2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)

Demystifying Crypto-Mining: Analysis and Optimizations of memory-hard PoW Algorithms

Runchao Han, Nikos Foutris, Christos Kotselidis
School of Computer Science
The University of Manchester, UK
runchao.han@student.manchester.ac.uk, {nikos.foutris, christos.kotselidis}@manchester.ac.uk

***Abstract*—Blockchain technology has become extremely popular, during the last decade, mainly due to the successful application in the cryptocurrency domain. Following the explosion of Bitcoin and other cryptocurrencies, blockchain solutions are being deployed in almost every aspect of transactional operations as a means to safely exchange digital assets between non-trusted parties.**

At the heart of every blockchain deployment is the consensus protocol, which maintains the consistency of the blockchain upon satisfying incoming transactions. Although many consensus protocols have been recently introduced, the most prevalent is Proof-of-Work, which scales the blockchain globally by converting the consensus problem to a competition based on cryptographic hash functions; a process called "mining".

The Proof-of-Work consensus protocol employs memory-hard algorithms in order to counteract ASIC or FPGA mining that may compromise the decentralization and democratization of the blockchain. Unfortunately, this leads to increased power consumption and scalability challenges since numerous processing units such as GPUs, FPGAs, and ASICs, are required to satisfy the ever-increasing operational requirements of blockchain deployments.

In this paper, we perform an in-depth performance analysis and characterization of the most common memory-hard PoW algorithms running on NVIDIA GPUs. Motivated by our experimental findings, we apply a series of optimizations on Ethash algorithm, the consensus protocol of the Ethereum blockchain. The implemented optimizations accelerate performance by 14% and improve energy efficiency by 10% when executing on three NVIDIA GPUs. As a result, the optimized Ethash algorithm outperformed its fastest commercial implementation.

***Keywords*-Blockchain, Crypto mining, GPUs, Ethereum, Optimizations, Energy Efficiency**

I. Introduction

Blockchain technology [1] has become prevalent since its successful application in the cryptocurrency domain. Bitcoin [2] is the first and most popular cryptocurrency with a market capitalization of ten billion US dollars in 2016 [3]. The objective of cryptocurrencies is to democratize and anonymize the currency by introducing a decentralized, censorship-resistant network. Bitcoin, as well as numerous other cryptocurrencies such as the Litecoin and Ethereum, use the Proof-of-Work (PoW) consensus protocol. This protocol utilizes cryptographic hash functions to acquire the consensus among a network of participants; a process which is also called "mining". Currently, the PoW consensus protocol constitutes the primary option for public blockchain deployments.

Initially, the Bitcoin algorithms [4] ran on CPUs while, soon after, GPUs, FPGAs, and ASICs implementations were introduced in an attempt to control the mining power of the network and generate more bitcoins. However, the proportional relationship between the mining power and the generated income, in the form of mined Bitcoins, initiated a continuous performance race between the various implementations at the expense of putting the democratization of the network at risk. In a nutshell, the more mining power a particular party or a group of parties have, the higher the probability will be to control over 51% of the mining power and therefore have the ability to "fork" the blockchain [5]. Additionally, the cryptocurrency domain consumes huge amounts of energy just to maintain the consensus. The results are not reusable, and this computation is therefore wasteful. For instance, the Bitcoin network consumed 42TWh in 2017, which was estimated to be higher than that of the Republic of Ireland [6].

As an attempt to avoid compromising the decentralization and democratization of the blockchain network and addressing the high energy demands, numerous alternatives, which replace the typical, computational-bound PoW algorithms, have been proposed. Such approaches that include Proof-of-Stake (PoS) [7], Delegated Proof-of-Stake (DPoS) [8] and the conventional Byzantine Fault Tolerance (BFT) [9] consensus protocols, do not require mining. However, from a technical perspective, they face scalability issues, since they rely on a number of fixed nodes to verify the transactions. Additionally, to combat the mining monopoly, Bitcoin successors have introduced the memory-hard PoW algorithms, which randomly combine compute and memory access operations. As a result, the "wealthy" participants can take less advantage of their high-end machines since they are bounded by the limited memory bandwidth, the high access latency and its scarce capacity. Thus, although the memory-hard PoW algorithms are energy-inefficient, they partially solve the problem of democratization.

This paper focuses on understanding the cryptocurrency mining process by performing an in-depth performance characterization of the most common state-of-the-art memory-

978-1-7281-0746-2/19/$31.00 ©2019 IEEE
DOI 10.1109/ISPASS.2019.00011

hard PoW algorithms. Then, motivated by our findings, we apply a series of optimizations aiming at improving their performance and energy efficiency. In detail, in this paper:

- We formalize and analyze the structure of state-of-the-art memory-hard PoW algorithms.
- We characterize the three most-popular memory-hard PoW algorithms such as the Ethash, CryptoNight and Scrypt, running on NVIDIA GPUs.
- We perform a series of code optimizations on Ethash, the consensus protocol of Ethereum blockchain. These optimizations resulted in more than 14% performance speedup (measured in Hashrate) and 10% energy-efficiency improvement (measured in Hashes/Joule), when running on GPUs of different classes. Additionally, our optimized open-source Ethash algorithm outperforms Claymore; the fastest commercial Ethereum mining software. The proposed optimization strategy can be seamlessly applied to other PoW algorithms.

The remaining of the paper is organized as follows: Section II provides the background information on blockchain technology and consensus protocols while Section III outlines the structure of memory-hard PoW algorithms. In Section IV, we present the experimental analysis of the selected PoW algorithms, while Section V describes our performance and energy efficiency optimization strategy. Finally, Section VII concludes the paper.

II. Blockchain and Consensus Protocols

Blockchain is the core technology of Bitcoin, the first and most well-known cryptocurrency. The Bitcoin's source code as well as its CPU mining toolkit were initially introduced in 2009 [10]. A year later, the GPU miner source code written in OpenCL [11] was released signifying the beginning of the GPU mining era. In 2011, the first FPGA Bitcoin miner was published [12] while in 2012, Butterfly Labs announced the production of the first ASIC miner [13]. Following Butterfly Labs, multiple ASIC miner providers such as ASICMiner [14], Canaan [15], BitFury [16] and Bitmain [17] emerged. ASIC- and FPGA-based mining yield more profit than the GPUs or CPUs which is attributed to the higher performance and energy efficiency. However, they put the decentralization and democratization of the Blockchain network at risk, since the participants with high financial capabilities can potentially control the mining power and therefore have the ability to fork the blockchain. On the contrary, the overwhelming majority of regular miners rely on GPU setups due to their cost-efficiency. GPU mining has also prevailed due to the ease of programmability, when compared to ASIC and FPGA, and the higher performance compared to CPUs.

Blockchain technology is a distributed ledger shared across a network of participants [18]. The data on a blockchain is organized in blocks, where each block is connected to the previous in a chain-like structure. Additionally, there is no central entity to ensure the consistency among the ledgers. As a result, when a new block is added, a consensus algorithm is executed to maintain the consistent blockchain state. For example, Bitcoin has innovatively introduced the Proof-of-Work consensus protocol, which is a transformation of the Byzantine Fault tolerance algorithm, to reach consensus among the untrustworthy network of participants.

