Filecoin Proof-of-Replication Encoding Optimization Report

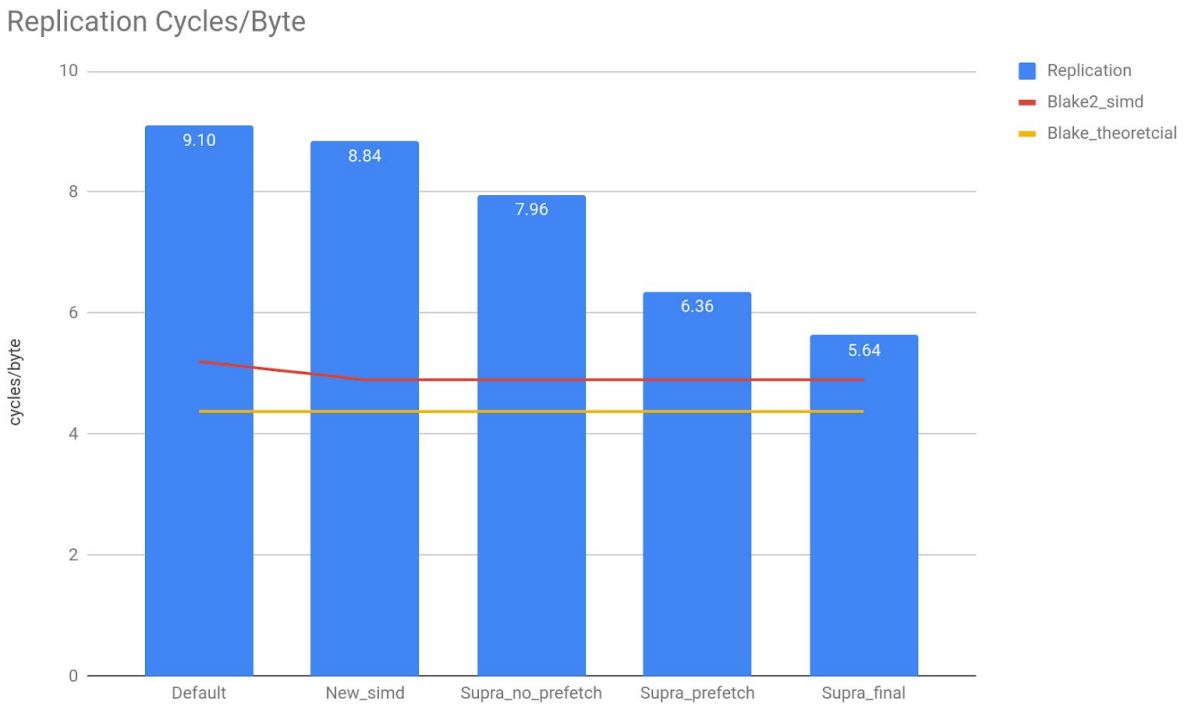
Prepared for: Protocol Labs
By: Supranational
Date: 09/15/2019
Version: 1.0 Final Report[1]

[1] This report makes no statements or warranties and is for discussion purposes only.

Table of Contents

Executive Summary

The encoding within the replication phase of Filecoin's Proof of Replication (PoRep) was analyzed for optimization potential. The foundation for analysis was the "r2" repo created to ideally implement the fastest replication (https://github.com/nicola/r2). After analyzing the bottlenecks we went ahead and implemented some of our proposed improvements. The performance improved 40% by modifying the hash function to avoid memory copies, improving the core Blake2s implementation, and eliminating unnecessary node encoding steps. Our Blake implementation is within 10% of the theoretical maximum on today's CPUs and accounts for about 92% of the replication processing time. Further analysis is required to determine if the conversions to/from field representation and the node encoding can be improved. All improvements can be found on the supra_hash branch of our r2 fork here: https://github.com/sean-sn/r2/tree/supra_hash.



Hash Algorithm Study

To start we explored three hash algorithms for possible usage by Filecoin in porep: Blake2s (currently used), Blake2b, and SHA-256.

Blake2(s/b) (https://tools.ietf.org/html/rfc7693)

	BLAKE2b	BLAKE2s
Bits in word	w = 64	w = 32
Rounds in F	r = 12	r = 10
Block bytes	bb = 128	bb = 64
Hash bytes	1 <= nn <= 64	1 <= nn <= 32
Key bytes	0 <= kk <= 64	0 <= kk <= 32
Input bytes	0 <= ll < 2**128	0 <= ll < 2**64
G Rotation constants =	(R1, R2, R3, R4) (32, 24, 16, 63)	(R1, R2, R3, R4) (16, 12, 8, 7)

Blake2s is currently used in Filecoin since the digest size (Hash bytes) are of size 32 bytes. This is a smaller digest than Blake2b by default which is appealing. However as seen in the table above Blake2b's digest could be truncated to whatever digest size is preferred. Both Blake2s and Blake2b share most of the same structure. The message schedule is the same between the two (note Blake2b has 2 more rounds therefore two more schedule rows). The mixing function G is also the same except for the rotation constants (R1-R4). The compression function which is comprised of the message schedule and mixing function is the same except as noted Blake2b has two more rounds.
As is typical with hashing functions the message schedule is not on the critical path, therefore we focus on the mixing function G.

```
FUNCTION G( v[0..15], a, b, c, d, x, y )
    |
    |   v[a] := (v[a] + v[b] + x) mod 2**w
    |   v[d] := (v[d] ^ v[a]) >>> R1
    |   v[c] := (v[c] + v[d])     mod 2**w
    |   v[b] := (v[b] ^ v[c]) >>> R2
    |   v[a] := (v[a] + v[b] + y) mod 2**w
    |   v[d] := (v[d] ^ v[a]) >>> R3
    |   v[c] := (v[c] + v[d])     mod 2**w
    |   v[b] := (v[b] ^ v[c]) >>> R4
    |
    |   RETURN v[0..15]
    |
```

```
END FUNCTION.
```

