Applying FPGA Runtime Reconfiguration to Multi-Hash Proof-of-Work Algorithms

Author:
Wu, Tong

Publication Date:
2022

DOI:
https://doi.org/10.26190/unsworks/25287

Applying FPGA Runtime Reconfiguration to Multi-Hash Proof-of-Work Algorithms

Tong Wu

A thesis in fulfilment of the requirements for the degree of

Master of Philosophy

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

October 2022

✓ The candidate has declared that **some of the work described in their thesis has been published and has been documented in the relevant Chapters with acknowledgement**.

A short statement on where this work appears in the thesis and how this work is acknowledged within chapter/s:

Chapter 5 is partially based on our research that was published in FCCM2022. I have acknowledged this in the first chapter.

Candidate's Declaration

✓ **I declare that I have complied with the Thesis Examination Procedure.**

Acknowledgement

I would like to express my immense thanks to my supervisor, Prof. Diessel, for his invaluable advice, support, and patience through my masters research.

I would also like to thank my parents, Lucy and Davis, and my girlfriend, Maddie, for their unending love and support, without which I would not have been able to embark on this work.

Abstract

In the cryptocurrency mining field, algorithms have been developed to discourage the development of ASICs that greatly out-compete general-purpose hardware in both performance and power efficiency. A class of algorithms that claims to be ASIC-resistant is the class of randomised multi-hash proof-of-work algorithms, such as X16R. For these algorithms, the result of one iteration depends on the chained application of several randomly selected hash functions, which has the effect of disadvantaging fixed-function ASICs due to their inflexibility. FPGAs lie between GPUs and ASICs in terms of raw performance and flexibility. We investigate the use of FPGAs for this type of proof-of-work, in particular, by leveraging the ability of modern FPGAs to quickly reconfigure at runtime. We implemented a design that runs the X16R algorithm by partially reconfiguring the FPGA for every hash function in the chain and processing the data in batches. We show that our system achieves better performance when compared to GPUs that are manufactured on the same semiconductor process technology node, while being several times more power efficient. The two key takeaways from this work are that FPGA runtime reconfiguration can be used to effectively accelerate algorithms for which the demand for different processing elements changes over time, and that proof-of-work algorithm designers should consider FPGAs as a class of computing device that is separate from fixed-function ASICs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Blockchain based cryptocurrencies, such as Bitcoin, are distributed ledgers that require many distributed bookkeepers in order to function. To protect the blockchain from spam, a proof-of-work based consensus protocol is used to reach an agreement on the current state of the ledger. Proof-of-work (PoW) algorithms require the bookkeepers to perform some work, which usually involves executing a hashing algorithm many times. The bookkeepers doing this work are referred to as miners. Initially, on cryptocurrencies such as Bitcoin, miners were able to use commodity hardware such as CPUs and GPUs to perform proof-of-work. However, over time, Application-Specific Integrated Circuits (ASICs) were developed that offered an increase of orders of magnitude in performance and energy efficiency over CPUs and GPUs. Today, many cryptocurrencies try to introduce ASIC-resistance to their proof-of-work algorithms in order to maintain the viability of CPU or GPU mining. One of these, ASIC-resistance methods, relies on *multi-hash* PoW algorithms that chain together several different hash functions to ensure that there is additional complexity when being implemented in hardware. Some multi-hash PoW algorithms, such as X16R, also require the sequence of hash functions being executed to be randomly selected at the start of every time window.

Field-Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a reconfigurable matrix of lookup tables (LUTs), flip-flops (FFs) and interconnects. They are capable of implementing any boolean logic circuit (given that the circuit fits within the number of LUTs and FFs available) and are often used to accelerate algorithms that are ill-suited for traditional processors (CPUs and GPUs). FPGAs differ from ASICs in that they are reconfigurable, allowing a single device to be re-purposed for many different applications, while sacrificing some performance and energy efficiency. It is also possible to reconfigure portions of the FPGA at run time, allowing for the user to change the functionality without a reboot. This feature is often referred to as dynamic partial reconfiguration (DPR) or runtime reconfiguration.

Multi-hash chain PoW algorithms that randomize the order of hash functions have the effect of introducing a significant overhead within ASICs and static FPGA implementations. This research aims to investigate the use of FPGA runtime reconfiguration to improve the performance of these types of algorithms, by allocating FPGA resources as needed by the algorithm during runtime.

