Irregular-Program-Based Hash Algorithms

Qi Zhou
QuarkChain Foundation LTD.
Singapore, Singapore

qizhou@quarkchain.org

***Abstract*—Because of their energy efficiency over general-purpose central processing unit (CPU), application-specific integrated circuit (ASIC) and field-programmable gate array (FPGA) have been used to accelerate blockchain mining, which creates the concern of centralization because the ASIC devices are dominated by a few manufacturers. Several algorithms are proposed to lower the benefits of ASIC including memory-intensive hash algorithms, where the memory access patterns are random and depend on input data. To further lower the potential hardware acceleration, in this paper, we introduce irregular-program-based hash algorithms, where both the code path and memory access are random and depend on input data. We present an example of the hash algorithm based on dynamic search tree (DST), which exhibits both control-path and memory-access irregularities. We compare the performance of a CPU implementation of the DST to an existing FPGA implementation, which shows comparable performance. An instance of the proposed hash algorithm is elaborated, and the test inputs and outputs are given.**

***Index Terms*—hash algorithm, application-specific integrated circuit (ASIC), irregular program, dynamic search tree.**

I. Introduction

The interest of using hardware accelerators for blockchain mining such as application-specific integrated circuit (ASIC) and field-programmable gate array (FPGA) has been popular because of their huge energy efficiency over general-purpose central processing unit (CPU) mining. For example, the mining of Bitcoin is dominated by ASIC miners because the ASIC devices could outperform CPU substantially because of the parallelization-friendly structure of the Bitcoin's double SHA2 hash algorithm. This creates the concern of centralization because the manufacturers of the high-performance ASIC mining devices are dominated by a few ones. To reduce the centralization concern, several algorithms are proposed to lower the efficiency of ASIC mining including Equihash [1], CyptoNight [2], and Ethash [3] by performing memory-intensive operations. However, some ASIC accelerators have been developed for both Equihash and CryptoNight and achieve various accelerations (see [5] for a summary).

The effectiveness of hardware accelerators highly depends on how the computations of a hash algorithm can be efficiently parallelized by either pipeline (in ASIC/FPGA) or single instruction multi-threads (in graphics processing units (GPUs)). If the hash algorithm is regular in the sense that the control flow and memory access patterns are predictable, hardware accelerators could achieve significant performance gain (in terms of performance per transistor or energy) over CPU. In contrast, the operations (e.g., read/write) of irregular data structures such as trees and graphics are much harder to parallelize because the code path and/or memory access patterns are input-dependent and difficult to predict [4].

There are two types of irregularities: 1, memory-access irregularity, where the memory access patterns are data dependent; and 2, control-flow irregularity, where the code paths are data dependent. Ethash exhibits some levels of memory-access irregularity, however, the instructions that Ethash executes are independent of data and are predictable. As a result, Ethash could still be efficiently parallelized by GPUs.

To further lower the advantages of hardware accelerators, in this paper, we propose a novel hash algorithm based on irregular programs that exploit both memory-access irregularity and control-flow irregularity. The hash algorithm is based on the order statistics of a dynamic list, whose efficient implementation is a dynamic search tree (DST). Since the code path executed by inserting/deleting the ith smallest element of a DST completely depends on i, the control flow and the memory access patterns of the proposed algorithm are random and unpredictable.

In addition, to investigate the potential performance gain of the proposed algorithm by hardware accelerators, we compare the performance of a CPU implementation of the DST to an existing FPGA implementation, which shows comparable performance between two implementations.

Moreover, we present an instance of the proposed DST-based hash algorithm, and list some test vectors and their outputs.

The rest of the paper is organized as follows. Section II introduces Ethash algorithm. Section III discusses our proposed hash algorithm based on DST. Section IV illustrates the comparison between CPU and FPGA implementations of DST. Section V elaborates an example of the DST-based hash algorithm and provides several test vectors for verification. Section VI concludes the paper.

II. Overview of Ethash Algorithm

Let us first introduce Ethash algorithm, which calculates a hash value by performing memory-intensive operations. The core part of Ethash can be illustrated as follows (Note that for simplicity, we ignore the pre-progressing and post-processing parts of Ethash).

To be published in IEEE International Conference on Decentralized Applications and Infrastructures (IEEE DAPPCON 2019).