The compression function is comprised of rounds that call the mixing function G eight times per round.

```
|      // Cryptographic mixing
|      FOR i = 0 TO r - 1 DO          // Ten or twelve rounds.
|      |
|      |   // Message word selection permutation for this round.
|      |   s[0..15] := SIGMA[i mod 10][0..15]
|      |
|      |   v := G( v, 0, 4,  8, 12, m[s[ 0]], m[s[ 1]] )
|      |   v := G( v, 1, 5,  9, 13, m[s[ 2]], m[s[ 3]] )
|      |   v := G( v, 2, 6, 10, 14, m[s[ 4]], m[s[ 5]] )
|      |   v := G( v, 3, 7, 11, 15, m[s[ 6]], m[s[ 7]] )
|      |
|      |   v := G( v, 0, 5, 10, 15, m[s[ 8]], m[s[ 9]] )
|      |   v := G( v, 1, 6, 11, 12, m[s[10]], m[s[11]] )
|      |   v := G( v, 2, 7,  8, 13, m[s[12]], m[s[13]] )
|      |   v := G( v, 3, 4,  9, 14, m[s[14]], m[s[15]] )
|      |
|      END FOR
```

The main point to note is the first four calls to G can all be done in parallel. As can the second four calls to G. With that in mind the optimal software solution is to utilize Single Instruction Multiple Data (SIMD) vector instructions in general purpose processors (e.g. Intel®, AMD®) that support 4 lanes of 32-bits each for Blake2s and 64-bits each for Blake2b. The Advanced Vector Extensions (AVX) family of instructions are ideal for this type of operation.

Analyzing the mixing function G closer, we note that there are only three core operations: add, XOR, and rotate. There are AVX instructions supporting vector addition and XORs, however none exist for rotation. A key note here is that there will likely never be instructions supporting rotate due to the complexity of wiring a programmable rotation operation in hardware. Therefore, there are two different methods used to implement the Blake rotations, 1) a byte shuffle for the byte aligned rotates (Blake2b – 32, 24, 16; Blake2s – 16, 8), and 2) to use shift operations. For Blake2b the fourth rotation is 63, which can be implemented a number of ways. Optimized code uses a shift right OR'd with an addition. For Blake2s the non-byte aligned rotations (12, 7) use a shift right XOR'd with a shift left.

Utilizing AVX based code, we can determine what the critical path of each algorithm is to calculate the maximum theoretical performance on a general purpose processor. The code for Blake2s shown below has a critical path of 14 cycles per 4-wide parallel mixing function G instance. There are two of those instances per round and 10 rounds, therefore the minimum number of cycles to compress a single block in Blake2s is:

14 cycles * 2 G's (4 in parallel) * 10 rounds = **280 cycles**

Each block is comprised of 64 bytes of data, therefore the minimum theoretical cycles/byte (lower is better) is:

280 cycles / 64 bytes = **4.375 cycles/byte**

Operation	Instruction	Cycle
a = a + b	vpaddd	1
d = a ^ d	vpxor	2
d >>> 16	vpshufb	3
c = c + d	vpaddd	4
b = c ^ b	vpxor	5
b >>> 12	vpsrld	6
b >>> 12	vpslld	6* (in parallel)
b >>> 12	vpxor	7
a = a + b	vpaddd	8
d = a ^ d	vpxor	9
d >>> 16	vpshufb	10
c = c + d	vpaddd	11
b = c ^ b	vpxor	12
b >>> 7	vpsrld	13
b >>> 7	vpslld	13* (in parallel)
b >>> 7	vpxor	14

Blake2s G critical path

The best known software implementation of Blake2s is by one of the algorithm's creators, Samuel Neves (https://github.com/BLAKE2/BLAKE2). This code has been ported to Rust by Jack O'Connor (https://github.com/oconnor663/blake2_simd) and is what Filecoin uses today. The code was instrumented with rdtsc and data was collected over a number of runs of a large buffer (8KB) to determine the implementations greatest achievable performance. The result is 5.25 cycles/byte. This is about a 20% difference in performance between the theoretical maximum and what is implemented. Keep in mind lower is better.

There are changes to the Blake2b code that remove a dependency on the diagonals which are not in the Blake2s code. We decided to implement those changes and push the new code

back to both Samuel Neves' and Jack O'Connor's code bases. Our changes improve performance by about 7%.

Optimized code	5.25 cycles/byte
Supranational optimized code	4.90 cycles/byte
Theoretical limit	4.375 cycles/byte

Blake2s Software vs Theoretical Pre-AVX-512

With the addition of the AVX-512 instruction set there are further enhancements that can be made to the Blake family. From the perspective of the critical path the one change is replacing the shift, shift, xor sequence with a single rotate instruction (vpror). This replaces 6 instructions with 2 and shifts the number of required cycles from 14 down to 12, resulting in a theoretical cycles/byte of 3.75. The code bases were run on a CannonLake NUC featuring the only client processor with the AVX-512 instruction set (Intel® Core™ i3-8121U). The compilers tried were gcc, icc, and clang. Interestingly clang stood far out from the rest doing almost as good a job as our AVX-512 hand optimized code.

Optimized code	4.17 cycles/byte
Supranational optimized code	4.11 cycles/byte
Theoretical limit	3.75 cycles/byte

Blake2s Software vs Theoretical AVX-512

The Blake2b critical path is very similar to Blake2s, with differences coming in how to handle the rotations. This drops the minimum number of cycles down to 13, although there are 12 rounds.

13 cycles * 2 G's (4 in parallel) * 12 rounds = **312 cycles**

The nice part is each block is now comprised of 128 bytes of data, therefore the minimum theoretical cycles/byte (lower is better) is:

312 cycles / 128 bytes = **2.4375 cycles/byte**

Operation	Instruction	Cycle
a = a + b	vpaddq	1

d = a ^ d	vpxor	2
d >>> 32	vpshufd	3
c = c + d	vpaddq	4
b = c ^ b	vpxor	5
b >>> 24	vpshufb	6
a = a + b	vpaddq	7
d = a ^ d	vpxor	8
d >>> 16	vpshufb	9
c = c + d	vpaddq	10
b = c ^ b	vpxor	11
b >>> 63	vpsrlq	12
b >>> 63	vpaddq	12* (in parallel)
b >>> 63	vpor	13

Blake2b G critical path

The best known software implementation of Blake2b is also by one of the algorithm's creators, Samuel Neves, although it is in a different repository coded explicitly for the AVX2 family of instructions (https://github.com/sneves/blake2-avx2). This code has also been ported to Rust by Jack O'Connor (https://github.com/oconnor663/blake2_simd), the same library Filecoin uses today for Blake2s. The code was instrumented with rdtsc and data was collected over a number of runs of a large buffer (8KB) to determine the implementations greatest achievable performance. The result is 2.95 cycles/byte. This again is about a 20% difference in performance between the theoretical maximum and what is implemented. Keep in mind lower is better.