1.2 Contributions

This thesis makes the following contributions:

1. A design exploration of potential FPGA-based designs for computing X16R and multi-hash PoW algorithms in general. We find that a static, fixed-function design is impractical to implement on the current generation of FPGAs and would be bottlenecked in performance due to the inherent properties of the X16R algorithm. We consider two dynamically reconfigurable architectures, *dynamic full chain*, and *time-sliced dynamic sub-chains,* which allow for the elimination of the bottlenecks apparent in the static design. We determine that the time-sliced approach can be more practically implemented.

2. An analysis of the overheads within the time-sliced dynamic sub-chains approach to X16R and the expected performance. We find that our time-sliced approach to X16R does not lose much efficiency for blockchains with lower average block intervals, even as low as 12 seconds.

3. A comparison of the maximum clock frequency achieved by using the auto-pipelining features of PipelineC, Vitis HLS, and hand-optimized RTL. We find that Vitis HLS

performs comparably or better than hand-optimized RTL on small to medium designs in general, and also when below 666MHz for large designs. PipelineC also performs well for small and medium designs, however, the long compile times required make it infeasible to use for large designs.

4. Implementations of three variants of our time-sliced dynamic sub-chains architecture, SFR, MFR, and SFR2. These achieved several times better power efficiency than GPUs, even when comparing to GPUs that were manufactured on a more recent technology node. Therefore, we demonstrate the utility of FPGA runtime reconfiguration in speeding up and making possible the computation of multi-hash PoW algorithms on widely available FPGA hardware.

1.3 Publications and awards

During the course of our research we published a portion of our work on the design and implementation of our time-sliced approach to computing X16R. Specifically, we submitted our Single-Function-per-Reconfiguration (SFR) design variant to **The 30th IEEE International Symposium On Field-Programmable Custom Computing Machines** conference (FCCM2022) as a 4-page short paper where it was **accepted**. My supervisor and co-author provided advice regarding the direction of research and editorial support.

- Wu, T. and Diessel, O., 2022. Leveraging FPGA Runtime Reconfigurability to Implement Multi-Hash-Chain Proof-of-Work. In: *The 30th IEEE International Symposium On Field-Programmable Custom Computing Machines.*

We also submitted a simplified version of our SFR2 design variant to the **AMD-Xilinx 2021 Adaptive Computing Challenge** and it was **awarded second place** in the *Big Data Analytics* category.

- Fast DFX for Multi-hash Algorithms — https://www.xilinx.com/developer/adaptive-computing-challenge/contest-2021.html

Chapter 2

Background

2.1 Cryptocurrency mining

2.1.1 Proof-of-Work

Cryptocurrencies based on blockchains are in essence distributed data structures where information is stored as ever-growing lists of transaction records, grouped into objects called blocks. Each block contains a cryptographic hash of the previous block, thus linking every block in a chain. Updates to the blockchain can only happen when a new block is added to the chain. Bitcoin is designed such that a new block is added to the chain approximately once every ten minutes. So as to allow any party to contribute, and to prevent all participants from attempting to do so at the same time, the system must choose a random participant to submit the next block. This is done by having the submitter attach a *Proof-of-Work* (PoW) along with their submission. This proof-of-work is the solution to a computationally intensive problem that everyone has a fair chance of solving and is representative of the effort they have invested. The first to find the solution and submit the new block is rewarded some freshly minted cryptocurrency (e.g. Bitcoin). Since there is a promise of reward, many are willing to expend computational power and energy to increase their chance of getting the reward. This is called mining, as it is analogous to mining for gold.

As an example, we examine the proof-of-work system for the most popular cryptocurrency, Bitcoin [23]. The data structure of a Bitcoin block is shown in Table 2.1.

Field	**Description**	**Size**
Block Size	size of this block	4 bytes
Block Header	other metadata	80 bytes
Transaction Counter	number of transactions in this block	1-9 bytes
Transactions	list of transactions	variable

Table 2.1: Bitcoin block structure

For PoW mining, we are only concerned with the 80-byte *block header* which is the input to the PoW algorithm. The data fields of a Bitcoin block header are shown in Table 2.2.

