CS 597
Research Report
Optimizing Proof of Work through Memoization

Adarsh Agrawal, A20517609

## Abstract

Proof of Work (PoW) is the leading technique to address the leader election in a trustless and decentralized blockchain system. PoW is considered the most energy intensive approach to securing blockchain technologies. An example of PoW in practice is Bitcoin, which has been estimated to use 107 terawatt hours in 2023, on-par with a medium size country's energy consumption. Traditional solutions, such as VISA, are able to implement a centralized trusted system in under 1 terawatt hours of energy consumption. Much of the crypto community have converged on Proof of Stake as the solution to reducing power consumption, but it comes at the cost of less decentralization reducing the security of the system in a trustless environment. This work aims to improve the power efficiency of PoW by remembering the work through memoization, a process that we call Proof of Space (PoS). The claim is that it should be more power efficient to store and recall a cryptographic hash than it would be to generate it from scratch as many times as possible in order to increase the chances of success. In PoS, we use persistent storage to store cryptographic hashes so that these hashes can be quickly retrieved on-demand. We take a minimalist approach to this problem and break down the problem into two parts: 1) hash generation and 2) sorting. In the first stage, we utilize the BLAKE3 hashing algorithm to efficiently generate a large number of hashes in parallel, maximizing processing power utilization. BLAKE3 was chosen for being lightweight with excellent performance on a wide range of devices from Raspberry Pis to many-core server systems. These hashes are stored on disk based on specific prefix values, in order to allow in-memory sort in the 2nd stage. The final collection of hashes are fully sorted, enabling fast retrieval through binary search when a cryptographic hash is required by the blockchain system. We implemented the PoS in C using the BLAKE3 C library. We evaluated the proposed work on a number of systems, such as 4-core Raspberry Pi 4, 12-core Intel Haswell desktop, a 64-core AMD Epyc server, to a 192-core Intel Skylake server, and compared it to the production Chia PoS implementation.

## Implementation:

Our proposed algorithm optimizes plot creation and proof generation through a two-stage process:

Stage 1: Parallel Hashing and Prefix-based Storage

- Harnessing Parallel Processing Power: We leverage the BLAKE3 hashing algorithm, known for its speed and security, to efficiently generate a massive number of hashes in parallel. This maximizes utilization of the available processing power during the initial, computationally intensive stage of plot creation.
- Strategic Storage with Prefixes: Instead of storing the generated hashes randomly, we employ a strategic approach. Each hash is stored based on a specific prefix value it

generates. This prefix acts like a fingerprint, allowing for efficient searching later during proof generation.

Stage 2: Targeted Search with Binary Search

- Challenge Prefix as a Search Key: When a challenge arrives from the blockchain, its prefix is extracted. This prefix serves as a crucial search key for the second stage.
- Fast and Efficient Search with Binary Search: Since the hashes are stored based on prefixes, we can leverage a binary search algorithm. This significantly reduces the search space compared to searching the entire plot. This minimization translates to faster proof generation with minimal resource consumption.

**Systems Used:**

| CPU Model | Sockets | Compute Power | RAM |
|---|---|---|---|
| ARM Cortex-A72 | 1 | 2.2 GHz | 1.8 GB |
| Intel(R) Xeon(R) Gold 5118 | 2 | 2.3 GHz | 62 GB |
| AMD EPYC 7501 | 2 | 2.3 GHZ | 330 GB |

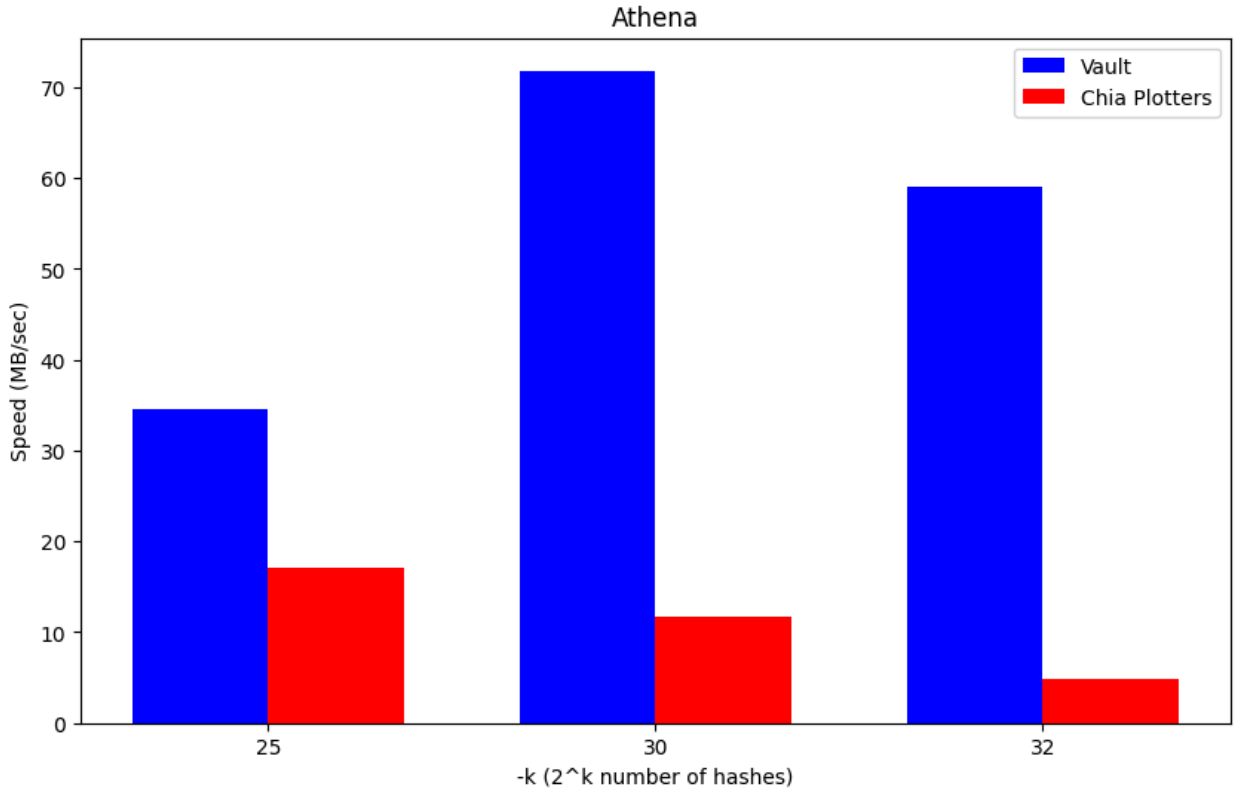
## Results:

In our analysis, we pitted our implementation against the renowned Proof of Space cryptocurrency Chia, conducting simulations across three distinct scenarios: k values of 25, 30, and 32, respectively. These simulations were executed on three diverse systems. Remarkably, our implementation consistently surpassed Chia across all scenarios and on all machines, showcasing superior performance across the board.

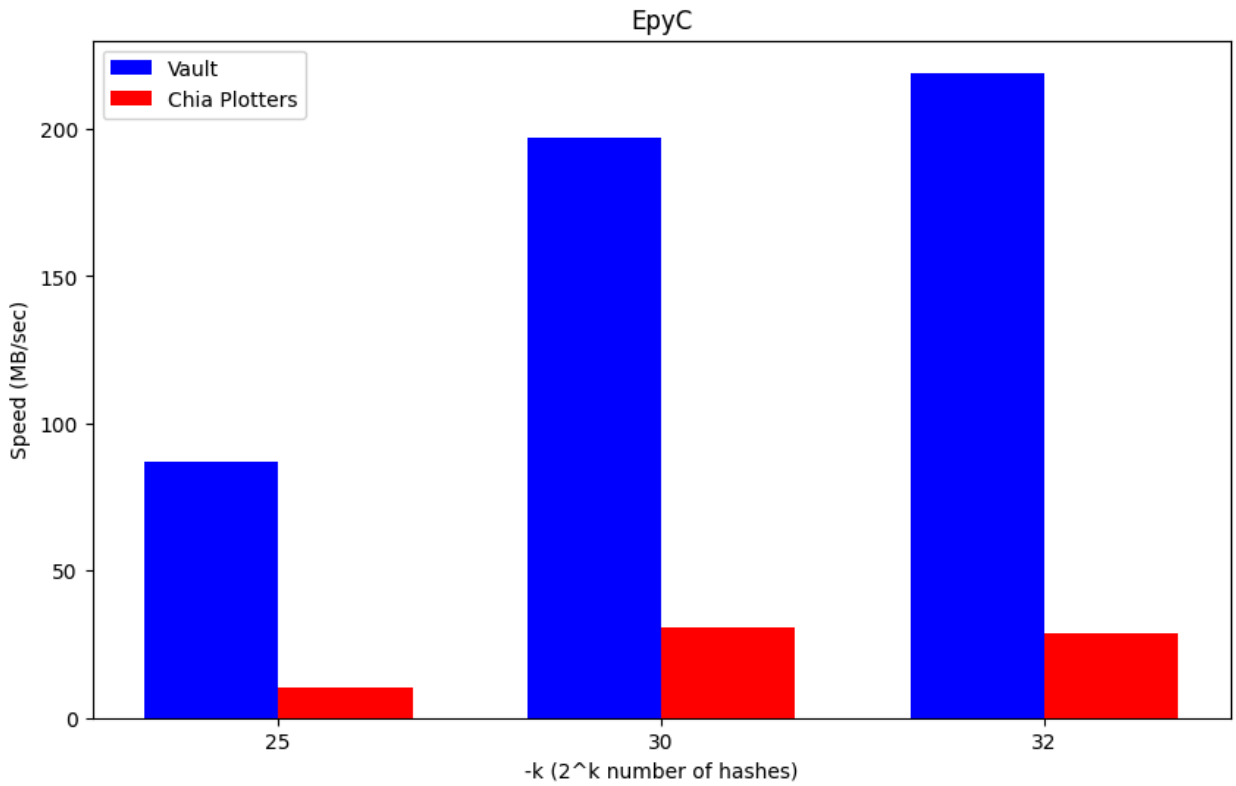
PS: The data highlighted in yellow is our implementation, and data in white is Chia's.

| athena | | | |
|---|---|---|---|
| k | file size | time | MB/sec |
| 25 | 1024 | 29.69 | 34.49 |
| 25 | 635.788595 | 37.2693 | 17.0593114 |
| 30 | 32768 | 456.27 | 71.82 |
| 30 | 24395.85 | 2077.53 | 11.7427178 |
| 32 | 131072 | 2221.65 | 59 |

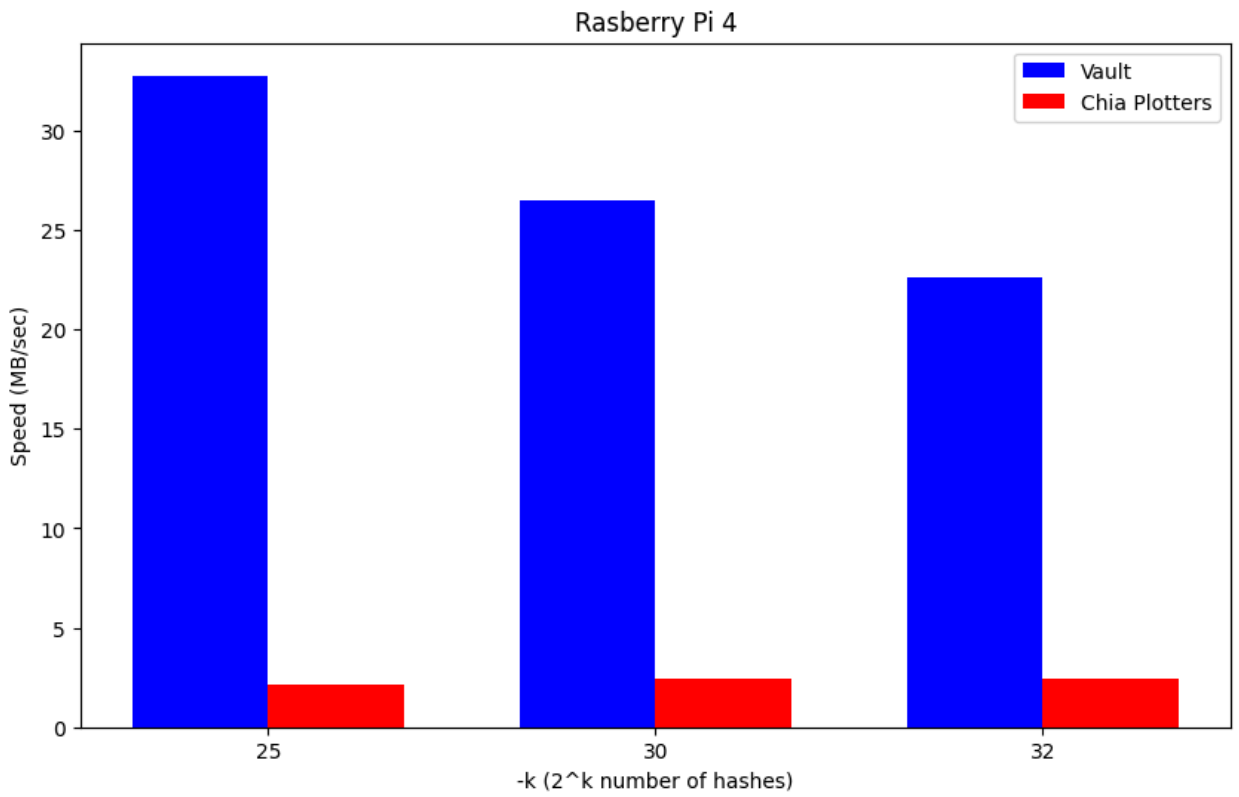
| | | | |
|---|---|---|---|
| 32 | 103793.6 | 21694.3 | 4.78437343 |



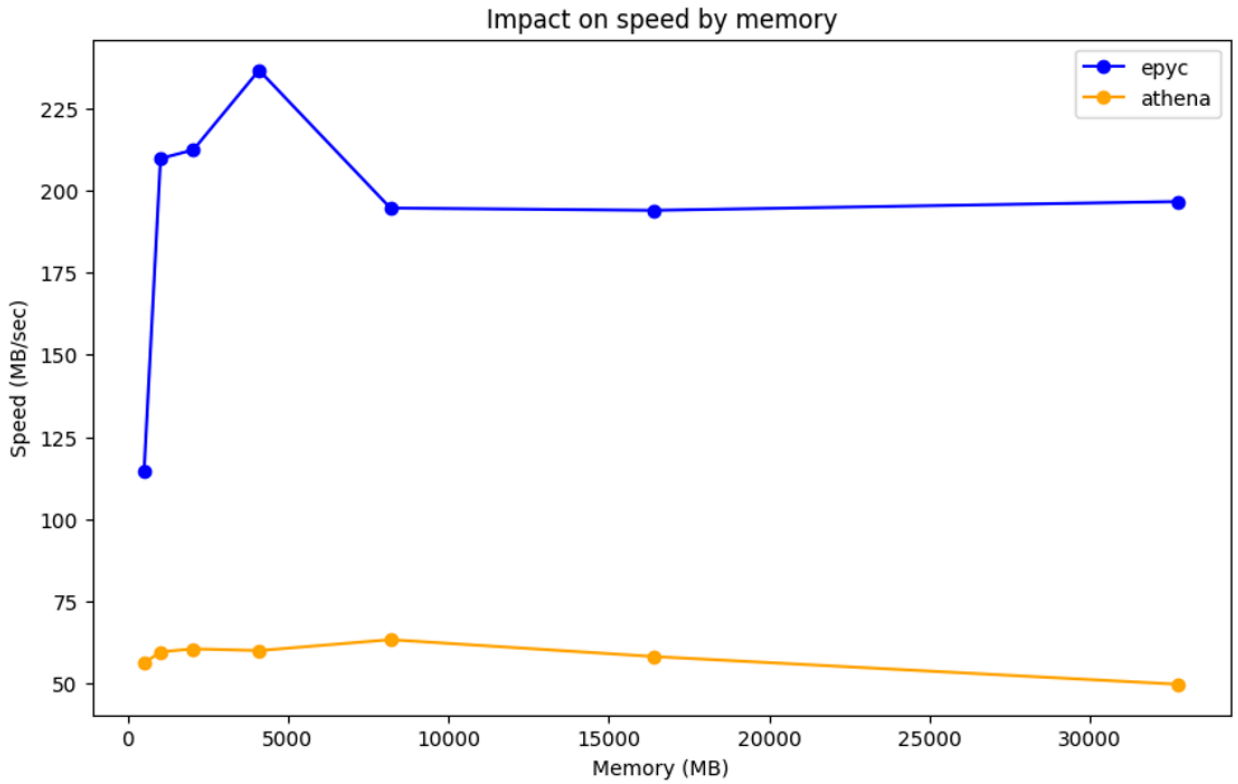

| epyc | | | |
|---|---|---|---|
| k | file size | time | MB/sec |
| 25 | 1024 | 11.75 | 87.13 |
| 25 | 641.7595 | 62.5986 | 10.2519778 |
| 30 | 32768 | 166.2 | 197.16 |
| 30 | 24389.26 | 795.544 | 30.6573307 |
| 32 | 131072 | 598.26 | 219.09 |
| 32 | 103793.414 | 3626.72 | 28.6190866 |