Optimized code	2.95 cycles/byte
Theoretical limit	2.4375 cycles/byte

Blake2b Software vs Theoretical Pre-AVX-512

As with Blake2s there is the possibility of improving the performance of Blake2b with AVX-512. The compilers we tried did not demonstrate any performance improvements therefore we omit the results.

SHA-256
(https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf)

SHA-256 is another hash function to consider for use in Filecoin. The block size, word size, digest size, and maximum message size are all the same as the currently used Blake2s. Similar to the Blake variants, the SHA-256 critical path is the compression function. Therefore we will ignore the message schedule in this analysis. The key differences between SHA and Blake are the number of rounds, the constants required, and the types of operations.

The compression function is comprised of 64 rounds, each with its own constant K:

3. For t=0 to 63:
   {
   $$T_1 = h + \sum_1^{\{256\}}(e) + Ch(e, f, g) + K_t^{\{256\}} + W_t$$
   $$T_2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$$
   $$h = g$$
   $$g = f$$
   $$f = e$$
   $$e = d + T_1$$
   $$d = c$$
   $$c = b$$
   $$b = a$$
   $$a = T_1 + T_2$$
   }

The sigma, Ch, and Maj functions are as follows:

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\sum_0^{\{256\}}(x) = ROTR^{2}(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$$
$$\sum_1^{\{256\}}(x) = ROTR^{6}(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$$
$$\sigma_0^{\{256\}}(x) = ROTR^{7}(x) \oplus ROTR^{18}(x) \oplus SHR^{3}(x)$$
$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

The best known software implementation for SHA-256 is in the OpenSSL library (https://github.com/openssl/openssl). Recently Intel® and AMD® implemented new instructions called the SHA Extensions to support SHA-256 natively. This dramatically improves performance on general purpose processors by almost 4x. One issue with SHA Extensions is support from Intel® is still nascent. Although support is limited, the expectation is soon the instructions will be ubiquitous, therefore theoretical analysis will focus on them.

The SHA Extensions have a SHA-256 round instruction called SHA256RNDS2, see detailed white paper for more information: https://software.intel.com/sites/default/files/article/402097/intel-sha-extensions-white-paper.pdfThe instruction performs two rounds per instance, therefore the 64 rounds of a single block in SHA-256 will take 32 calls to the instruction. The theoretical minimum number of cycles for a single block is: SHA256RNDS2 latency * 32 calls. As opposed to the basic AVX instructions used in Blake, the SHA instructions typically have latencies greater than 1 cycle. That is why the performance is defined in this fashion.

Each block is comprised of 64 bytes of data, therefore the minimum theoretical cycles/byte (lower is better) is:

SHA256RNDS2 latency * 32 / 64 bytes = **latency/2 cycles/byte**

Currently the best known latency is on the AMD® Ryzen™ class of CPUs which has a 4 cycle latency. Therefore the minimum theoretical limit is 4/2 = **2 cycles/byte**. In fact Ryzen™ hits this number, therefore the architecture and code are already aligned to reach maximum performance.

In theory future processors could decrease the latency of the SHA256RNDS2 instruction. This is a difficult task given the critical path and frequencies those processors must support. An aggressive upper bound estimate on lowest possible latency is 2 cycles. Therefore in the future we highly doubt there will be any x86 general processor that approaches or goes lower than 1 cycle/byte.

Optimized AVX code (ubiquitous)	7.56 cycles/byte
Optimized SHA Extensions code (limited)	~2.0 cycles/byte
Theoretical SHA Extensions limit	2.0 cycles/byte
Theoretical SHA Extensions limit future This is an assumption	1.0 cycles/byte

SHA-256 Software vs Theoretical

FPGA Analysis

As a comparison to CPU performance, we looked at what could be achieved on an FPGA for hashing performance. The PoRep operation is purposely designed to be sequential, therefore latency is the most important factor in evaluating optimal performance. This is in stark contrast to something like cryptocurrency mining where throughput is the dominating performance indicator.

A quick note on GPGPUs. A major advantage of GPGPUs over CPUs is performance in extremely parallel algorithms or throughput oriented applications. Again this is not the case with PoRep, a purposely sequential operation. Therefore for the purposes of this analysis, which was directed at improving the encoding operation, GPGPUs were not considered.

In an ideal setting the encoding operation is comprised entirely of the collision resistant cryptographic hashing function. Therefore to make the case for FPGAs over a CPU we need to understand if an FPGA can hash faster than a CPU. This is a rough generalization since data movement to/from the hashing engine, graph lookups, and the sloth based encoding are also a part of the operation. However, the coarse hash comparison will provide insight into possible future directions.

We started with SHA-256 to get a feel for what could be done on the FPGA. We implemented a low latency optimized SHA2 core based closely on the “Improving SHA-2 Hardware Implementations” paper by Ricardo Chaves et al. Our core operates at a frequency of 400 MHz on a Xilinx® Virtex® Ultrascale+™ based VCU118 FPGA board and performs one round per cycle (which implies one cycle/byte). This is dramatically faster than what we found published, although to be fair, the FPGA is much more capable than the ones most academic papers tend to use.