Originally, the Proof-of-Work algorithms were designed for protecting against spam emails [19] and denial-of-service attacks. Currently, they have been successfully applied to other application domains, such as cryptocurrencies and password hashing [20]. Additionally, memory-hard PoW consensus protocols have been designed to counteract ASICs' and FPGAs' mining power that may compromise the decentralization and democratization of the blockchain. However, they have been highly debated due to their extreme energy consumption [21]–[25], since numerous processing elements (GPUs, FPGAs, and ASICs) are required to satisfy the ever-increasing operational requirements of the blockchain deployments. Nevertheless, a wide range of cryptocurrencies, such as Scrypt [26], CryptoNight [27], Ethash [28], X11 [29] and Equihash [30], adopt memory-hard consensus protocols.

The kernel of a Proof-of-Work algorithm is the calculation of a hash function. In particular, during the mining process, each node of the network calculates a hash value of a constantly changing block header. The consensus algorithm requires that value to be less than a given threshold, which is called difficulty. This threshold determines the competitive nature of the mining process, since the more computing power is added to the network, the higher the threshold will be. Finally, the number of produced hash values per second (called *Hashrate*) is the metric used to evaluate the efficiency of Proof-of-Work algorithms.

III. Memory-hard PoW algorithmic analysis

This section formalizes the structure of a typical memory-hard Proof-of-Work consensus protocol in the context of the three most well-known PoW cryptocurrency algorithms: Ethash, CryptoNight, and Scrypt.

Algorithm 1 presents the pseudo-code of a generic memory-hard PoW algorithm which is logically divided to the following three execution phases:

1) *Initialization*: *Scratch_pad* generation (Line 1).
2) *Memory-hard Loop*: Compute hash value and access *Scratch_pad* memory (Lines 3 to 6).
3) *Finalizing*: Final output value formatting (Line 7).

A typical PoW algorithm works as follows: At first, a relatively large data block is generated (called the *scratch_pad*) based on a *Nonce* value and allocated in memory (Line 1). Consequently, the larger the *scratch_pad* is, the higher the memory requirements of the algorithms are. Note that the *Nonce* is typically a 32 bit arbitrary value that is used once. The operations in Lines 3 to 6 introduce the memory-intensive section of the algorithm that dominates the execution time. In each loop iteration, a small segment *Z* of the *scratch_pad* is randomly accessed. This segment is indexed by hashing the authenticated value (the *proof*) generated from the previous iteration. To enforce data dependencies and increase the memory requirements of the PoW algorithms, the *proof* is constantly updated by mixing the *Z*, *proof* and *Nonce* values (Line 5). As a result, the sequence of *scratch_pad* memory accesses (Line 4) is unpredictable due to the randomness of the *proof* value and, thus, improves the memory-hardness of the loop. Finally, the *finalize* routine maps the arbitrary sized *proof* value to a fixed-length string to be the final output (Line 7).

Algorithm 1: The pseudocode of a typical memory-hard Proof-of-Work algorithm.

```
Data: Random value: nonce
Result: Fixed-length proof: proof
1 scratch_pad := generate_scratch_pad(nonce);
2 proof := init_proof(nonce);
3 for i ← 0 to num_rounds do
4     z := scratch_pad[ hash(proof) ];
5     proof := mix(nonce, proof, z);
6 end
7 proof := finalize(proof);
```

According to Algorithm 1, by carefully selecting the following parameters we can maximize its memory hardness:

- *scratch_pad*: The data block allocated in memory.
- *num_rounds*: The number of iterations of the memory-hard loop.
- *hash*(*proof*): The hash function for randomly accessing the *scratch_pad* memory.
- *Z*: The segment size of each memory access.
- *mix*(*nonce*, *proof*, *z*): The mixing function for adding data dependencies.

Table I: The parameters of the memory-hard Ethash, CryptoNight and Scrypt.

	Ethash	**CryptoNight**	**Scrypt**
scratch_pad size	∼1GB	4KB	128KB
scratch_pad generation	Per 30K blocks	Ad-hoc	Ad-hoc
Number of iterations	64	524,288	1024
Segment size	128B	8B	128B
Memory access pattern	Read	Read and write	Read
Hash function	SHA3	Keccak	PBKDF2

To put this analysis into the perspective of commercial PoW algorithms, we assess the implications of the aforementioned set of algorithmic parameters on Ethash, CryptoNight and Scrypt algorithms (Table I). Ethash, which is a Proof-of-Work algorithm used on Ethereum cryptocurrency, is designed to be ASIC-resistant due to the high memory requirements. In particular, Ethash uses a custom-generated 1 GB direct acyclic graph (DAG) dataset representation (the *scratch_pad*), which is re-generated for every 30,000 blocks. The DAG dataset is structured as a two dimensional array of 4-bytes unsigned integer values. Therefore, the majority of a miner's effort is devoted to perform memory-hard computations on the *scratch_pad*. On the other hand, CryptoNight and Scrypt (adopted by Litecoin [31]) have small datasets (4KB and 128B, respectively), which are generated on-the-fly using the *nonce* as a seed. This characteristic indicates that the mining threads can potentially run in parallel without the need for inter-thread communication, since each mining iteration can have a unique dataset. As a result, the more parallel mining threads are used, the higher memory bandwidth will be required. In particular, the CryptoNight algorithm initializes the *scratch_pad* with pseudo-random data. This data is stored in memory (e.g. DRAM) and numerous read and write operations are initiated at pseudo-random memory locations within the scratchpad. This random access pattern results in low cache memory utilization. Finally, the algorithm hashes the entire *scratch_pad* to produce the final output. Similarly, Scrypt generates and allocates the *scratch_pad* to a high latency memory and, then, reads the data in a random way. Overall, Ethash, CryptoNight and Scrypt are bound by the memory bandwidth, latency and size. As such, they take less advantage of the high computational capacity of application-specific hardware and contribute to the decentralization and democratization of the blockchain network.

IV. Memory-hard PoW Experimental Analysis

In this section, we present the experimental results as derived from the comprehensive performance characterization of Ethash, CryptoNight and Litecoin Scrypt. We start by measuring the hashrate of the selected algorithms, when running on three NVIDIA-based computing systems, and then we conduct a fine-grained analysis on the implications that various hardware and software characteristics have on the performance of each PoW algorithm. As our experimental analysis highlights, the size of the *scratch_pad* and the data dependencies generated by the *mix* routine are the two features that bound the performance of the memory-hard PoW algorithms.

A. Experimental Setup

GPUs are the mainstream hardware used for executing memory-hard PoW algorithms due to the ease of programmability, the high memory throughput, and the low to moderate cost. In our experimental setup, we employ the following three classes of commodity GPUs: NVIDIA GeForce GTX 960M [32], NVIDIA GeForce Titan X Pascal [33], and NVIDIA Quadro GP100 [34]. These model three systems with different compute and memory capabilities (Low, Intermediate, High). Table II presents the configuration details of each system.