Field	**Description**	**Size**
Version	number to track protocol updates	4 bytes
Previous Block Hash	reference to previous block	32 bytes
Merkle Root	Merkle tree root for transactions in this block	32 bytes
Timestamp	approximate creation time of the block	4 bytes
Difficulty Target	encoded PoW target for this block	4 bytes
Nonce	a counter used for PoW	4 bytes

Table 2.2: Bitcoin block header

The Bitcoin PoW algorithm, called SHA256D, is a double application of the standard SHA256 hash function, which takes the block header as input and outputs a 32-byte hash. Specifically, SHA256D is defined as:

$$\text{SHA256D}(x) = \text{SHA256}(\text{SHA256}(x))$$

If the resulting hash is numerically smaller, when interpreted as an unsigned integer, than a specific *target* value, then the block header input that produced the hash is considered a valid proof-of-work. The target is a 256-bit value that is derived from the 4-byte *difficulty target* that is contained within the block header. The difficulty target is encoded as a 24-bit significand and an 8-bit exponent, such that the actual target is calculated as:

$$\text{significand} \times 256^{\text{exponent - 3}}$$

Miners are able to change a few fields of the block header as a source of variation in order to generate many different inputs to the PoW hash function. The miner is free to choose to include or not any transactions in the block and their ordering in the list, as long as all transactions are valid (i.e. correct accounting has been applied). This changes the *Merkle root* value in the block header, as it is a digest of all transactions in a block. The miner is also allowed to change the *timestamp* to a value that is greater than the median timestamp of the previous 11 blocks and less than the network adjusted time[1] plus 2 hours. Miners are also free to change the *nonce* field to any value of their choosing.

The Bitcoin mining algorithm is simply:

```
def mine_bitcoin(block_header):
  while sha256d(block_header) >= target:
    block_header.nonce += 1
  return block_header
```

Once a miner finds a *wining* hash (meaning a hash that is smaller than the target), it broadcasts its version of the block to the rest of the network. If the network is satisfied that the rules for including transactions have been followed and the PoW is valid, then the winning miner will be credited with a freshly minted Bitcoin reward (6.25 Bitcoins as of 2020-2024) and the transaction fees for each transaction included in the block, and then that particular block is added to the blockchain. While it takes many trials for a miner to find a valid PoW, it only takes one hash for others to verify its validity.

The exact mechanism with which the winning miner is awarded Bitcoins is that, by protocol, every miner is allowed to set the first transaction in the block called the *coin-base transaction*, which is a transaction that mints new Bitcoins to a specified address. Typically the miner will set the receiving address for the coin-base transaction to an address they own.

The Bitcoin protocol adjusts the difficulty target every 2016 blocks in order to keep the expected *block interval* (i.e. the time between blocks being added to the chain), as close to 10 minutes as possible.

2.1.2 Pooled mining

Well-established blockchains are typically made up of thousands of individual miners. Thus, the likelihood of an individual miner winning a block is extremely low, which leads to a high variance in the frequency of their payouts. However, miners have regular operational

[1]The network adjusted time is the median of the wall clock times returned by all other nodes that are connected to the miner.

costs, such as their electricity, maintenance, and labour costs. Therefore, it is common practice for smaller miners to pool their computational power and internally distribute their earnings according to the amount of computational power each miner brings to the pool. With their combined hash rate, the pool is more likely to find a valid PoW, which lowers the variance of payout frequency.

To prevent cheating by pool participants, called *workers*, the pool operator chooses which transactions are included in the block and, in particular, sets the coin-base transaction such that the payout is awarded to the pool itself, and not any of the workers. The pool provides a block header *template* to each individual worker connected to the pool, where the Merkle root is already set and the miners are only allowed to change the timestamp and nonce fields. To keep track of the amount of computational power each worker is contributing, the pool requires that the worker also reports any hashes that satisfy a much easier target (called the pool target) than the global target. This way, the pool is able to get frequent reports, called shares, which are essentially mini-PoWs that the pool can use to verify that the workers are working on the correct block header and to be able to estimate the hash rate contributed by each worker. Any hash that meets the global target will generate a payout to the pool itself, which is then distributed among the workers.

Pooled mining introduces the risk that the pool operator refuses to pay out. In such cases the individual miner can only cut their losses and switch to a different pool.