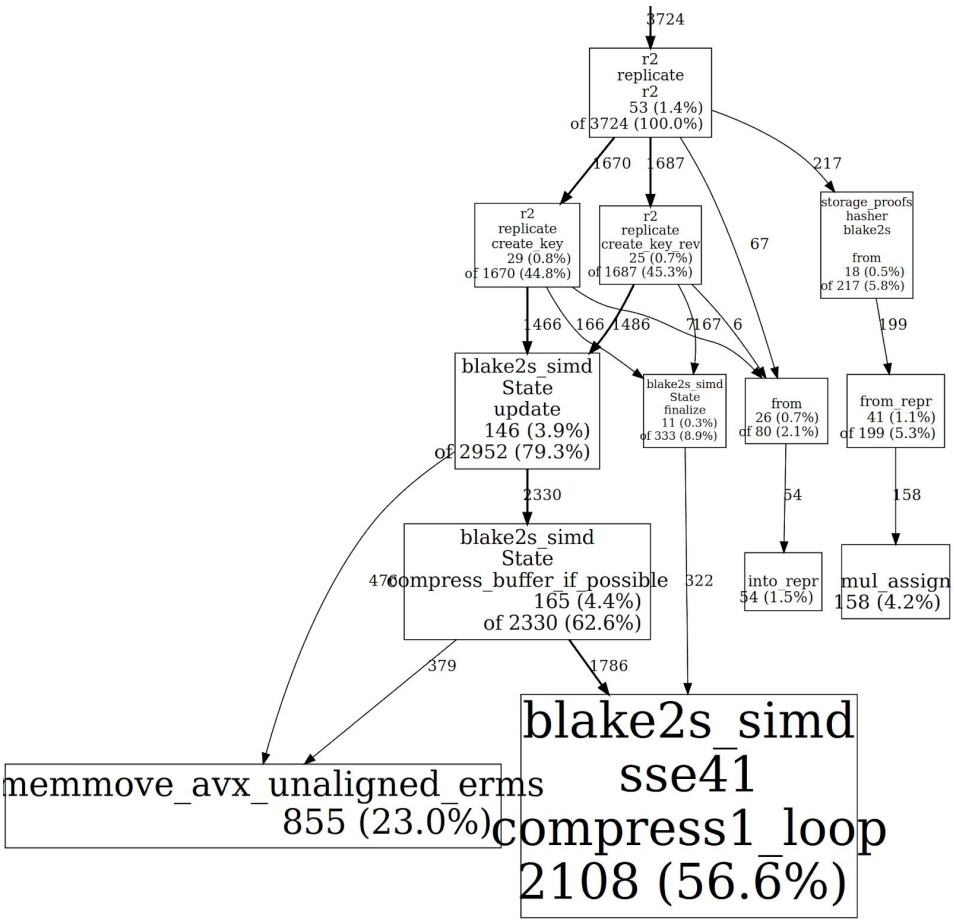
At one cycle/byte the FPGA does twice as much work as the CPU at two cycles/byte in a given cycle. However the FPGA is dramatically slower. A CPU running at more than 800 MHz with the SHA Extensions will have better performance than the FPGA. Since frequencies are generally in the 2-4 GHz range for a CPU, there will be no latency advantage doing replication on an FPGA with SHA-256.

For Blake2s we designed a straightforward 200 MHz FPGA core that takes 20 cycles per block, equating to 0.3125 cycles/byte. Compare this to the 4.90 cycles/byte we saw with the optimized rust code on CPU. The case is not as clear cut as it was with SHA2, as the cycles/byte is dramatically (~16x) lower. However at 3.2 GHz and above the CPU still has an advantage. We believe there is further opportunity to squeeze more performance out of the FPGA, although not enough to make a large difference. Given the extra overhead of data movement on top of the hashing function, the belief is for this latency bound problem the FPGA is still not the answer.

Replication Study and Improvements

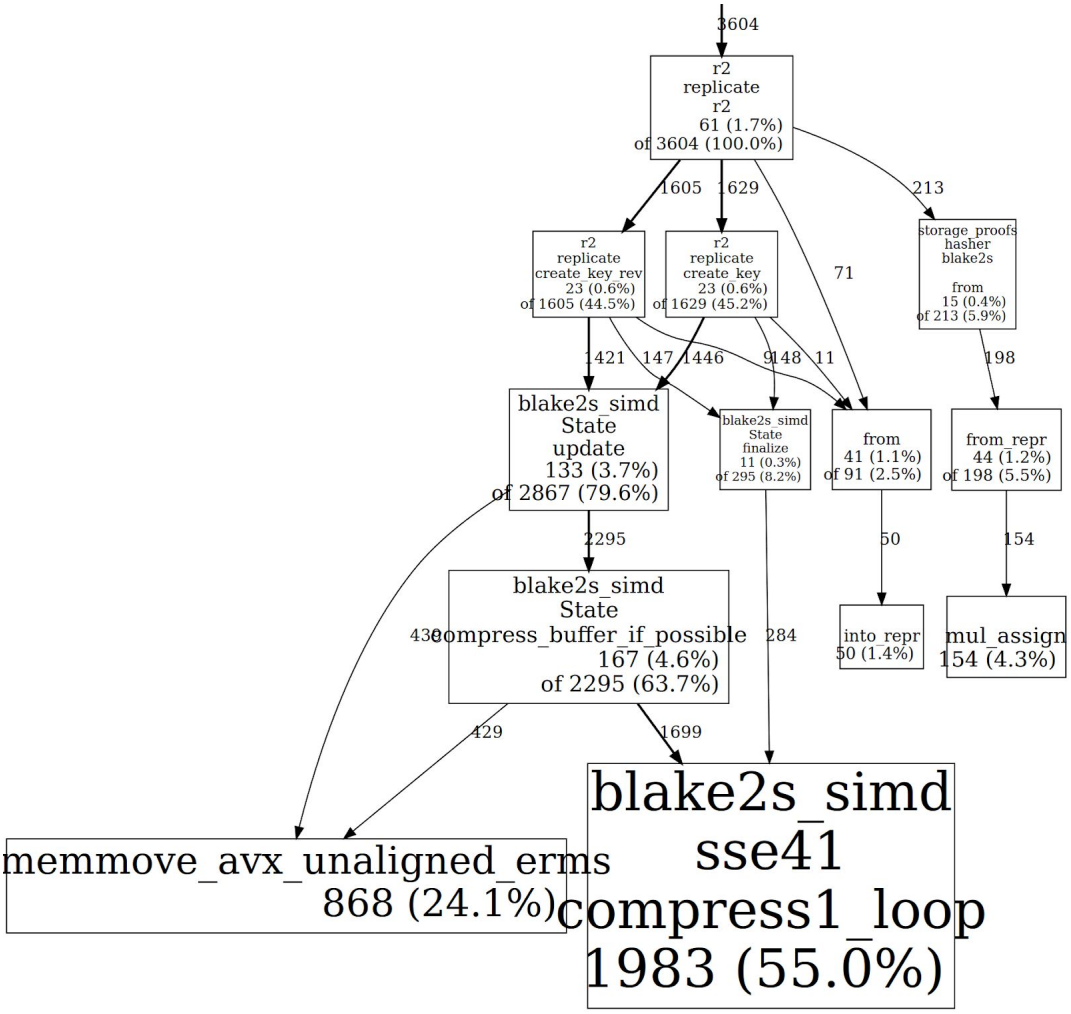
We started with the independent replication repository “r2” (https://github.com/nicola/r2) to better understand the bottlenecks of the PoRep.Replicate algorithm. In particular, we used the better performing “parents-iter” branch of the repository as a starting place.

