture for the integration of it into official Daedalus builds.

References

- [1] *Cardano Blockchain*. <https://cardano.org/>.
- [2] *Today's Cryptocurrency Prices by Market Cap*. <https://coinmarketcap.com/>.
- [3] *Messari.io: Cardano Adjusted Transaction Volume*. <https://messari.io/asset/cardano/chart/txn-tsfr-val-adj>.
- [4] *Daedalus Wallet*. <https://daedaluswallet.io/>.
- [5] Pyrros Chaidos and Aggelos Kiayias. "Mithril: Stake-based Threshold Multisignatures." In: *IACR Cryptol. ePrint Arch.* (2021), p. 916. URL: <https://eprint.iacr.org/2021/916>.
- [6] *Cardano Wallet GitHub*. <https://github.com/input-output-hk/cardano-wallet>.
- [7] Ulrich Drepper. "What Every Programmer Should Know About Memory." In: *LWN.net* (2007). URL: <https://www.akkadia.org/drepper/cpumemory.pdf>.
- [8] *Cardano Node GitHub*. <https://github.com/input-output-hk/cardano-node>.
- [9] Minsoo Jeon and Dongseung Kim. "Parallel Merge Sort with Load Balancing." In: *Int. J. Parallel Program.* 31.1 (2003), pp. 21–33. DOI: 10.1023/A:1021734202931. URL: <https://doi.org/10.1023/A:1021734202931>.
- [10] Shin-Jae Lee et al. "Partitioned Parallel Radix Sort." In: *J. Parallel Distributed Comput.* 62.4 (2002), pp. 656–668. DOI: 10.1006/jpdc.2001.1808. URL: <https://doi.org/10.1006/jpdc.2001.1808>.
- [11] *LZ4 - Extremely fast compression*. <https://github.com/lz4/lz4>.
- [12] *Cardano Ledger Specifications*. <https://github.com/input-output-hk/cardano-ledger>.
- [13] *Mithril - Stake-based threshold multisignatures on top of the Cardano network*. <https://github.com/input-output-hk/mithril>.
- [14] *Vultr Cloud Hosting and Services*. <https://www.vultr.com/>.
- [15] *AMD EPYC™ 7443P Processor Specifications*. <https://www.amd.com/en/products/cpu/amd-epyc-7443p>.
- [16] *Dool, a Python 3 compatible fork of Dstat*. <https://github.com/scottchiefbaker/dool>.

Appendices

A.1 Setup of the benchmarking environment

A.1.1 Hardware

For experiments, a dedicated server with the following configuration was rented by **Vultr** [14]:

- AMD EPYC 7443P processor with 24 cores and 48 threads [15];
- 256GB of RAM;
- 2 NVMe SSDs of 1.92TB each combined into a software RAID0 using Linux’s **mdadm**.

A.1.2 Data

A snapshot of Cardano blockchain and its derived data was taken at the slot number **77374448**:

- A copy of the **ImmutableDB** created by a Cardano Node instance.
- A copy of the **LedgerDB** created by a Cardano Node instance.
- A copy of the **PoolDB** created by a Cardano Wallet instance.

All data were copied to the SSD RAID0 volume to ensure the best possible storage throughput.

A.1.3 Operating System

Ubuntu Linux 22.04 LTS was used with the following packages additionally installed:

- docker-compose
- dool [16]
- node.js 18

The maximum number of allowed open files was set to 32768 using bash **ulimit** command.

A.1.4 Cardano Wallet and Cardano Node

The following Docker images provided by the IOG on the Docker Hub were used:

- inputoutput/cardano-node:1.35.3-configs
- inputoutput/cardano-wallet:2022.10.6

In addition, the following actions were performed before each experiment run:

- Internet access was blocked inside the containers to keep the blockchain unchanged.
- The host's filesystem caches were flushed;
- Wallet-specific data created by the cardano wallet were deleted;
- cardano-node and cardano-wallet containers were restarted.