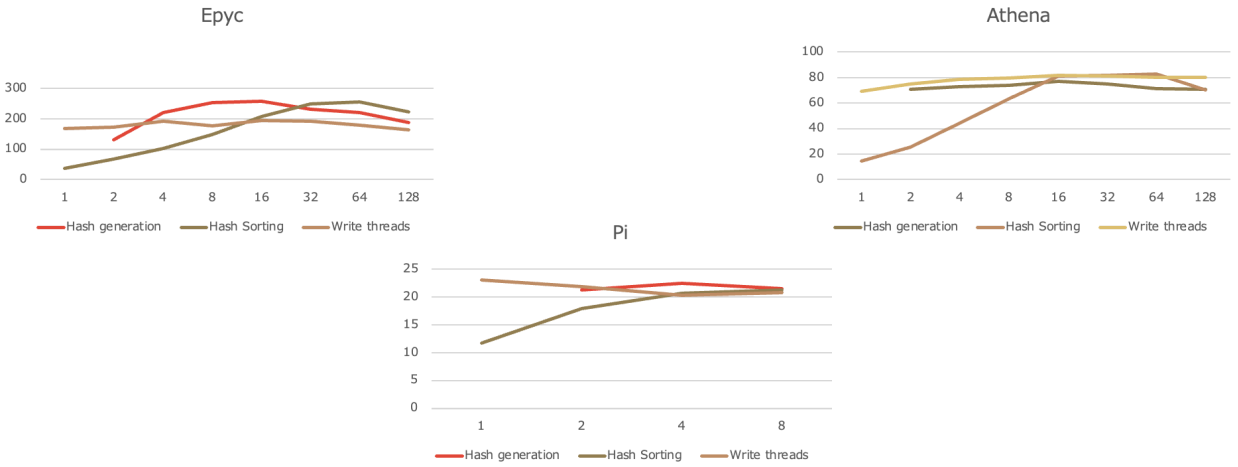

| pi | | | |
|---|---|---|---|
| k | file size | time | MB/sec |
| 25 | 1024 | 31.24 | 32.77 |
| 25 | 638 | 302.101 | 2.11187649 |
| 30 | 32768 | 1235.56 | 26.52 |
| 30 | 25560 | 10614 | 2.40814019 |
| 32 | 131072 | 5792.42 | 22.63 |
| 32 | 103424 | 43094 | 2.39996287 |




In addition to our comparison with Chia, we conducted benchmarking to evaluate the effect of varying memory sizes on our code. Initially assuming that performance would linearly increase with memory sizes, with a potential plateau toward larger capacities, we observed a contrasting outcome. Surprisingly, in both cases, performance reached its peak at specific memory sizes before experiencing a decline. This unexpected finding underscores the complexity of optimizing performance and highlights the need for nuanced analysis beyond initial assumptions.




In our comprehensive benchmarking process, we further examined the influence of varying thread counts on distinct segments of our code. Specifically, we analyzed three critical parts where thread count significantly affects performance: hash generation, sorting, and input/output operations. By systematically varying thread counts at different stages, we gained valuable insights into how concurrency impacts each aspect of our implementation. This meticulous analysis sheds light on optimal thread utilization strategies and enhances our understanding of parallel processing dynamics within the codebase.



**Future Work:**

Currently, our code efficiently compresses the stored hash from 27 bytes to 11 bytes and 3 bytes, showcasing promising compression capabilities. To explore the potential for further compression, we intend to conduct tests using larger k sizes and compare the results against Chia. Additionally, we aim to investigate alternatives to qsort, considering whether a less memory-intensive sorting method could enhance overall performance. By systematically evaluating these aspects, we endeavor to optimize our implementation for improved efficiency and competitiveness within the cryptocurrency landscape.