We attached gperftools to the code and created a profile for where time was being spent in the replicate function.



Profile of original r2 codebase on parents-iter branch.

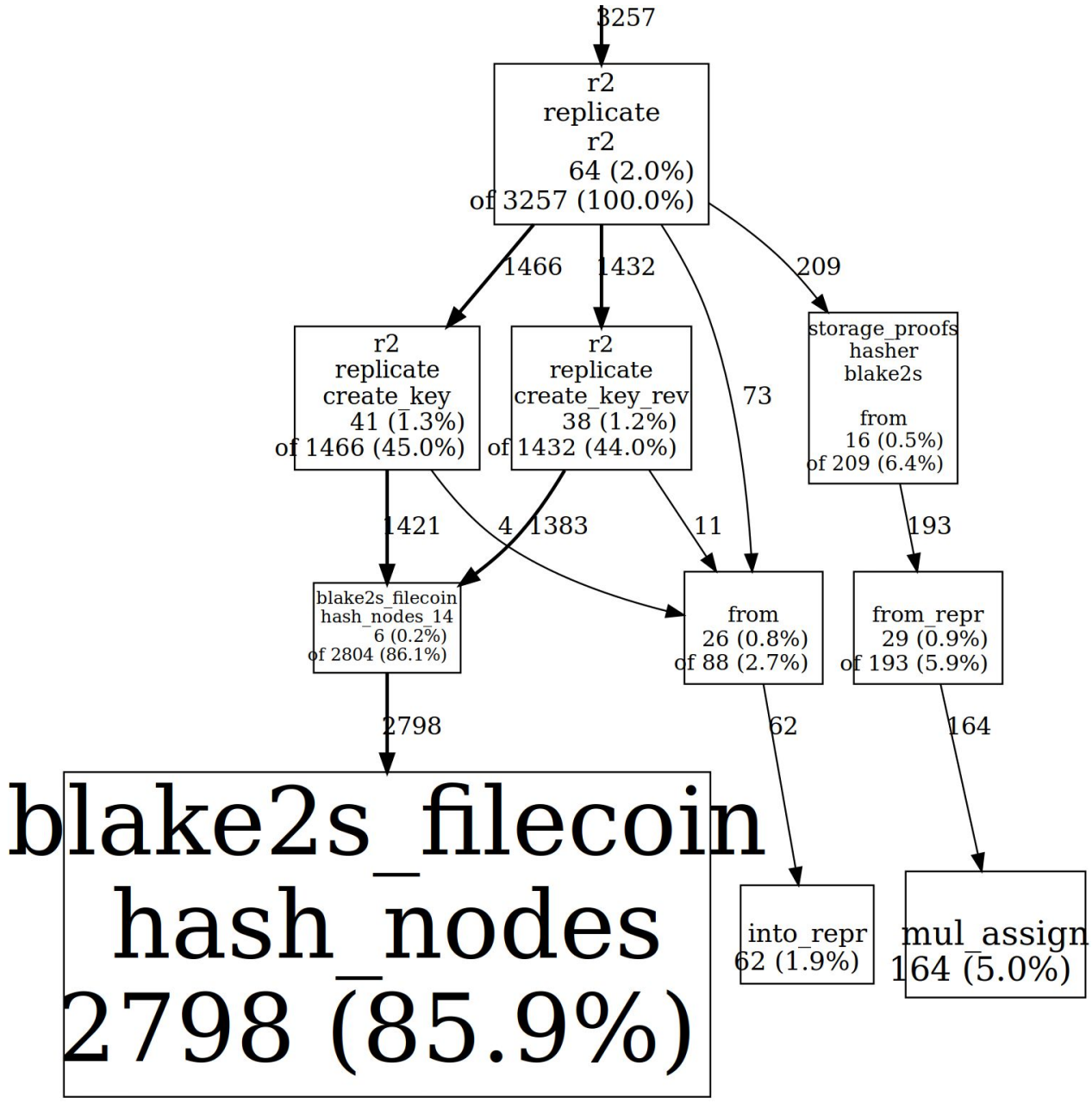
Given that Blake2s took the majority of the time, we first focused our efforts on improving the performance of the Blake2s library. After we improved the blake2_simd repository our pull request was quickly incorporated and a new version of the crate was released (0.5.5) by the owner. This improved the overall performance by about 3%, which makes sense given that we were able to improve Blake2s by about 6% and Blake was consuming about 50% of the replication function.



Profile of original r2 codebase on parents-iter branch with Supranational improved blake2_simd.

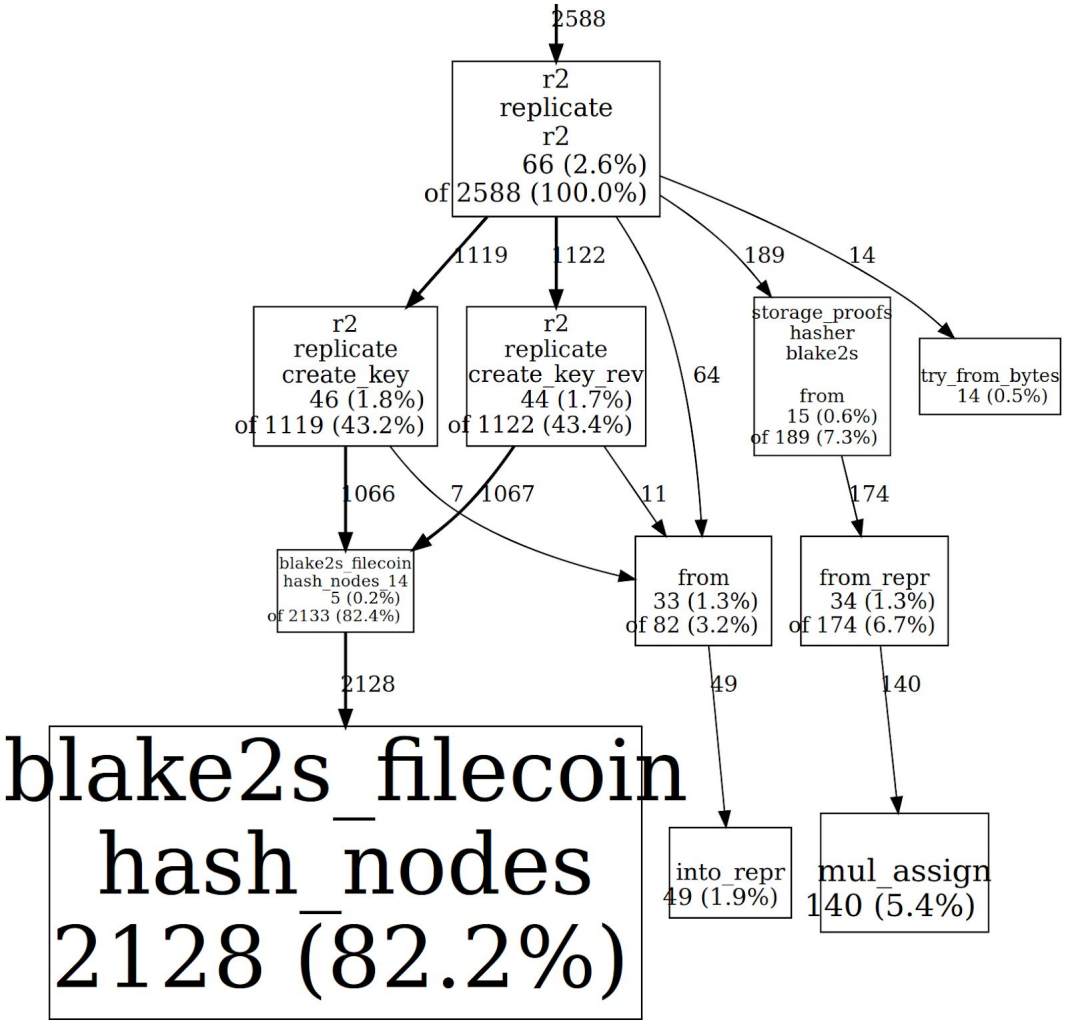
The next place to tackle was the issue of over 20% of the time being spent copying data. We determined the cause to be repeated calls to the hash update function, which copies data into a temporary buffer if a full block is not available to be hashed. This occurs half the time the function is called.

In order to improve the performance we wrote a hashing function specifically for Filecoin that eliminates the memory copies. This was accomplished by passing a reference to each of the parents all at once. The repo containing the new hashing function is here: https://github.com/sean-sn/blake2s_filecoin. The fork that integrates into r2 can be found here: https://github.com/sean-sn/r2. The result was a complete elimination of the memory copies and another 10% improvement over the previous run with the optimized blake2_simd.



Profile of Supranational r2 codebase with no copy hash function.

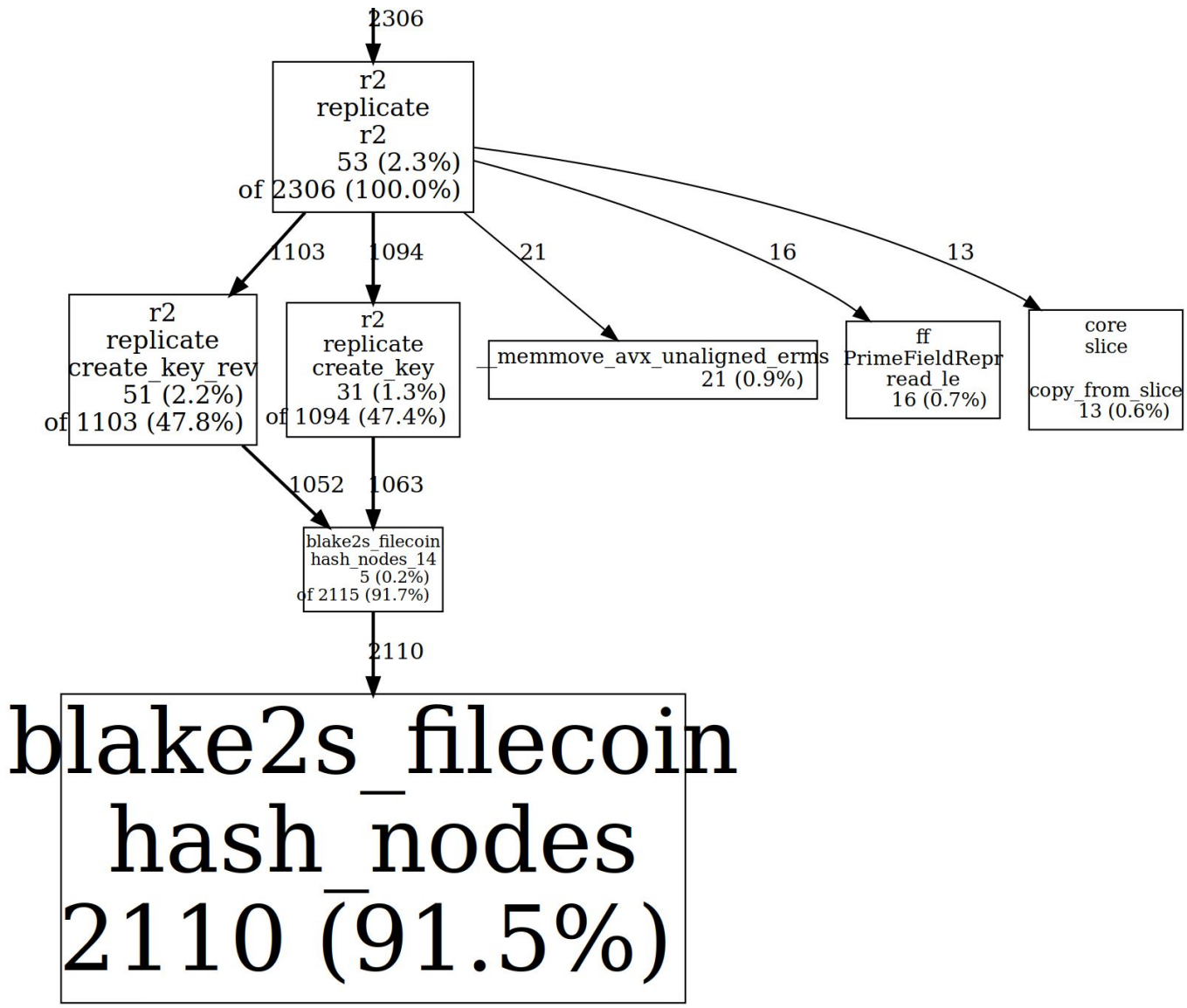
The result of having Blake be the majority of the computation is great, however the overall performance was still well off that of the underlying Blake library capabilities. After looking at vtune results we made a change to the hashing function to add data prefetching. To do this we loop through each parent and signal to the CPU to prefetch the data prior to the start of hashing. This lead to the most dramatic change, another 20% gain in performance.