Input:

- s is an input state with N bytes; and
- $\mathcal{L}$ is an array of pre-generated random data with each data being M bytes and $|\mathcal{L}| = L$; and
- R is the number of hash rounds.

Body:

```
For $i$ := 1 to $R$ do
1) $s, p = G_0(s)$
2) $p = \text{Int}(p) \bmod |\mathcal{L}|$
3) $v =$ the $p$th element in $\mathcal{L}$
4) $s = M_0(s, v)$
```

Output: s

where

- s is the input state after pre-processing the input data (output from SHA3_512 algorithm); and
- $s', p = G_0(s)$ is a fast pseudo random number generator (RNG), s is the input state of the RNG, s' is the output state, and p is a newly generated random number; and
- $s' = M_0(s, v)$ is a merging function that merges the input data v with the state s, and outputs s' as the new state; and
- state s has $N = 128$ bytes; and
- the number of hash rounds $R = 64$, and for each round, the algorithm reads $M = 128$ bytes data from $\mathcal{L}$ and merges it with current state; and
- the initial size of $\mathcal{L}$ is 1 gigabyte (GB) with $L = 8 \times 1024 \times 1024$ elements.

Since G_0 and M_0 take minimal arithmetic operations, the main cost of the algorithm is to randomly read the data from $\mathcal{L}$. As $\mathcal{L}$ is quite large (initial size is 1GB), storing $\mathcal{L}$ in registers or cache is cost-prohibited, and thus $\mathcal{L}$ has to be stored in the main memory. As the result, the algorithm's performance is determined by the memory latency/bandwidth, which is hard to be improved by ASIC. In addition, to make the memory access pattern harder to predict and optimize, $\mathcal{L}$ will be changed for every 30000 blocks and its size grows linearly as block height increases.

III. Proposed Irregular-program-based Hash Algorithm

Although Ethash has random memory access patterns and exhibits some levels of irregularity, it reads a fixed-size (128-bytes) data from main memory, which can be done in a single memory block read in GPU. In addition, the control flow of Ethash is deterministic. As a result, Ethash can be easily parallelized and accelerated by GPU, which generally has higher memory bandwidth than that of CPU and could run multiple instances of Ethash concurrently before the memory bandwidth is saturated. To further limit the gain of GPU acceleration, in this section, we propose an irregular-program-based hash algorithm, which employs both memory-access irregularity and control-flow irregularity (Again, pre-processing and post-processing parts are omitted for simplicity).

Input:

- s is an input state with N bytes; and
- $\mathcal{L}$ is a set of pre-generated random data with each data being M bytes and $|\mathcal{L}| = L$; and
- R is the number of hash rounds.

Body:

```
For $i$ := 1 to $R$ do
1) $s, p = G_1(s)$
2) $p = \text{Int}(p) \bmod |\mathcal{L}|$
3) $v =$ the $p$th smallest element in $\mathcal{L}$
4) $\mathcal{L} = \mathcal{L}/\{v\}$
5) $s = M_1(s, v)$
6) $s, d = G_1(s)$
7) $\mathcal{L} = \mathcal{L} \cup \{d\}$
```

Output: s

where

- s is the input state after pre-processing the input data (output from SHA3_512 algorithm); and
- $s', p = G_1(s)$ is a fast pseudo random number generator, s is the input state, s' is the output state, and p is a newly generated random data with M bytes; and
- $s' = M_1(s, v)$ is a merging function that merges the input data v with the state s, and outputs s' as the new state.

Compared to Ethash, the proposed hash algorithm has several major differences:

- Instead of looking for the data with a random index p in Ethash, the proposed algorithm looks for the data with the pth smallest value.
- After reading one element in $\mathcal{L}$, the element will be removed and a new element with random value is inserted into $\mathcal{L}$, while Ethash only reads the data from $\mathcal{L}$.

Therefore, $\mathcal{L}$ is essentially a dynamic list with order statistic lookup/removal. There are a couple of implementations of $\mathcal{L}$:

- A unordered array with average $O(L)$ to find/remove the pth smallest value and average $O(1)$ to insert a new value (the resulting array is still unordered); or
- A sorted array with average $O(\log(L))$ time to find the pth smallest value using binary search, average $O(L)$ time to remove the value, and average $O(L)$ time to insert a value into $\mathcal{L}$; or
- A balanced dynamic search tree (DST) with ordered statistics (e.g., augmented search tree [8] such as AVL tree, red-black tree, or B+ tree) so that the average cost of inserting/removing the pth element is $O(\log(L))$. DST tends to be irregular. The memory access patterns of the tree are highly input-dependent as the operation may traverse completely different tree nodes depend on the input. In addition, given random input p, the control flow path of deleting/inserting the pth element is also different. E.g, consider a balanced tree implementation of the DST, the execution sequences of left and right rotations of an insert/delete operation differ on different p's.