2.1.3 Evolution of mining hardware

Initially, on blockchains such as Bitcoin, miners were able to use commodity hardware such as CPUs and GPUs to perform proof-of-work. As cryptocurrency mining became more financially lucrative, miners sought to gain efficiency advantages over their competitors by utilizing specialized hardware. In [3] a timeline following the evolution of Bitcoin hardware was summarised, with CPUs dominating the network until late 2010, after which GPUs and FPGAs took over and then finally ASICs started dominating the network from 2013.

In Figure 2.1 we plot the historical Bitcoin network hashrate, marking the different mining hardware eras[2]. The increase in total network hash rate is driven by a combination of

[2]data from https://www.blockchain.com/charts/hash-rate, eras from [3]

more miners coming onto the network (due to Bitcoin's price increasing) and technological advancements in mining hardware.

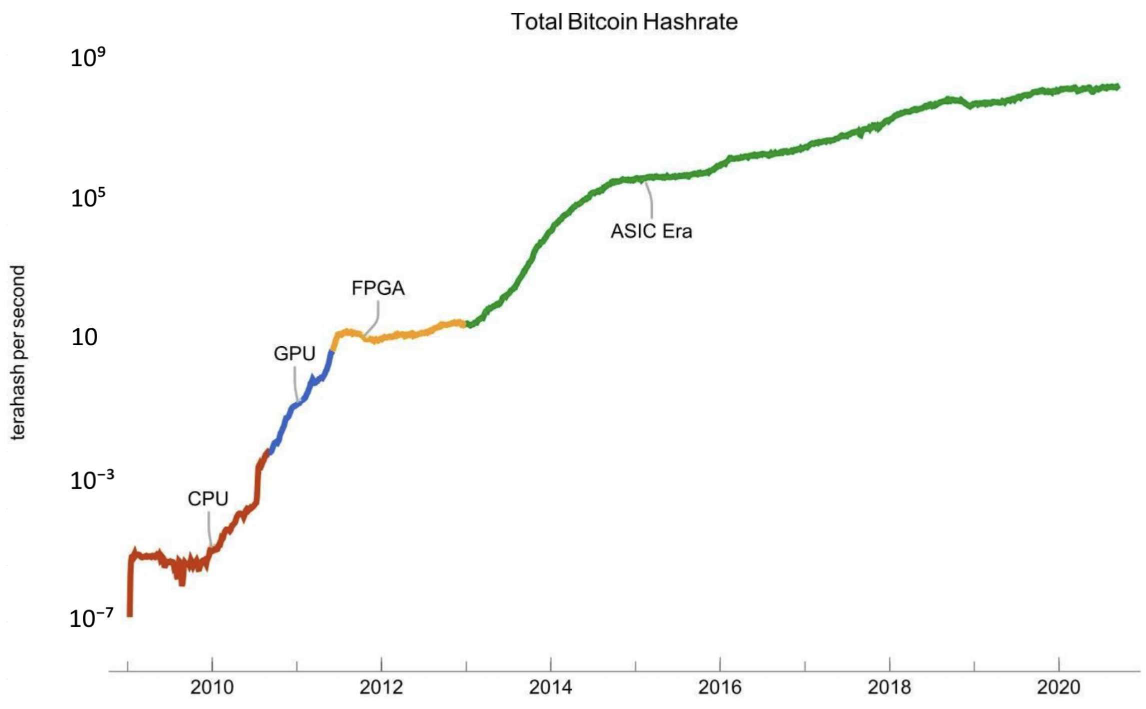

Figure 2.1: Bitcoin network hashrate

Typically for the SHA256D PoW algorithm, CPUs, GPUs, FPGAs, and ASICs are able to achieve on the order of tens to hundreds of mega-hashes per second, hundreds of mega-hashes to giga-hashes per second, giga-hashes to tens of giga-hashes per second, and tera-hashes per second respectively.

2.1.4 ASIC-resistance

Due to ASICs having a high barrier to entry for the average consumer and the at-home miner, it is feared by the crypto-community, as surveyed in [28], that cryptocurrency mining will become increasingly centralized among large ASIC farms and ASIC manufacturers. Therefore, many cryptocurrencies (e.g. Ethereum [8] and RavenCoin [11]) decided to introduce forms of ASIC-resistance into their PoW algorithms. We consider three main classes of ASIC-resistant algorithms; multi-hash, memory-hard, and algorithmic.