Profile of Supranational r2 codebase with prefetching in new hash function.

Ideally the hash function is fully optimized and consumes 100% of the replication time. Given the hash function for non AVX-512 machines is a little over 10% above theoretical limit, there is not much more we can take from the hash function itself.

The next largest time consumer was the mul_assign function, which we were able to eliminate all together. We discovered that during the sloth encoding the inputs were being unnecessarily being transformed into Montgomery space to perform a basic modulo addition. By removing the transformation of data into the field representation we were able to eliminate the $A*R^2$ mod M operation and all of the associated mul_assign compute time. The new code is a straightforward addition followed by a conditional subtraction of the modulus if the result is greater than the modulus. The outcome of an updated profiling shows the hash function jumped an additional 10% to 91.5% of the overall operation, closer to the idealistic 100% goal.

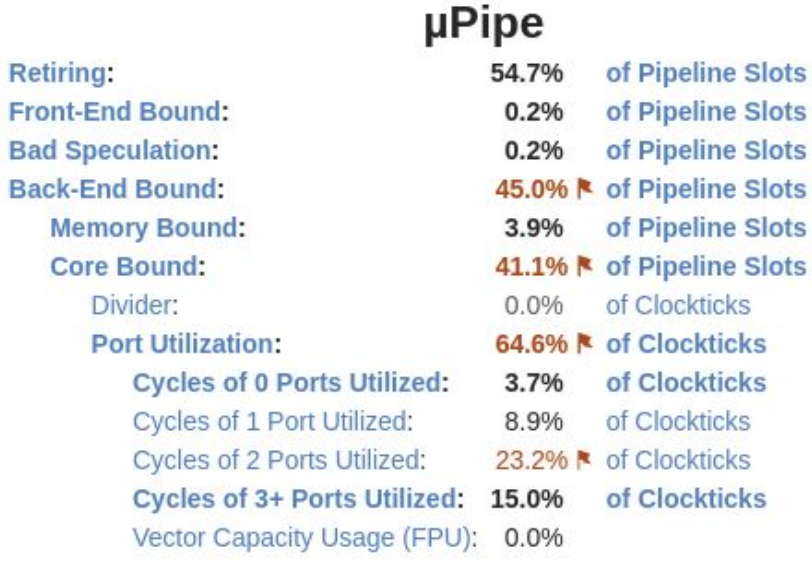


Profile of Supranational r2 codebase after elimination of mul_assign.

There are some additional minor improvements that could be made from here such as incorporating the encoding directly into assembly with the hash function. This would not be a difficult task and may improve the performance slightly.

The micro-architectural analysis of the final codebase shows the hash function is indeed core execution unit bound. As can be seen in the next two figures comparing with and without prefetching, the method has pretty much eliminated the memory issues. Interestingly the 3.9% of the hash that is still memory bound comes from LLC misses that occur only in the reverse direction. This should be noted for examination in the mainline to ensure there are no discrepancies between forwards and backwards.

hash_nodes() mirco-architectural analysis of Supranational r2 codebase without prefetching.



hash_nodes() mirco-architectural analysis of final Supranational r2 codebase.

Data Scaling

Test runs of up to 64GB of data were attempted on an AWS machine with 128GB of memory. Tests were failing during the 8GB and 16GB runs due to memory allocation issues during both the store and load. This was due to writing the entire graph to an in memory JSON buffer before writing it to disk. Similarly on load the entire graph JSON representation was read from disk into memory prior to passing it to the JSON parser. The graph caching code was modified to use buffered I/O for both reads and writes of the graph. This allowed for reliable runs at 8GB and 16GB configurations.

When running the test configurations the following usage was observed. Memory usage was captured using the time command: `/usr/bin/time -f time.txt <cmd>`. The memory scaling behavior is shown to be linear with the number of nodes being processed.

Size	**Node**	**Max Resident(k)**	**Bytes/Node**
128	4194304	825956	202
256	8388608	1645388	201
512	16777216	3284616	200
1024	33554432	6561632	200
2048	67108864	13115524	200
4096	134217728	26222744	200
8192	268435456	52437276	200
16384	536870912	104865744	200

The 32GB and 64GB runs failed due to out of memory conditions.

Nodes become the driving factor of memory usage, as the graph allocates records for tracking the parents of each graph. Analysing the source code we see that there are allocation in the main data structures that match this size.