Note that the aforementioned performance analysis is based on the assumption that L and R are sufficiently large. If L or R is small, the DST implementation may not be the optimal. For example, if L is small enough, a sorted array implementation may outperform the DST implementation as moving a small data is cache friendly and can be done more efficiently than the DST operations, which are generally unfriendly to cache because of tree traversal. One way to eliminate such optimization is probably to incorporate DST specific data into the state, e.g., merging the hash of the DST with the state s just before outputting s, or merging the numbers of left rotations and right rotations of a specific balanced tree implementation with the state s. This will enforce the use of a specific implementation of the DST in the proposed hash algorithm.

The size of $\mathcal{L}$ can be further tunable in the favor of CPUs. If the memory size of the DST implementation of $\mathcal{L}$ is smaller than 2MB, the DST could be fitted into the L3 cache of most modern CPUs, where the L3 cache of CPU is faster than GDDR5 of GPU and a close of 2MB memory is unlikely to be accepted for the modern ASICs [2]. To make control flow and memory access more irregular over time, similar to Ethash, $\mathcal{L}$ could be changed for every several blocks, and its size may grow linearly as block height increases.

IV. Performance Comparison of CPU and FPGA Implementations of DST

In this section, we compare the performance of CPU implementation of DST to that of an existing FPGA implementation in [6]. The code of the CPU version can be found here [7]. For the CPU implementation, we use the following configurations

- CPU: Intel i7-7700K
- OS: Ubuntu 16.04 LTS
- Compiler: g++ 5.4.0
- Compilation command: g++ -O3 -std=gnu++17
- Number of threads: 1
- Number of keys: 64K
- Key type: unsigned 64-bit random integers

The performance numbers are:

- FPGA: 3.97 million inserts/deletes per sec
- CPU: 4.46 million inserts/deletes per sec

where the CPU single core performance is slightly better than that of FPGA.

A couple of comments are of interests:

- Compared to the search performance of FPGA implementation (242 million DST searches per sec [6]), where every search can be done in one clock cycle because of pipeline, the performance of insert/delete operations are much lower because more cycles are required for per insert/delete operations.
- In addition, the FPGA performance is based on Virtex 5 LX330 FPGA, which may be out-dated. With the latest FPGA, the FPGA performance could be greater.
- The performance of the CPU implementation is single thread/core, which could be greater if multiple threads/cores are used.
- The key size of the CPU performance is 64-bit, which is larger than 32-bit of the FPGA version.

V. Example of Irregular-Program-Based Hash Algorithm

In this section, we present an example of irregular-program-based hash algorithm. We will also provide a few test vectors to help verifications. Note that, the code consists of necessary pre-processing and post-processing steps to help reproduction, and takes arbitrary bytes as input.

Input:

- d is an array of bytes; and
- $\mathcal{L}$ is a set of pre-generated random data with each data being 64-bit unsigned integers.

Body:

```
$h = \texttt{SHA3\_512}(d)$
$s = h[:]$
$s.\texttt{extend}(h)$
For $i$ := 0 to 63 do
1) $p = \texttt{FNV}(i \oplus h[0], s[i \mod \texttt{len}(s)])$
2) For $j$ := 0 to 15 do
    a) $v = \mathcal{L}.\texttt{erase\_by\_order}(p \mod |L|)$
    b) $d = \texttt{FNV}(p, v)$
    c) $\mathcal{L} = \mathcal{L} \cup \{d\}$
    d) $p = \texttt{FNV}(d, v)$
    e) $s[j] = \texttt{FNV}(s[j], v)$
$r = []$
For $i$ := 0 to 3 do
1) $j = i \times 4$
2) $r.\texttt{add}(\texttt{FNV}(\texttt{FNV}(\texttt{FNV}(s[j], s[j+1]), s[j+2]), s[j+3]))$
```