In terms of software configuration, we have selected the following PoW algorithms:

- **Ethminer** [28]: The most well-known and open-source Ethereum miner.
- **CCminer** [35]: An implementation of CryptoNight [27] which is currently used in various cryptocurrencies, such as Monero [36].
- **Scrypt** [26]: Used in Litecoin [31] and others [37].

B. High-Level PoW Characterization

We evaluate the performance, measured in *hashrate*, of Ethash, CryptoNight and Scrypt running on top of the three computing systems. The *hashrate* of each algorithm was sampled for every second of operation, using the miners' inherent benchmarking modules [38], [39]. In turn, the samples were averaged after completing 60 seconds of execution time.

As shown in Table III, the *hashrate* of Ethash is higher than Scrypt's, while Scrypt's *hashrate* is greater than CryptoNight's. Additionally, this performance trend is consistent across all computing systems, while the highest *hashrate* was measured on the machine with the highest memory capabilities (i.e. "High"). Furthermore, the performance variation of the PoW algorithms is attributed to the different amount of memory-hard loop iterations. Clearly, the *hashrate* is inversely related to the number of iterations. Therefore, the *hashrate* of Ethash is the highest, since it has the smallest amount of loop iterations (Table I). Finally, it is important to note that the profit of a miner is determined by the portion of its computing power amongst the whole network of participants, rather than the mining power itself; meaning that higher hashrate does not necessarily translate to higher profits.

Next, we measure the execution time distribution of the three execution phases (described in Section III) when executing Ethash, CryptoNight and Scrypt on the "intermediate" computing system (Figure 1). The selection of the "intermediate" machine to further analyze the performance implications was based on the fact that it represents a moderate, commodity hardware setup (this is also the baseline setup for all characterization experiments). As analyzed in Section III, Ethash's *scratch_pad* is generated once per 30,000 blocks, while in Scrypt and CryptoNight a new *scratch_pad* is produced for every input *Nonce*. To circumvent this variation and perform unbiased comparison between the algorithms, we sampled Ethash on every block, which equals to 1/30,000 of the original generation time. Finally, the execution time distribution presented in Figure 1 is the output of averaging 100 disjoint runs.

As our experimental results demonstrate, Ethash and CryptoNight spend most of their execution time on the memory-hard loop phase. This is attributed to the generation of random memory access patterns which inevitably leads to a low cache hit rate and to frequent, high-latency main memory accesses. To verify this behaviour, we used NVIDIA's `nvprof` to measure the L1 cache hit rate of each application running on the intermediate computing system. As expected, the L1 cache hit rate was quite low, ranging from 0% to 20%, as measured on Scrypt. Therefore, the majority of the memory requests were fetched from main memory. However, it was infeasible to extract more fine-grained information about the data layout on the memory system, since NVIDIA does not disclose any details about the cache hierarchy or the internals of the `nvprof` profiling infrastructure. Another interesting finding was that Scrypt does not exhibit the same execution breakdown like the rest of the algorithms. On the contrary, the execution time is almost uniformly distributed among the initialization and the memory-hard loop phase, while the finalizing phase is minimal. This behaviour is attributed to the particular implementation of Litecoin's Scrypt. According to our algorithmic-level analysis, the dataset generation of Litecoin's Scrypt is an iterative process, where a new random data segment is produced in each loop iteration. Moreover, the initialization phase mainly consists of time-consuming write operations, compared to the lightweight read-only operations encountered during the memory-hard loop phase.

Overall, across all algorithms, the majority of the execution time is spent, as expected, on the memory-hard loop. Thus, the optimization strategy presented in Section V focuses on this phase.

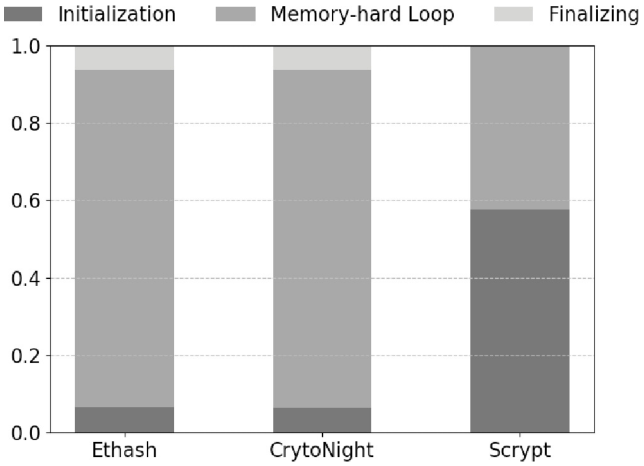

Figure 1: Execution time breakdown of the Ethash, CryptoNight, and Litecoin Scrypt running on the intermediate system.

C. Low-level PoW Characterization

To better understand the implications of the memory-hardness on the performance of Ethash, CryptoNight, and Scrypt, we conducted further low-level characterization experiments. In particular, we analyzed the following low-level attributes:

- Compute and main memory utilization.
- Throughput of load and store operations.
- Peak memory usage.

Table II: The computing systems configuration.

	Low	**Intermediate**	**High**
CPU	Intel Core i7-6700HQ @ 2.60GHz	Intel Core i7-4770 @ 3.40GHz	Intel Core i7-7700K @ 4.20GHz
Main memory	24GB	16GB	64GB
Disk	210GB	2.6TB	1.9TB
GPU	960M (Maxwell)	TitanXP (Pascal)	GP100 (Pascal)
GPU memory	4GB	12GB	16GB
Memory Bandwidth	80.19 GB/s	547.6 GB/s	732.2 GB/s
Operating system	Ubuntu 18.04	Ubuntu 18.04	Centos 7
Nvidia driver	390.48	390.48	384.111
CUDA	9.1	9.1	9.0
GCC	5.4.0	5.4.0	4.8.5

Table III: The hashrates of Ethash, CryptoNight, and Scrypt on the different computing systems. Note that *hashrate* is measured in KH/s.

	Low	**Intermediate**	**High**
Ethash	8690	31990	70470
Scrypt	41.33	926.61	1586.09
CryptoNight	0.06	0.79	1.56

- Distribution of PTX assembly instructions.
- Execution stall conditions.

At first, we measure the compute and the main memory (DRAM) utilization of each PoW workload (Figure 2). In particular, the compute utilization is measured as the fraction of the *active warps* to the *maximum number of concurrent warps*, while the memory utilization equals to the *used memory bandwidth* over the *maximum available memory bandwidth*. Intuitively, the DRAM utilization should be higher than the compute utilization since the memory is the bottleneck.

Notably, CryptoNight spends only 10% of its execution on computing, while the memory utilization is only at 20%. The low utilization of the compute and the memory resources is due to the existence of the time-consuming, non-overlapped write operations, which dominate the memory accesses. On the contrary, Ethash and Scrypt obtained memory utilization of up to 60%, which is near the practical limit. This is attributed to the fact that a large portion of Scrypt's *scratch_pad* can fit into the L1 cache. Moreover, CryptoNight differs from Ethash and Scrypt due to the write operations to the *scratch_pad*, which result to intensive memory accesses with few data transfer instructions within the GPU's memory. As a result, CryptoNight is the most memory-hard algorithm, followed by Ethash and Scrypt, since it has the lowest compute utilization.