Allocation	Type	Size (Bytes)
Data	u8[NODE_SIZE]	32
Graph		
bas	usize[5]	40

exp	usize[8]	64
exp_reversed	usize[8]	64
Total		200

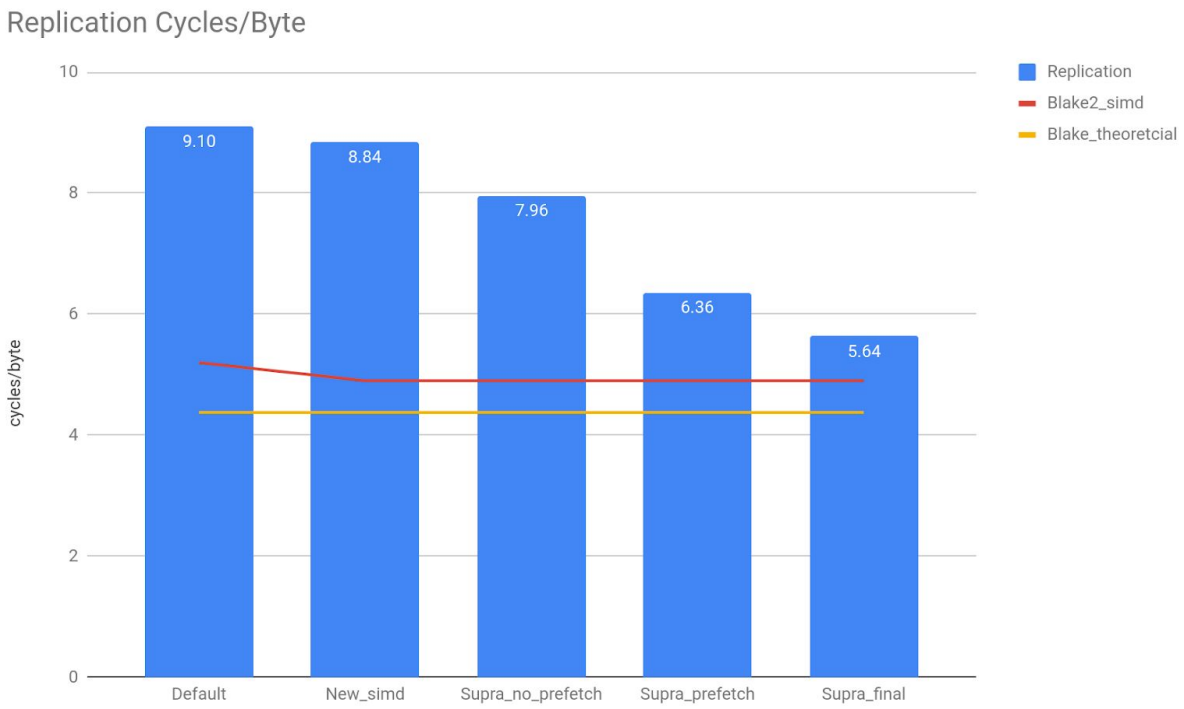
Performance Improvements Summary

We chose cycles/byte as the preferred metric for calculating performance due to the ease of comparing against a purely compute bound theoretical implementation of Blake2s. This gave us a good idea on what the operation's overhead is and guided optimization points. An additional benefit is the ability to be CPU SKU agnostic and provides an expected performance on any given platform by dividing the frequency by the cycles/byte value to provide bytes/second.

All performance data was collected on machines operating with static frequency to ensure consistency run to run. The cycle counts were collected using the read timestamp counter instruction which takes a snapshot of the current monotonically incremented processor clock cycle counter. A reading was placed at the start and end of the replication layer macros. This was chosen to demonstrate the differences in forward vs. reverse graph traversal.

The following data was collected on an Intel® Core™ i9-9900K at 3.60 GHz with 64 GB of RAM and a 1 TB NVMe drive. The DATA_SIZE parameter is set to encode 100 MB.

Model	Cycles/Byte
Default parents-iter	9.10
parents-iter with new blake2_simd	8.84
New supra hash without prefetch	7.96
Supra hash with prefetch	6.36
Final supra hash with no mul_assign	5.64



The recommended next steps are to incorporate the r2 changes into the mainline Filecoin repository to see the overall performance improvements. A fresh look at the hotspots there would point to areas for further optimization if any.

PoRep HW Acceleration and AMax Estimates

In addition to evaluating the performance of PoRep on CPU and FPGA architectures, we also assessed the cost and performance of a dedicated PoRep accelerator. Based on the above performance analysis we focused primarily on the speed at which data could be encoded. As mentioned above, over 90% of the encoding time is spent hashing data, and the below analysis assumes the compute engine can be fed with data sufficiently fast such that memory or data movement is not a bottleneck. The shape of the graph is known in advance of encoding so it should be possible to place data in fast memory close to the compute engine (DRAM or SRAM) just-in-time through intelligent caching. Data can be moved to near-memory over a variety of interfaces including SATA, NVMe, PCIe, and Ethernet, with bandwidths over 200 Gb/s.

Compute Performance							
Hash Function	**Compute Engine**	**Compute Frequency (MHz)**	**Cycles / Byte**	**Single Threaded Bandwidth (GB/s)**	**Variable Compute Engine Cost[2]**	**Latency $ / GB/s**	**Fixed Engineering Cost**
SHA	Ryzen w/ SHA-NI	5000	2.0	2.5	~$400	$160.00	$0
SHA	Intel w/ SHA-NI	5000	3.0	1.7	~$400	$240.00	$0
SHA	x86 w/ AVX2	5000	7.5	0.7	~$400	$600.00	$0
SHA	FPGA	450	1.0	0.5	~$2,000	$4,444.44	$0
SHA	ASIC	7500	1.0	7.5	$20-$200	$3.33	$1-$20M
SHA	SmartNIC	3500	1.0	3.5	$500-$1500	$400.00	$1-$5M
Blake2s	x86 w/ AVX2	5000	5.0	1.0	~$400	$400.00	$0
Blake2s	FPGA	250	0.3	0.8	~$2,000	$2,560.00	$0
Blake2s	ASIC	4000	0.3	12.5	$20-$200	$2.00	$1-$20M
Blake2s	SmartNIC	3500	0.3	10.9	$500-$1500	$128.00	$1-$5M
Blake2b	x86 w/ AVX2	5000	3.0	1.7	~$400	$240.00	$0
Blake2b	FPGA	250	0.2	1.3	~$2,000	$1,536.00	$0
Blake2b	ASIC	4000	0.2	20.8	$20-$200	$1.20	$1-$20M
Blake2b	SmartNIC	3500	0.2	18.2	$500-$1500	$76.80	$1-$5M

Table demonstrating single threaded performance of different hashing algorithms and architectures.

AMax Estimates			
Hash Function	**Off-the-Shelf Latency Optimized (GB/s)**	**Custom Latency Optimized (GB/s)**	**Amax**
SHA	2.5	7.5	3
Blake2s	1.0	12.5	12.5
Blake2b	1.7	20.8	12.5

Table demonstrating the performance of 'off-the-shelf' encoding hardware vs. custom hardware

After analyzing the various hashing algorithms on different hardware architectures we were able to determine that SHA provides the fastest encoding time on 'off-the-shelf' hardware due the presence of embedded SHA accelerators on modern CPUs that support the SHA-NI

[2] All cost estimates are approximate estimates. Actual costs will depend on a variety of factors such as SKU, process technology, compute engine capabilities, etc.

instruction. The presence of these accelerators, in combination with high operating frequencies, also provide the lowest Amax of the hashing algorithms when compared with custom hardware. While a dedicated SHA encoding ASIC may be able to perform SHA with a lower cycles/byte and at a greater frequency than a CPU, it is unlikely that this hardware could be more than three times faster without incurring significant R&D costs. However, if it is decided that a custom hardware strategy were to be pursued, both Blake2s and Blake2b could provide significantly higher single threaded encoding throughput due to their significantly lower cycles/byte performance.