Output: r

where

- $\texttt{SHA3\_512}(\cdot)$ takes an array of bytes as input, performs SHA3_512 algorithm, and outputs the hash value as an array of 8 64-bit unsigned integers using little-endian encoding; and
- $\texttt{FNV}(a, b) = (a \oplus b) \times 0x100000001b3$ is the 64-bit Fowler-Noll-Vo hash function, where $\oplus$ is bit-wise XOR operator; and
- $\mathcal{L}.\texttt{erase\_by\_order}(p)$ removes the pth smallest element in $\mathcal{L}$ and returns the value of the element; and
- p, v, d are 64-bit unsigned integers, and s is an array of 16 64-bit unsigned integers with $s[j]$ being the jth integer (0-based indexing); and
- the output r is an array of 4 64-bit unsigned integers.

A. Generation of Random Dataset $\mathcal{L}$

We use the following program to generate the random dataset $\mathcal{L}$:

Input:

1) $seed$ is an array of bytes; and
2) L is the number of expected elements in $\mathcal{L}$ and should be a multiple of 8.

Body:

```
For $i$ := 0 to $(L/8) - 1$ do
1) $h = \texttt{SHA3\_512}(seed + \texttt{htobe}(i))$
2) For $j$ := 0 to 7 do
```

a) $\mathcal{L} = \mathcal{L} \cup h[j]$

Output: $\mathcal{L}$

where `htobe`$(\cdot)$ converts an integer to a big-endianness 4-bytes array, and $+$ is an array concatenation operator if both sides are arrays.

Note that $|\mathcal{L}|$ may be smaller than L if a generated random number is duplicated in $\mathcal{L}$, which should be extremely rare for uniformly distributed 64-bit unsigned integer.

B. Test Vectors

In this subsection, we give several hash values of some test inputs. $\mathcal{L}$ is generated with $seed$ being empty array and $L = 65536$. The DST implementation of $\mathcal{L}$ uses left-lean red-black (LLRB) trees in [9].

Input data: “” (empty data)

Output vector:

$$r = \begin{bmatrix} 11967621512234744254 \\ 11712119753881699857 \\ 4190255959603841725 \\ 6654395615551794006 \end{bmatrix} \tag{1}$$

The numbers of left and right rotations of the LLRB tree are 9573 and 9388, respectively.

Input data: “Hello World!”

Output vector:

$$r = \begin{bmatrix} 12754842531904701011 \\ 8384861435613290118 \\ 2739024099562295228 \\ 4448910328080420635 \end{bmatrix} \tag{2}$$

The numbers of left and right rotations of the LLRB tree are 9523 and 9335, respectively.

VI. Conclusion

In this paper, we proposed a hash program based on irregular data structure, or more preciously dynamic search tree. Unlike Ethash, which has only memory-access irregularity, the proposed algorithm exhibits control-flow irregularity and memory-access irregularity. Moreover, we compared the performance of a CPU implementation to that of FPGA one, and we found comparable performance between two implementations. Finally, we presented an example of the implementation and gave several test inputs and outputs of the hash algorithm.

Given the DST is just one of the irregular data structures, we believe that more irregular data structures (such as Graph) can be exploited to construct ASIC-, FPGA-, or even GPU-resistant hash algorithms.

References

[1] A. Biryukov and D. Khovratovich, “Equihash: Asymmetric Proof-of-Work Based on the Generalized Birthday Problem,” Network and Dist. System Security Symp., Ledger 2, pp. 1-30, 2016.

[2] CryptoNight, https://en.bitcoin.it/wiki/CryptoNight

[3] Ethash, https://github.com/ethereum/wiki/wiki/Ethash

[4] M. Burtscher, R. Nasre, K. Pingali, “A Quantitative Study of Irregular Programs on GPUs,”, IEEE Inter. Symp. on Workload Char., pp. 141-151, 2012.

[5] ProgPOW, https://github.com/ifdefelse/ProgPOW

[6] Y.-H. E. Yang, V. K. Prasanna, “High Throughput and Large Capacity Pipelined Dynamic Search Tree on FPGA,”, Proc. of the 18th Annual ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA), 2010, pp. 8392.

[7] Set Benchmark, Link to be disclosed in official paper.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms. MIT press, 2009.

[9] R. Sedgewick, Left-leaning red-black trees, https://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf