## Understand TrueChain Main Net – Resisting ASIC

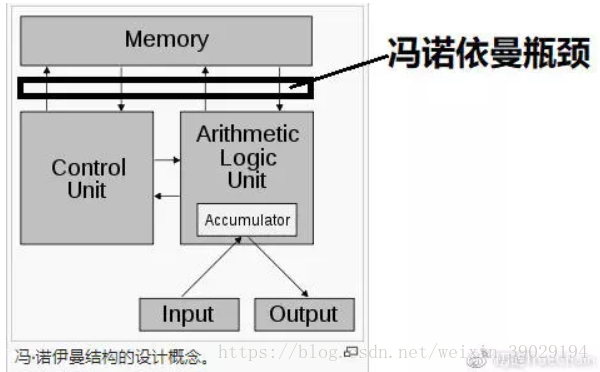
### Part One. How does ASIC avoid the Von Neumann bottleneck?

Prior to introducing the algorithm of TrueHash, we’re going to introduce what the AISC is.

Firstly, let’s think about the mining machine’s development path, and then we can easily find out the path below:

**CPU->GPU->FPGA->ASIC**

Then, we must be wondering why the ASIC defeated the CPU/GPU and become the final architecture of mining machine. The major reason is due to the Von Neumann Architecture which the CPU/GPU used to improve their variety of calculation.



**Von Neumann Bottleneck**

The Von Neumann Architecture goes with three parts, they’re Memory-Unit, Control-Unit and Compute-Unit successively. The program and its data are all written in the Memory-Unit and will be transferred to the Compute-Unit while the calculation is running. The transfer process mentioned before will influence the efficiency of computing which brings us a concept - The Von Neumann Bottleneck.

We can treat an integrated-circuit as a combination of several field-effect transistors. The energy consumption of CPU is related to the number of field-effect transistors involved in the work, and is positively related to the frequency. When an instruction is executed in the CPU, whether it needs to schedule the calculator, access the external memory, or write back, there will be a difference in the number of transfer transistors in L1. We can’t miss a vital indicator in mining efficiency, which is the Hash per Watt.

Some instructions with no relations about mining in CPU, such as Fetch and Decode, have a big part in calculating, but the instructions for running Hash process only hold 10 percent or less in calculating. There’s why for completing the same task, the ASIC would consume much less energy than CPU: The ASIC won’t execute instructions with no relations about mining and will write the Hash program directly in the Computing-Unit for only calculating the Hash. Just use all the advantages to do one thing well, that’s why the ASIC counts.

### Part Two. The meaning of resisting ASIC

The ASIC has a big power, but why do we need to resist it?

We can find the reason in the White Paper of Bitcoin, which is located in the first paragraph of the sixth chapter below:

“By convention, the first transaction in a block is a special transaction that starts a new coin owned by the creator of the block. This adds an incentive for nodes to support the network, and provides a way to initially distribute coins into circulation, since there is no central authority to issue them. The steady addition of a constant of amount of new coins is analogous to gold miners expending resources to add gold to circulation. In our case, it is CPU time and electricity that is expended.”

The essence of the mining process has already been brought in the statement above: “a way to initially distribute coins into circulation”. That turned out a most important concept: mining is a process to fairly distribute coins.

Finally, we can formally dive into TrueChain’s mining algorithm -- **TrueHash**.

### Part Three. The meaning of resisting ASIC

After BTC’s unveiling, a number of BlockChain developers always keep trying to find their anti-ASIC mining algorithm, but basically, all of them ends with failure. In the new generation of public chains, the TrueChain’s research team, as the defender and saver of PoW, has created a mining algorithm that is essentially anti-ASIC and randomly changes itself. That makes any of the AISC machines unable to avoid the Von Neumann Bottleneck, which is good news to protect general miners’ benefits.

To resist ASIC fundamentally, the algorithm needs to have these three properties below:

1. Let S be a set of algorithms, all algorithms that implement S can not bypass the Von Neumann bottleneck.
2. For each T block switching algorithm, the switching mode must satisfy verifiability and randomness.
3. Algorithm switching does not depend on manual adjustment.

In general mining algorithm, after padding and other calculating, the blockhead, nonce and other information will form a vector V (nonce).

Let’s look at **TrueHash** algorithm. In the first, it will detect the dataset state.

go

func (m \*Minerva) CheckDataSetState(blockNum uint64) bool{

dataset := m.dataset

if dataset.dateInit == 0{

//If blockNum(the amount of block) is less than 12000

if blockNum <= UPDATABLOCKLENGTH{

//Initialize truehashTable

m.truehashTableInit(dataset.evenDataset)

dataset.dataset = &dataset.evenDataset

}else{

//Key point here，the dataset is about to exchange

bn := (blockNum / UPDATABLOCKLENGTH -1 ) \* UPDATABLOCKLENGTH + STARTUPDATENUM + 1

in := (blockNum / UPDATABLOCKLENGTH) % 2

//If blockNum > UPDATABLOCKLENGTH then change lookutable to generate the even or odd dataset

if in == 0{

//Configure even dataset

dataset.dataset = &dataset.evenDataset

//Initialize flags

dataset.oddFlag = 0

dataset.evenFlag = 0

}else{

//Configure odd dataset

dataset.dataset = &dataset.oddDataset

dataset.oddFlag = 0

dataset.evenFlag = 0

}

//Update LookupTBL

m.updateLookupTBL( bn, \*dataset.dataset)

}

dataset.dateInit = 1

}

if blockNum %UPDATABLOCKLENGTH >= STARTUPDATENUM {

//Start to update lookuptable

in := (blockNum / UPDATABLOCKLENGTH) % 2

//Change lookutable between even or odd

if in == 0{

//Set dataset to odd if it’s currently even

if dataset.oddFlag == 0 {

//Update LookupTBL

res := m.updateLookupTBL(blockNum, dataset.oddDataset[:])

if res {

//Set the odd flag of dataset to 1 if the update is successful

dataset.oddFlag = 1

}else{

//Fail to update

return false

}

}

}else{

//Set dataset to even if it’s currently odd

if dataset.evenFlag == 0 {

res := m.updateLookupTBL(blockNum, dataset.evenDataset[:])

if res {

dataset.evenFlag = 1

}else{

return false

}

}

}

}

if blockNum %UPDATABLOCKLENGTH == 1{

in := (blockNum / UPDATABLOCKLENGTH) % 2

//Change lookutable to generate dataset

if in == 0{

//Configure even dataset

dataset.dataset = &dataset.evenDataset

dataset.evenFlag = 0

}else{

//Configure odd dataset

dataset.dataset = &dataset.oddDataset

dataset.oddFlag = 0

}

}

return true

}

The function involves dataset which has even version and odd version. The implement of this function is to check some conditions and generate (< 12000) or randomly switch (> 12000) the dataset for exchanging some essential algorithm ingredients which make anti-ASIC available.

The configuration variable UPDATABLOCKLENGTH is set to 12000.

If the block number is less than 12000, then generate and initialize the dataset.

If the block number is greater than 12000, then start to switch the dataset according to some block headers in the BlockChain.

The change of dataset directly effect the **TrueHash** algorithm, so let’s look at one version of the **TrueHash** function blew:

```go

*// Calculate the work of this nonce*

digest, result := truehashFull(\*m.dataset.dataset, hash, nonce)