According to the utilization analysis, memory is the source of the performance bottleneck on the selected memory-hard PoW algorithms. In particular, this behaviour may be attributed to either the limited available memory bandwidth or the scarce capacity. Subsequently, we performed a memory throughput analysis as well as we measured the peak memory usage in order to accurately understand the reasons that bound the performance. To do so, we used NVIDIA's `nvprof` profiler to record the memory throughput. In addition, we used NVIDIA's system management interface (`nvidia-smi`) to record the peak memory usage. Finally, the experiments were conducted on the intermediate computing system.

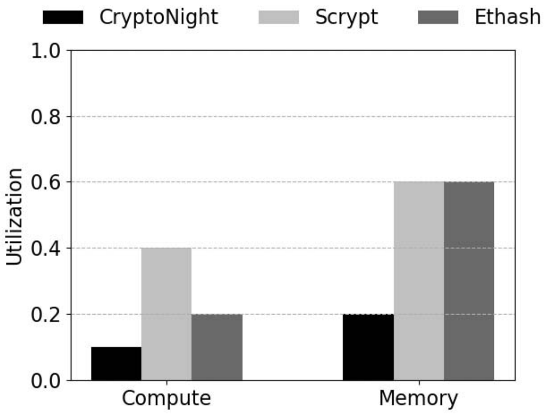

Figure 2: The distribution of the compute versus main memory utilization.

Figure 3 presents the amount of Load/Store operations that access the Shared Memory, Texture Cache, L2 Cache and DRAM. In particular, Ethash and Scrypt occupy almost the maximum available load bandwidth on the texture cache, L2 cache and DRAM, while the store throughput is substantially lower. This indicates that Ethash and Scrypt have frequent load operations. On the contrary, CryptoNight falls behind the maximum load throughput, while store operations are executed more frequently than Ethash and Scrypt. Finally, Ethash does not utilize the texture cache.

Interestingly, only CryptoNight uses the Shared memory. This is because only the 4KB size *scratch_pad* of Cryptonight can fit into the Shared memory, which is 64KB. On the contrary, the *scratch_pad*s of Ethash and Scrypt exceed the capacity of the Shared Memory (1GB and 128KB, respectively). Regarding the texture cache, which is actually read-only, the store throughput is consistent with the L2 cache for CryptoNight and Scrypt. The reason is that the metric we used is *l2_tex_write_throughput*, which actually includes the operations for overwriting itself based on the cache replacement policy. Therefore, if a variable is cacheable for both the texture cache and the adjacent L2 cache, the read and write throughputs are consistent. Finally, only Ethash did not utilize the Texture cache completely. This is due to the fact that the mining threads of Ethash communicate mainly by the warp shuffle intrinsic rather than using the Texture cache.

Figure 4 shows the peak DRAM usage for each algorithm. As shown, Scrypt utilizes significantly more memory than CryptoNight and Ethash. However, the overall usage is approximately only half of the total DRAM size. Furthermore, Ethash uses slightly more than 1GB of memory which is the lowest amount among all algorithms.

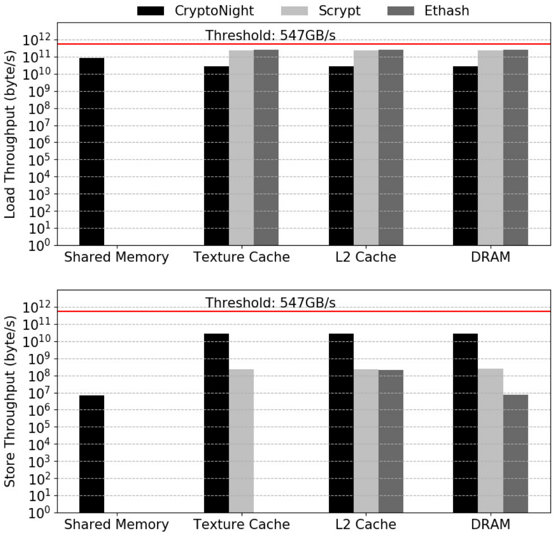

Figure 3: The Load, Store operations throughput on each level of the memory hierarchy.

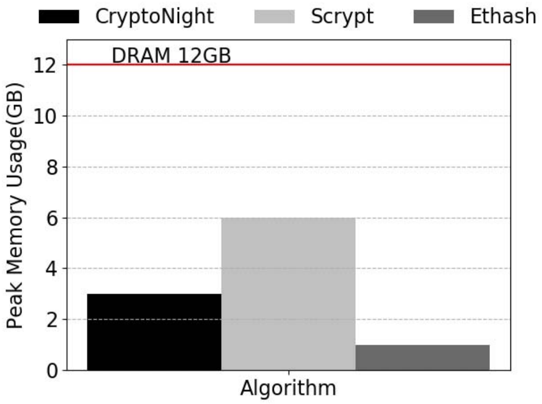

Figure 4: The peak memory usage of each algorithm on the intermediate computing system.

Next, we measure the frequencies of various instruction types as an attempt to further understand the performance of Ethash, CryptoNight, and Scrypt. Again, the experiments were performed on the intermediate machine and the results were collected using `nvprof`.

- Integer Arithmetic Instructions.
- Floating-Point Instructions.
- Load/Store Instructions.
- Control Flow Instructions.
- Inter-Thread Instructions.
- Miscellaneous Instructions (e.g. barrier synch, etc.).

As shown in Figure 5, the key highlight is that integer arithmetic instructions have the highest frequency in all PoW algorithms. Additionally, CryptoNight has executed more memory instructions (14%) than Ethash and Scrypt. Moreover, Ethash and Scrypt have less than 5% of inter-thread instructions, while CryptoNight did not issue any instructions of this type.

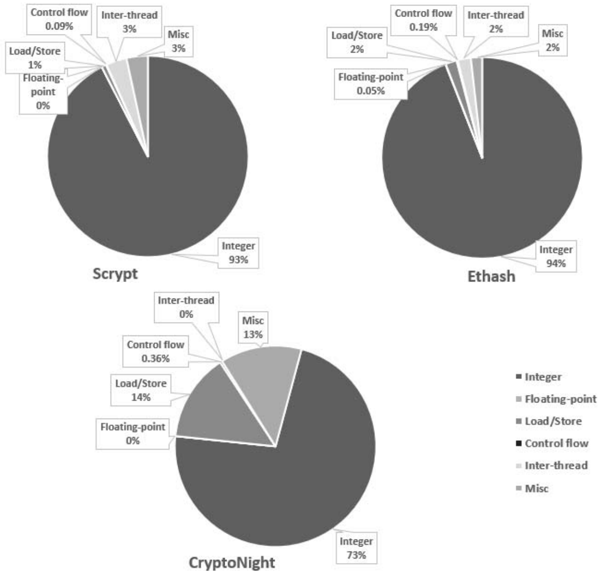

Figure 5: Instruction type frequencies on Ethash, CryptoNight and Scrypt algorithms. For clarity, the data labels of the instruction types with tiny frequencies were omitted.

Our results show that "memory-hardness" does not directly translate to memory instructions. According to the algorithms' structures, within the memory-hard loop phase a mixed value is generated (*proof*), which enforces data dependencies. In particular, in each loop iteration *proof* combines multiple variables along with segments from the *scratch_pad* memory. In addition, the mixing function consists of complex cryptographic processes that utilize a significant number of integer instructions. Nevertheless, the majority of the integer instructions only take one or two cycles to execute and, therefore, have minimal effect on the performance. Additionally, CryptoNight consumes more memory instructions than the other two algorithms, making the compute utilization lower while keeping the memory fully utilized. This is because the *scratch_pad* is read and written twice in an interleaved manner rather than simply read once like in Ethash and Scrypt.

To further analyze the performance inefficiencies of the Ethash, CryptoNight and Scrypt algorithms, we study which of the following execution stall events have the highest frequency.

- *Memory Throttle*: A large number of pending memory operations prevent forward progress.
- *Not Selected*: A warp is ready to be issued but the scheduler selects another.
- *Instruction Fetch*: An instruction fetch is pending; a delay usually caused by branch divergence.
- *Execution Dependency*: An instruction requires an input, which has not been available yet.
- *Data Request*: Wait to fetch data from memory.
- *Texture*: The texture subsystem is fully utilized or has too many outstanding requests.
- *Synchronization*: A warp is blocked at a *_syncthreads*() call.
- *Immediate Constant*: A constant load is blocked due to a miss in the constant cache.
- *Pipe busy*: No available resources.
- *Other*: Register bank conflicts, wraps waiting to resolve branches, etc.

The frequency of the aforementioned instruction stall conditions is shown in Figure 6. As expected, the execution pipeline was mainly stalled due to data requests. This is also justified by the algorithms' structures, in which memory is randomly accessed. Additionally, CryptoNight is stalled for data requests more frequently than Ethash and Scrypt (97%, 90% and 79%, respectively). As a result, the high frequency of data requests adversely affects the performance of memory-hard PoW algorithms. The next most frequent stall events, which are independent of the particular workload, are attributed to instruction fetch and execution dependency conditions.

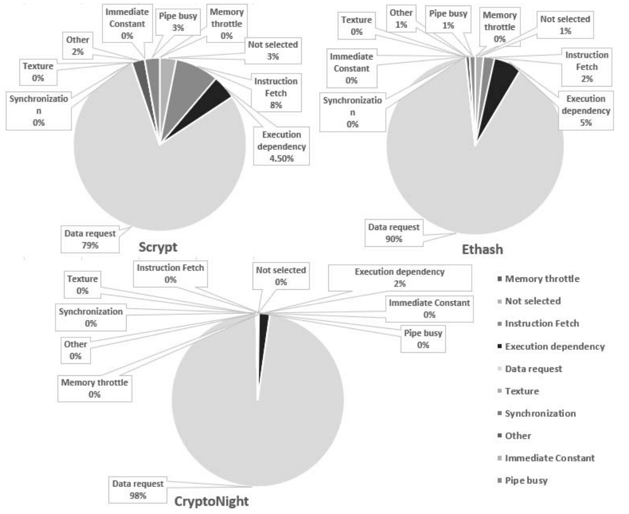

Figure 6: The frequency of the various instruction stall conditions on the execution pipeline. For clarity, the data labels of the instruction types with negligible frequencies were omitted.

D. *Design guidelines to maximize memory-hardness*

The PoW consensus protocols employ memory-hard algorithms in order to counteract ASIC or FPGA mining that may compromise the decentralization and democratization of the blockchain. To achieve that, the mining process has to be memory-intensive, while the computing power requirements need to be minimized. This section summarizes a set of guidelines that maximize PoW algorithms' memory-hardness, as derived from Ethash's, CryptoNight's and Scrypt's performance characterization:

- **Frequent and arbitrary memory accesses:** In the memory-hard loop phase, the frequent and random memory accesses are the sources of the memory-hardness, since they generate "data request" and "execution dependency" stalls. Thus, the more unpredictable the memory access locations are, the higher the memory-hardness of the PoW will be.
- **Generate large datasets:** The dataset should be large enough, so that it will not fit in caches and all accesses will be propagated to DRAM. As a result, the memory-hard PoW algorithms will depend on DRAM's bandwidth and latency.
- **Reducing the computing overhead of the mixing function:** While maintaining the necessary data dependencies, the compute overhead of the mixing function (Line 5, on Algorithm 1) should be minimized. As a result, the memory requirements will be even more exaggerated compared to the compute power needs.

V. Optimizing memory-hard PoW algorithms

This section describes and evaluates the proposed optimizations on Ethash. As our experimentation vehicle, we have selected the Ethash algorithm, since it supports one of the largest blockchain networks globally; Ethereum. Nevertheless, the proposed optimizations can seamlessly be applied to Scrypt and CryptoNight.

We evaluate the performance of the unmodified and optimized Ethash algorithm on all computing systems ("low", "intermediate" and "high") presented in Table II. On the contrary, the power measurements were performed on the "intermediate" and "high" systems, since the `nvidia-smi` did not work out-of-the-box on the "low" computing system. However, this type of GPU is rarely used for mining due to their low performance. The *hashrate* (H/s) was collected by the inherent monitoring tool of *ethminer*, and the power (Watts) was measured with NVIDIA's `nvidia-smi` interface. Finally, note that the *hashrate* and power consumption were averaged over a time period of 60 seconds (i.e. a value was collected for each second of execution time).

The configuration parameters of Ethash are presented in Table I. In particular, the Ethash memory-hard loop contains 64 iterations, each of which randomly fetches a 128-byte segment from the *scratch_pad* (*d_dag* variable) and mixes (*fnv4* function) the segment with the output value. The *scratch_pad* is accessed as an array of segments having an integer, pseudo-randomly generated value as index. Therefore, the random access of *scratch_pad* is exactly the source of memory-hardness.

A. Optimization strategy guidelines

Our objective is to optimize the open-source cryptomining implementations on GPUs. Moreover, we focus only on system-level optimization techniques since the PoW consensus protocols are fixed and any alternations to them will not be applicable to the blockchain. According to our characterization results, the cache memories have quite low hit rates (up to 20% hit rate on L1 cache memory). Additionally, the execution time is delayed mainly due to "data request" stalls. As a result, the limited memory bandwidth and the low cache hit rate bound the performance of the memory-hard PoW algorithms. Thus, in order to tackle the aforementioned issues, the proposed optimizations are based on the following guidelines:

1) Overlap the compute and memory operations.
2) Increase data lifetime on the cache memory.

B. Optimization Techniques

Based on the optimization guidelines, we apply data prefetching and software pipelining on Ethash as illustrated in Figures 7 and 8, respectively.

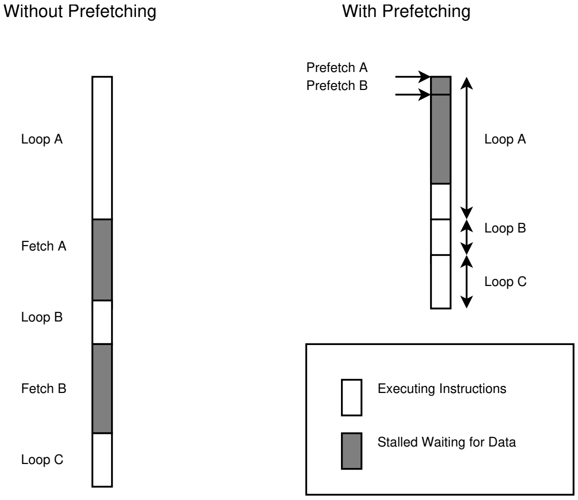

Figure 7: Data Prefetching.

*1) **Software-based Data Prefetching**:* Data prefetching is used to hide the memory latency and increase the available parallelism at the instruction level or at the thread level. As shown in Figure 7, through this technique we attempt to hide the time-consuming memory accesses to the main memory by fetching data to the local cache before it is actually needed. Therefore, when a memory access operation is executed, it will not be stalled, thereby having a positive impact on the performance [40].

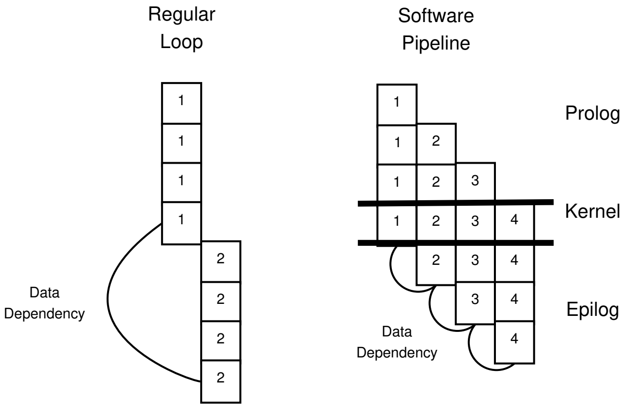

Figure 8: Software Pipelining.

Data prefetching provides a way of explicit cache management. However, if prefetching is not accurate enough, it can degrade performance (and increase power) by polluting the cache and by wasting shared resources. On the contrary, accurate prefetching can reduce the average memory latency and lower the need for larger cache memories.

*2) **Software Pipelining**:* Besides data prefetching, software pipelining is another way to hide the high latency of accessing the main memory. Software pipelining is based on interleaving the data-independent operations of subsequent loop iterations. As shown in Figure 8, the last steps of subsequent loop iterations have data dependencies (Fig. 8 - Left). Therefore, the first three steps of iterations '1' and '2' are executed sequentially which is quite inefficient. On the contrary, when software pipelining is applied, the data-independent instructions are interleaved (Fig. 8 - Right), which increases the available instruction-level parallelism. Moreover, the memory accesses can be performed in advance, so the memory latency is further diminished.

C. Optimizing the Ethash algorithm

We applied data prefetching and software pipelining on the Ethash algorithm. Figures 9 and 10 illustrate the code snippets of the unmodified and optimized memory-hard loops of Ethash, respectively.

Initially, to optimize the memory-hard loop phase of Ethash algorithm, we prefetch the *scratch_pad*. The unmodified algorithm randomly fetches 128-bytes segments. Therefore, we moved the fetch operation just after determining the offset in order to hide the memory access latency. Similarly, the software pipelining focuses on the *scratch_pad* execution. As the *_PARALLEL_HASH* "parallel" attempts are actually sequential, pipelining those attempts will increase the efficiency of the workload. In addition, the two inner loops (Lines 4 and 5; the outer for computing the offset and the inner for fetching and computing the *mix* value) were merged in order to be pipelined.

```
for (uint32_t a = 0; a < ACCESSES; a += 4) {
  int t = bfe(a, 2u, 3u);
  for (uint32_t b = 0; b < 4; b++) {
    for (int p = 0; p < _PARALLEL_HASH; p++) {
      // Calculate fetching offset from DAG
      // fnv is a non-cryptographic hash function
      offset[p] = fnv(init0[p] ^ (a + b), ((uint32_t *)&mix[p])[b]) % d_dag_size;
      // Synchronize offset among threads
      // in each warp
      offset[p] = __shfl_sync(0xFFFFFFFF, offset[p], t, THREADS_PER_HASH);
    }
    #pragma unroll
    // Fetch the slice from the DAG scratch_pad
    // Compute the final mix value
    for (int p = 0; p < _PARALLEL_HASH; p++) {
      mix[p] = fnv4(mix[p], d_dag[offset[p]].uint4s[thread_id]);
    }
  }
}
```

Figure 9: The unmodified Ethash memory-hard loop phase.

```
for (uint32_t a = 0; a < ACCESSES; a += 4) {
  int t = bfe(a, 2u, 3u);
  for (uint32_t b = 0; b < 4; b++) {
    uint4 dag_val[_PARALLEL_HASH];
    // Apply software pipelining
    offset[0] = fnv(init0[0] ^ (a + b), ((uint32_t *)&mix[0])[b]) % d_dag_size;
    offset[0] = __shfl_sync(0xFFFFFFFF, offset[0], t, THREADS_PER_HASH);
    dag_val[0] = __ldg( &(d_dag[offset[0]].uint4s[thread_id]) );
    offset[1] = fnv(init0[0] ^ (a + b), ((uint32_t *)&mix[0])[b]) % d_dag_size;
    offset[1] = __shfl_sync(0xFFFFFFFF, offset[1], t, THREADS_PER_HASH);

    #pragma unroll
    for (int p = 0; p < (_PARALLEL_HASH-2); p+=1) {
      mix[p] = fnv4(mix[p], dag_val[p]);
      // Apply software prefetching and pipelining
      dag_val[p+1] = __ldg( &(d_dag[offset[p+1]].uint4s[thread_id]) );
      offset[p+2] = fnv(init0[p+2] ^ (a + b), ((uint32_t *)&mix[p+2])[b]) % d_dag_size;
      offset[p+2] = __shfl_sync(0xFFFFFFFF, offset[p+2], t, THREADS_PER_HASH);
    }
    // Apply software pipelining
    mix[_PARALLEL_HASH-2] = fnv4(mix[_PARALLEL_HASH-2], dag_val[_PARALLEL_HASH-2]);
    dag_val[_PARALLEL_HASH-1] = __ldg( &(d_dag[offset[_PARALLEL_HASH-1]].uint4s[thread_id]) );
    mix[_PARALLEL_HASH-1] = fnv4(mix[_PARALLEL_HASH-1], dag_val[_PARALLEL_HASH-1]);
  }
}
```

Figure 10: The optimized Ethash memory-hard loop implementation.

D. Optimized Ethash evaluation

We have implemented the following *Ethminer* versions: (1) *Ethminer* with data prefetching, (2) *Ethminer* with software pipelining, and (3) *Ethminer* with both data prefetching and software pipelining. Note that, all optimized versions have been verified against the test suite provided by Ethash, with all tests successfully completed. Finally, we compare our optimized implementations against both the unmodified *Ethminer* implementation and the high-performing closed-source *Claymore* miner [41].

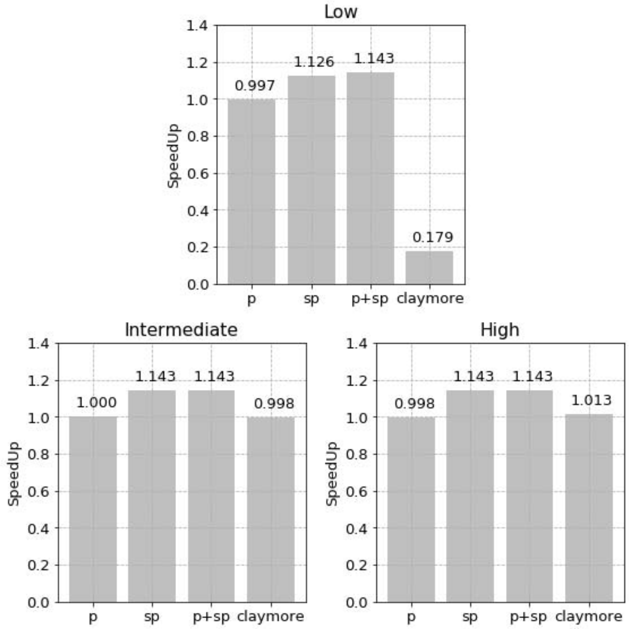

Figure 11: The measured performance speedup of Ethminer with data prefetching (‘*p*’ notation), with software pipelining (‘*sp*’ notation), with both optimization techniques and Claymore’s. Note that, the values are normalized against the unmodified version of Ethminer.

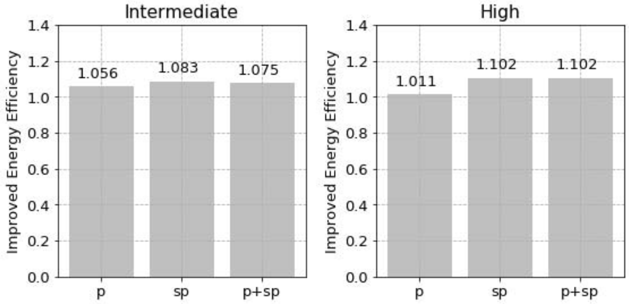

Figure 12: The calculated energy efficiency of Ethminer algorithm when integrating data prefetching (‘*p*’ notation), software pipelining (‘*sp*’ notation), and both optimization techniques. Note that, the values are normalized against the unmodified version of Ethminer.

To measure the performance speedup (in Hashrate) achieved through our optimization strategy, we use the following formula:

$$Speedup = \frac{Hashrate_{new}}{Hashrate_{original}}$$

where $Hashrate_{new}$ is the hashrate of the optimized Ethash implementation, while $Hashrate_{original}$ denotes the unmodified version of the algorithm.

According to Figure 11, when both data prefetching and software pipelining are integrated into *Ethminer’s* source code, the speedup is more than 14%, with this trend persisting across all computing systems. Additionally, the highest performance increase is measured with the software pipelining technique. This is attributed to the fact that the memory segment usage on a specific iteration of the memory-hard loop is efficiently interleaved with data-independent operations. From this, we manage to exploit more efficiently the available instruction-level parallelism. On the contrary, solely the data prefetching technique has neutral impact on the performance. Intuitively, this behaviour is expected since the Ethash algorithm and, in PoW algorithms in general, were designed to be memory inefficient. As a result, any attempt to predict the next memory location accessed by the algorithm is highly difficult.

Interestingly, *Claymore's* performance on the "low" computing system was significantly lower than Ethminer's. In particular, it achieves only 18% of the unmodified Ethminer's performance, while performed similarly on the "intermediate" computing system and slightly outperformed it on the "high" performance machine by 0.1%. Since the *Claymore* miner is a commercial software, the implementation details are undisclosed. Therefore, we can only speculate that *Claymore* favors only a specific type of GPUs; for example, AMD GPUs[1].

Apart from measuring the performance, we have calculated the energy efficiency (in Hash/Joule) of Ethminer using the following formula:

$$Eff = \frac{H}{E}$$

where the energy efficiency (Eff) is proportional to the number of hashes H generated with an amount of energy E. Moreover, having the $Hashrate$ be equal to H/s and the Power P to E / $Time$ (Watts), Eff is transformed to:

$$Eff = \frac{H}{E} = \frac{H/Time}{E/Time} = \frac{Hashrate}{P}$$

Therefore, to evaluate the energy efficiency improvements of the optimized version of Ethash algorithm, the following fraction is used:

$$Improvement_{Eff} = \frac{Eff_{new}}{Eff_{original}}$$

where Eff_{new} is the energy efficiency of the optimized Ethash implementation, while $Eff_{original}$ denotes the unmodified version of the algorithm.

Our experimental results show that the energy efficiency of Ethminer has been increased up to 10.2% on the "high" performing computing system (Figure 12). This is attributed to the fact that the optimization strategy focuses on increasing the utilization of the cache memories, which is highly more efficient than fetching data from the main memory. Overall, the proposed optimization strategy improves the performance and energy efficiency of Ethminer, outperforming Claymore; the fastest commercial implementation of Ethash.

VI. Related Work

The related work on Cryptomining algorithms mainly focuses on analyzing the algorithmic features of the PoW consensus algorithms, rather than presenting the implementation details and assessing their performance implications. In particular, [20], [42] conducted algorithmic-level analysis on numerous memory-hard PoW functions. Moreover, [19] proposed the idea of combating spam emails by using memory-hard algorithms. Additionally, [26], [43]–[46] presented various memory-hard PoW algorithms with provable security and memory-hardness. This paper formalizes the analysis of three state-of-the-art memory-hard PoW algorithms and performs a detailed performance characterization. Then, motivated by our findings, we propose an optimization strategy to accelerate the performance and increase the energy efficiency of the most popular memory-hard PoW algorithm.

VII. Conclusions and Future Work

Blockchain is regarded to be among the next major disruptive technologies with Bitcoin representing its most popular application. The majority of blockchains utilize Proof-of-Work algorithms to convert a consensus problem to a competition based on cryptographic hash functions. Although the proliferation of application-specific hardware has enabled the exponential growth of the hashing power, it has also resulted in the centralization of mining process. As a result, consensus algorithms transformed from computational-intensive to memory-hard algorithms with the aim to keep commodity hardware competitive. This transition, however, led to increased energy costs for public blockchains that consume tremendous amounts of power to maintain their operation.

In this paper we perform a detailed characterization of the three most popular memory-hard PoW algorithms, the Ethash, CryptoNight and Scrypt on Nvidia GPUs. Motivated by this analysis, we exploit data prefetching and software pipelining to leverage the memory-hardness. As our experimental results demonstrate, we obtain more than 14% performance speedup and improve energy efficiency up to 10% compared to the original Ethash algorithm. Finally, the proposed optimized Ethash algorithm outperforms, Claymore, the fastest commercial implementation of Ethash.

Our future plan is to analyze various cryptomining algorithms, such as Proof-of-Stake and dBFT, and evaluate them on a wide range of hardware configurations including AMD GPUs and 3D stacked architectures.

VIII. Acknowledgments

We thank Dr. Foivos S. Zakkak and Dr. Katerina Doka for their valuable feedback. This work is partially supported by the EU Horizon 2020 E2Data 780245, ACTiCLOUD 732366, and EuroEXA 754337 grants.

[1] https://github.com/ethereum-mining/ethminer/issues/869

References

[1] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Consensus in the age of blockchains," *CoRR*, vol. abs/1711.03936, 2017. [Online]. Available: http://arxiv.org/abs/1711.03936

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf."

[3] "State of blockchain q1 2016: Blockchain funding overtakes bitcoin." [Online]. Available: http://www.coindesk.com/state-of-blockchain-q1-2016/

[4] K. Okupski, "Bitcoin developer reference," *Eindhoven*, 2014.

[5] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," *Commun. ACM*, vol. 61, no. 7, pp. 95–102, Jun. 2018. [Online]. Available: http://doi.acm.org/10.1145/3212998

[6] A. Hern, "Bitcoin's energy usage is huge we can't afford to ignore it," Jan 2018. [Online]. Available: https://www.theguardian.com/technology/2018/jan/17/bitcoin-electricity-usage-huge-climate-cryptocurrency

[7] S. King and S. M. Nadal, "Ppcoin : Peer-to-peer crypto-currency with proof-of-stake," 2012.

[8] D. Larimer, "Delegated proof-of-stake (dpos)," *Bitshare whitepaper*, 2014.

[9] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[10] "Bitcoin v0.1 relased." [Online]. Available: http://www.metzdowd.com/pipermail/cryptography/2009-January/014994.html

[11] Diablo-D3, "Diablo-d3/diablominer," Jul 2017. [Online]. Available: https://github.com/Diablo-D3/DiabloMiner

[12] "Open-source fpga bitcoin miner," 2013. [Online]. Available: https://github.com/progranism/Open-Source-FPGA-Bitcoin-Miner

[13] B. Labs, "Final rendering of jalapeno for production (which is under way). pic.twitter.com/rks5m49f," Sep 2012. [Online]. Available: https://twitter.com/ButterflyLabs/status/250958891544895488

[14] "Asicminer." [Online]. Available: https://asicminer.co

[15] "Canaan." [Online]. Available: https://canaan.io/

[16] "Bitfury." [Online]. Available: https://bitfury.com/

[17] "Bitmain." [Online]. Available: https://www.bitmain.com/

[18] M. Swan, *Blockchain: Blueprint for a New Economy*, 1st ed. O'Reilly Media, Inc., 2015.

[19] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, "Moderately hard, memory-bound functions," *ACM Trans. Internet Technol.*, vol. 5, no. 2, pp. 299–327, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1064340.1064341

[20] A. Biryukov and D. Khovratovich, "Tradeoff cryptanalysis of memory-hard functions," in *Proceedings, Part II, of the 21st International Conference on Advances in Cryptology — ASIACRYPT 2015 - Volume 9453*. Berlin, Heidelberg: Springer-Verlag, 2015, pp. 633–657. [Online]. Available: https://doi.org/10.1007/978-3-662-48800-3_26

[21] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 3–16. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978341

[22] C. Natoli and V. Gramoli, "The balance attack or why forkable blockchains are ill-suited for consortium," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017, pp. 579–590.

[23] P. McCorry, S. F. Shahandashti, and F. Hao, "Refund attacks on bitcoins payment protocol," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 581–599.

[24] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *International conference on financial cryptography and data security*. Springer, 2014, pp. 436–454.

[25] A. Sapirshtein, Y. Sompolinsky, and A. Zohar, "Optimal selfish mining strategies in bitcoin," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 515–532.

[26] C. Percival and S. Josefsson, "The scrypt password-based key derivation function," Tech. Rep., 2016.

[27] N. Van Saberhagen, "Cryptonote v 2.0," 2013. [Online]. Available: https://cryptonote.org/whitepaper.pdf

[28] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.

[29] E. Duffield and D. Diaz, "Dash: A privacy-centric crypto-currency," 2014.

[30] A. Biryukov and D. Khovratovich, "Equihash: Asymmetric proof-of-work based on the generalized birthday problem," *Ledger Journal*, vol. 2, pp. 1–30, 2017.

[31] Litecoin-project, "Litecoin," 2011. [Online]. Available: https://github.com/litecoin-project/litecoin

[32] "Geforce gtx 960m graphics card." [Online]. Available: https://www.geforce.com/hardware/notebook-gpus/geforce-gtx-960m

[33] "Nvidia titan xp graphics card." [Online]. Available: https://www.nvidia.co.uk/titan/titan-xp

[34] "Nvidia quadro professional graphics." [Online]. Available: https://www.nvidia.com/en-gb/design-visualization/quadro-store

[35] Tpruvot, "Ccminer," Jun 2018. [Online]. Available: https://github.com/tpruvot/ccminer

[36] "Monero cryptocurrencies." [Online]. Available: https://ww.getmonero.org

[37] "Scrypt." [Online]. Available: https://cryptorival.com/algorithms/scrypt/

[38] "ethereum-mining." [Online]. Available: https://github.com/ethereum-mining/ethminer/blob/master/ethminer/main.cpp

[39] "ccminer." [Online]. Available: https://github.com/tpruvot/ccminer/blob/windows/ccminer.cpp

[40] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, Jun. 2000. [Online]. Available: http://doi.acm.org/10.1145/358923.358939

[41] Nanopool, "Claymore-dual-miner," 2018. [Online]. Available: https://github.com/nanopool/Claymore-Dual-Miner

[42] J. Alwen, P. Gazi, C. Kamath, K. Klein, G. Osang, K. Pietrzak, L. Reyzin, M. Rolinek, and M. Rybar, "On the memory-hardness of data-independent password-hashing functions," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS '18. New York, NY, USA: ACM, 2018, pp. 51–65. [Online]. Available: http://doi.acm.org/10.1145/3196494.3196534

[43] J. Alwen and V. Serbinenko, "High parallel complexity graphs and memory-hard functions," in *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, ser. STOC '15. New York, NY, USA: ACM, 2015, pp. 595–603. [Online]. Available: http://doi.acm.org/10.1145/2746539.2746622

[44] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: New generation of memory-hard functions for password hashing and other applications," in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, March 2016, pp. 292–302.

[45] J. Alwen, J. Blocki, and B. Harsha, "Practical graphs for optimal side-channel resistant memory-hard functions," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 1001–1017. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134031

[46] J. Alwen and J. Blocki, "Efficiently computing data-independent memory-hard functions," in *Proceedings, Part II, of the 36th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016 - Volume 9815*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 241–271. [Online]. Available: https://doi.org/10.1007/978-3-662-53008-5_9