```

And then, let’s look at the implement of the fchainmining() function which is inside of the **TrueHash** function:

```go

func fchainmining( plookup []uint64, header []byte, nonce uint64) ([]byte, []byte){

var seed [64]byte

output := make([]byte, DGSTSIZE)

val0 := uint32(nonce & 0xFFFFFFFF) //Generate one number from 0 to 4294967295

val1 := uint32(nonce >> 32)

for k:= 3; k >= 0; k-- {

seed[k] = byte(val0) & 0xFF

val0 >>= 8

}

for k := 7; k >= 4; k-- {

seed[k] = byte(val1) & 0xFF

val1 >>= 8

}

dgst := make([]byte, DGSTSIZE)

for k := 0; k < HEADSIZE; k++ {

seed[k+8] = header[k]

}

sha512 := makeHasher(sha3.New512())

var sha512\_out [64]byte

sha512(sha512\_out[:],seed[:])

byteReverse(sha512\_out[:])

var permute\_in [32]uint64

for k := 0; k < 8; k++ {

for x := 0; x < 8; x++ {

var sft int= x \* 8

val := (uint64(sha512\_out[k\*8+x]) << uint(sft))

permute\_in[k] += val

}

}

for k := 1; k < 4; k++ {

for x := 0; x < 8; x++ {

permute\_in[k \* 8 + x] = permute\_in[x]

}

}

scramble(permute\_in[:], plookup[:])

var dat\_in [256]byte

for k := 0; k < 32; k++ {

val := permute\_in[k]

for x := 0; x < 8; x++ {

dat\_in[k \* 8 + x] = byte(val & 0xFF)

val = val >> 8

}

}

for k := 0; k < 64; k++ {

var temp byte

temp = dat\_in[k \* 4];

dat\_in[k \* 4] = dat\_in[k \* 4 + 3];

dat\_in[k \* 4 + 3] = temp;

temp = dat\_in[k \* 4 + 1];

dat\_in[k \* 4 + 1] = dat\_in[k \* 4 + 2];

dat\_in[k \* 4 + 2] = temp;

}

//Generate random hash by sha256

sha256 := makeHasher(sha3.New256())

sha256(output, dat\_in[:])

// reverse byte

for k := 0; k < DGSTSIZE; k++ {

dgst[k] = output[k];

}

return dgst, dgst

}

```

There are many shifting operations and permutation operations in the function. Among them, the encryption algorithm SHA256 and SHA512 are used, and then a random hash is generated by the makeHasher() function.

There is a key function called scramble().

According to all functions demonstrated above, switching dataset and scrambling data that inputs into Hash operation can achieve a change in Hash calculation process. The dataset is extremely complicated, so there is no opportunity to write all versions of it into the Compute-Unit. Due to the random switch of Hash algorithm, Von Neumann Bottleneck is unable to bypass.

Summary of the **Truehash** algorithm switching principle: every 12,000 PoW blocks (complete generating needs about 83 days) changes the dataset once. The new dataset is composed of 1st - 8192th blocks in the previous cycle, which are generated by analyzing the hash values of 1101th - 11256th blocks. Because the hash value of blocks can not be predicted in advance, the information of the new algorithm was unknown before the 11256th block mined. From the 11257th block last week to the abolition of the algorithm, there is only 88 days for the ASIC producer. It is meaningless to produce ASIC in such a short period of time, thus destroying the desire of producing mining machine is possible.

### Part Four. The fruitchain integrated with PoW is to end the age of big miners

In the current mining machine ecosystem, the problems of Selfish Mining Attack and Eclipse Attack need to be avoided in PoW chain. TrueChain’s team tried to replace Nakamoto’s original PoW protocol with fPoW protocol, and implemented the integration of fruitchain into PoW in engineering, which turns PoW to fPoW.

fPoW is a new concept of mining design. Its highlights are as follows:

1. Low-difficulty mining: The difficulty of mining fruit is 1/600 of mining general blocks. Each fruit records some transactions from the PBFT consensus committee. Ordinary mining only needs to verify the transaction information. Therefore, mining can be implemented without adding into mining pool, which avoids the concentration of calculation (forming mining pool).
2. Resisting selfish mining: As long as the fruit keeps fresh and existing, rewards can be distributed to the miner who mined this fruit.
3. Remuneration for fruit mining: When block miners collect fruits into the mined block, they distribute the rewards to the fruit miners in proportion according to their calculation power.

The realization of fruit chain enables ordinary miners to participate in the mining process more fairly and get more fair reward distribution, which makes the TrueChain a decentralized public chain and ends the age of big miners.

References:

[1]: https://blog.csdn.net/qq\_34239128/article/details/82875587?from=timeline&isappinstalled=0

[2]: https://mp.weixin.qq.com/s/g1MzhpXk-M5iNbVikXqiYw

[3]: https://www.jianshu.com/p/4cb87a227d94