2.1.4.1 Multi-hash

One example of a multi-hash PoW algorithm is X11 [10], which chains together eleven hash functions in order; Blake, BMW, Grøstl, JH, Keccak, Skien, Luffa, Cubehash, Shavite, Simd and Echo. A descendant of X11 is X16R [7] which uses up to sixteen different hash functions (all SHA3 Round 2 candidates plus whirlpool and SHA512). However, while X11 uses a fixed chain of hash functions, the sequence of hash functions is randomised for every new block in X16R. Additionally, the same hash function can appear more than once in the chain of sixteen, or not at all.

In [9], the author aimed to assess the ASIC-resistance of multi-hash proof-of-work algorithms. Three types of multi-hash PoW algorithms were identified: fixed sequence hash chains (e.g. X11), hash chains with a variable sequence per block (e.g. X16R), and hash chains with a variable sequence per nonce.

The author then detailed three hardware platforms which were used for their experiments, an Intel i7-8700K CPU, an NVIDIA GeForce GTX 1080 Ti GPU and a Xilinx Zynq UltraScale+ ZU9EG FPGA. The FPGA platform was used to stand-in for ASICs, as they are commonly used to prototype ASICs and have implementation characteristics that allow prediction of ASIC performance. Due to the usage of an FPGA instead of an ASIC and the difference in price of the three hardware platforms, the author measured performance relative to a SHA256D baseline implementation for each of the three setups. The author proposed a metric called the ASIC-disadvantage metric, which is the ratio of the relative performance loss when going from SHA256D to a multi-hash chain algorithm when implemented using an FPGA compared to the loss in performance when implemented on other platforms. In order to demonstrate ASIC-resistance, a tested algorithm must have an ASIC-disadvantage of greater than one, and ideally significantly greater than one.

In the experiment, the author implemented the following algorithms on an FPGA:

- a fixed hash chain sequence using a fixed pipeline,
- a hash chain sequence that varies per block, using a crossbar that connects the hash cores, and
- a hash chain sequence that varies per nonce, using a crossbar that connects the hash cores.

It was shown that both the fixed chain algorithm and a version of the hash chain sequence that varies per block, but where no two hash functions were reused, were not able to achieve an ASIC-disadvantage of 1.0. However the version of variable sequence hash chain per block, where hash functions could be reused, achieved an ASIC-disadvantage of 1.33 relative to GPUs. In the case of variable sequence hash chain per nonce, the experiment showed a significant performance loss for GPUs due to control flow divergence and was therefore not able to achieve an ASIC-disadvantage of 1.0.

In the article, the author describes implementing the variable sequence hash chain algorithm (with 15 hash functions) using a 16x16 crossbar to connect the hash cores. This introduced a small overhead in resource usage, while offering low latency and high bandwidth. However, if we design an algorithm that uses hundreds of different hash functions, the resource overhead used up by a crossbar grows at an $O(n^2)$ rate, where n is the number of hash functions. This overhead quickly becomes unsustainable. In such a situation an ASIC must instead use a more scalable network topology, such as a ring, and thus sacrifice latency and bandwidth. For CPUs and GPUs, increasing the number of hash function should only increase the program size and have an insignificant effect on performance.

ASIC mining hardware has been available for fixed-chain multi-hash algorithms since 2016. As of writing, the highest performing X11 miner is the Bitmain Antminer D7[3], which achieves a hash rate of 1.157 TH/s.

A miner which was advertised as the OW1 ASIC[4], was released in late 2019, which marketed an X16R hash rate of 182 MH/s, while consuming 1400 watts. It is speculated by the community that these OW1 ASICs have an FPGA-like, reconfigurable architecture.

2.1.4.2 Memory-hard

Memory-hard PoW algorithms are intentionally designed to require a large memory capacity or to be bottlenecked by memory bandwidth, and usually a combination of both. The rationale for this is that fast on-chip memory is typically SRAM-based, which is low in capacity, therefore if the algorithm requires access to a dataset (typically sized at sev-

[3]Bitmain Antminer D7:
https://shop.bitmain.com/product/detail?pid=00020210721103817933De3r7nxP06AD
[4]OW1 ASIC:
https://en.cryptonomist.ch/2019/09/17/mining-ravencoin-hashrate/

eral gigabytes) that is much larger than the amount of SRAM that can fit onto an ASIC, an external memory is needed. However, external memory bandwidth is limited by the number of I/O pins on the chip. The I/O limitation was thought to level the playing field between ASICs and off-the-shelf hardware such as CPUs, GPUs, and FPGAs, as it was thought that it would be impractical to build an ASIC with an order of magnitude more I/O pins than is already generally available.

Examples of memory-hard PoW algorithms include, Ethash [8], Equihash [5], and CuckooCycle [32]. Ethash is currently the second most popular PoW algorithm, behind Bitcoin's SHA256D. A detailed description of Ethash can be found in Appendix A.1.

GPUs are most commonly used for mining memory-hard PoW. However, recently both FPGA and ASIC miners for Ethash became available. Commercial implementations of Ethash using the Xilinx Virtex Ultrascale+ HBM series of FPGAs[5] are able to outperform GPUs[6] in efficiency by 1.8x. The Jingle Mining Jasminer X4[7] is an ASIC composed of two silicon dies, a logic die and a DRAM die, which are vertically stacked. The device is able to achieve 1TB/s memory bandwidth while consuming very little energy, and outperforms FPGA implementations by 3.1x in efficiency.

2.1.4.3 Algorithmic

Algorithmic PoW algorithms work by generating a new instruction-based random program for each block interval. The randomly generated programs are typically designed to stress as many components of a standard x86 CPU as possible. Branch instructions are also used so that a branch predictor is required to achieve good performance. This type of PoW is CPU-biased and can not be efficiently mined by GPUs, FPGAs or ASICs.

The only example of an algorithmic PoW that is used in practice is RandomX [30], which is the PoW for the Monero cryptocurrency. A detailed description of RandomX can be found in Appendix A.2.

[5]Xilinx Varium C1100 card (with 460GB/s of memory bandwidth) achieves ~73MH/s in Ethash while using ~81W.
https://github.com/todxx/teamredminer

[6]NVIDIA RTX 3070 (with 448GB/s of memory bandwidth) achieves ~60MH/s in Ethash while using ~120W.
https://www.nicehash.com/profitability-calculator/nvidia-rtx-3070

[7]Jasminer X4 achieves ~64MH/s while using ~23W.
https://www.jinglemining.com/pages/about-jasminer-x4

2.2 FPGA runtime reconfiguration

Modern FPGAs can be dynamically reconfigured at runtime without needing to be power-cycled. Most basic resources in the FPGA fabric can be selectively and partially reconfigured without disrupting the ongoing operation of the surrounding logic. FPGA configuration data are called bitstreams, whereas configuration data that configures only a portion of the FPGA are called partial bitstreams. In this section, we will examine the FPGA architectures from Xilinx and Intel in regards to their runtime reconfiguration capabilities, and outline some previous work in the area of runtime reconfiguration.

2.2.1 FPGA hardware

2.2.1.1 Xilinx Virtex Ultrascale+

The Xilinx Virtex Ultrascale+ family of FPGAs is manufactured on TSMC's 16nm technology node and has been available since 2016. These devices are constructed from one to four silicon dies called super logic regions (SLRs). The SLRs are stacked on top of a single passive silicon substrate and are electrically connected to each other using through-silicon vias (TSVs) and wires in the silicon substrate. The High-Bandwidth Memory (HBM) variant of these devices also contains up to two DRAM dies, which are co-packaged in the same manner alongside the SLRs. The amount of routing between SLRs is limited, so care must be taken when partitioning large designs across SLRs. The maximum number of LUTs contained within a Virtex Ultrascale+ SLR is ~440K.

These FPGAs can be reconfigured at runtime by writing a partial bitstream to the *Internal Configuration Access Port* (ICAP). The time taken to reconfigure a module is equal to the time it takes to load the partial bitstream. For Virtex Ultrascale+, the ICAP has a maximum clock frequency of 200MHz and is 32-bits wide, giving a throughput of 800MB/s, meaning that a 20MB partial bitstream, which is large enough to configure an entire SLR, can be loaded in 25ms. Each SLR has a separate ICAP, one of which is designated the master ICAP. The master ICAP can be used to reconfigure any SLR at a reduced clock frequency of 125MHz (500MB/s). However, each ICAP can configure its local SLR at the maximum clock frequency of 200MHz [38] and in parallel.

Chips from this family such as the VU9P, VU13P, and the HBM-enabled VU33P, VU35P, and U55N are all popular in the cryptocurrency mining community. There have been several mining targeted boards utilizing these chips, including the SQRL FK33[8], SQRL JC-series, Bittware CVP-13[9], derivatives of the Xilinx VCU1525[10], and most recently the Xilinx Varium C1100 blockchain accelerator card[11].

2.2.1.2 Intel Stratix 10

Intel's Stratix 10 family of FPGAs is manufactured on the Intel 14nm node and consists of a single monolithic logic die, where the FPGA fabric resides, and several smaller surrounding dies which can be memory, transceivers, or ASICs. The separate dies are connected using a packaging technology called Embedded Multi-Die Interconnect Bridge (EMIB).

The FPGA fabrics of these devices consists of an array of identical logical sectors [18], such that any module that fits completely within a sector can be relocated to any other sector without the need for recompilation. This feature can be used to implement designs that utilize course-grained module relocation. The FPGA fabric architecture is very suitable for deeply pipelined designs due to the *hyper-flex* architecture, which adds millions of pipeline registers within the switch fabric itself, allowing the designer to be more liberal with pipeline register usage.

There is one dedicated *secure device manager* (SDM), which handles loading partial bitstreams. However, the maximum clock frequency for the 32-bit configuration bus is 125MHz. The relatively slow configuration speed (especially if we consider parallel ICAP usage) coupled with the fact that the maximum bitstream size for Stratix 10 is similar to that of Virtex Ultrascale+, means that reconfiguration of Stratix 10 devices is much slower than for Virtex Ultrascale+ devices.

Compared to Xilinx FPGAs, there are not many Intel FPGAs that are currently being used for mining.

[8]Squirrels Research Labs LLC is defunct as of 2021, their product page is no longer on the web.
[9]https://www.bittware.com/cvp-13/
[10]Such as the SQRL BCU1525, the Osprey Mining ECU200, and the TUL BTU9P. https://shop.fpga.guide/products/btu9p-by-tul?variant=14315446861936
[11]https://www.xilinx.com/products/accelerators/varium/c1100.html

2.3 Related Work

We do not know of any other academic work that has been done regarding the application of FPGA runtime reconfiguration to proof-of-work algorithms. However, there has been much previous work in the field of FPGA runtime reconfiguration.

2.3.1 Applications of runtime reconfiguration

With current FPGA hardware, it takes a significant amount of time, depending upon its size, to reconfigure a module — typically on the order of several milliseconds. Therefore there is a significant overhead involved when applying runtime reconfiguration. Still, there are many applications that greatly benefit from runtime reconfiguration, as discussed in the literature.

One common use case for runtime reconfiguration is for image or video processing pipelines, where large amounts of data are streamed through multiple filter stages. In real-time camera applications, where perhaps image quality changes throughout the day, different filters may be needed. A static design, with all the potential filters already baked in, consumes more power and requires a larger FPGA, when compared to a runtime reconfigurable approach, where only the filters that are needed are configured on the FPGA. In [4] the authors implemented two video processing filters on a Virtex-4 FPGA and showed an FPGA resource saving of almost 50 percent.

In [24] the authors demonstrated a time-shared computer vision pipeline, where multiple different processing pipelines can be time-multiplexed to serve different users. Their processing pipeline consists of several identically-sized reconfigurable partitions that are connected to a crossbar interconnect. Each reconfigurable partition can implement several reconfigurable processing modules, such as edge detection, colour-based object tracking, and template tracking. Up to two 1080p video frames are processed at a time, which are streamed from one reconfigurable module to another. Once the two frames have finished processing, the reconfigurable modules are replaced by modules that are needed for a different pipeline. They showed that their time-multiplexed approach is capable of processing a 1080p video through three different pipelines at 30Hz, when implemented on a Xilinx ZC706 board, with a Zynq FPGA.

In [21] the authors presented a scalable H.264/AVC de-blocking filter that uses runtime reconfiguration to adapt to changes in the video resolution or frame-rate. They did this to improve throughput and save power by selecting the most performant processing implementation for each situation.

Another common application for runtime reconfiguration is software-defined radio (SDR), which is able to operate across multiple radio standards. Traditional custom ASICs with fixed function processing units are not flexible enough to implement SDR. However, purely software-based solutions have high latency and therefore can not be used for some applications. In [26], the authors present an *end-to-end multi-standard OFDM transceiver architecture*, that uses FPGA runtime reconfiguration to support rapid switching between three different standards: IEEE 802.11, IEEE 802.16, and IEEE 802.22. To avoid link disruption, the incoming data is buffered in memory during reconfiguration.

In [25], a high performance DES encryption implementation is presented that takes advantage of FPGA runtime reconfiguration by specialising the encryption circuit on a key-by-key basis. For symmetric key encryption, typically the encryption key, is the same for the entire connected session. By substituting the key into the circuit as a constant, the circuit can be greatly simplified. The author presents a tool *JBits,* which is capable of creating and modifying bitstreams with LUT contents that are specialized to a certain key. By this method, they observed better performance than a DES ASIC.

2.3.2 Runtime reconfiguration methodology

In [22], the authors describe three styles of layout for reconfigurable modules; island-style, slot style and grid style. In an island style layout, the FPGA is divided into islands that are capable of containing any reconfigurable module. However this method leads to under-utilization of resources as islands must be sized according to the largest module. In a slot-style layout, the FPGA is divided along one dimension (e.g. horizontally) into smaller slots where reconfigurable modules are allowed to take up multiple slots. The grid layout is similar to the slot layout with the difference being that the FPGA is divided in two dimensions creating a grid of reconfigurable regions.

In [2], the authors present a tool for module relocation, that is the ability to move a pre-implemented module from one location on the FPGA fabric to another. This can be done without pre-defining the layout of boundaries for each reconfigurable partition as in [22]. This allows for denser packing of reconfigurable modules and thus better utilisation of FPGA resources.

Chapter 3

Design exploration

3.1 X16R algorithm

The X16R algorithm is a PoW algorithm that was primarily used by the cryptocurrency RavenCoin [11], which had an average block interval of one minute. The algorithm was used from January 2018 to May 2020, after which RavenCoin switched to a memory-hard PoW called KawPow [19], due to the threat of ASIC and FPGA miners outcompeting GPU miners [6].

3.1.1 Definition

The X16R algorithm is a hash algorithm, consisting of the chained application of these sixteen hash functions: (0) Blake, (1) BMW, (2) Groestl, (3) JH, (4) Keccak, (5) Skein, (6) Luffa, (7) Cubehash, (8) Shavite, (9) Simd, (A) Echo, (B) Hamsi, (C) Fugue, (D) Shabal, (E) Whirlpool, and (F) SHA512. With the exception of Whirlpool [31] and SHA512 [29], each hash function was submitted to the NIST SHA3 competition and passed to at least round 2 [33]. Each hash function is assigned a hexadecimal identifier (0x0 - 0xF). The order in which the hash functions are applied is determined by the last sixteen nibbles of the hash of the previous block header. For example, if the previous block was:

`0x0000006b444bc2f2ffe627be9d9e7e7a0730000870ef6eb6da46c8eae389df90`

Then reading the right-most sixteen hexadecimal digits of the hash from left to right, we find the hash chain ordering:

```
0xda46c8eae389df90
```

which translates to:

```
Shabal -> Echo -> Keccak -> Luffa -> ... -> Simd -> Blake
```

The X16R hash function is described by the following pseudo-code:

```
hash_functions = [
    blake, bmw, groestl, jh, keccak, skein, luffa, cubehash,
    shavite, simd, echo, hamsi, fugue, shabal, whirlpool, sha512
]
def x16r(header):
    order = get_last_16_nibbles(header.prev_hash)
    midstate = header
    for i in order:
        midstate = hash_functions[i](midstate)
    return midstate
```

Note that the input into the first hash function is the 80-byte block header, which is the typical header size for many blockchains. However, as all hash functions in X16R produce a 64-byte digest, all subsequent function applications take a 64-byte input.

3.1.2 Properties

As the ordering of the hash functions is determined by the hash of the previous block header, we can assume the that ordering is effectively uniformly random. We note that there is the possibility of a hash function occurring more than once in the chain, we call these occurrences *repetitions*.

The probability of there being no repetitions in an X16R chain is the number of chains where each hash function is unique, i.e. 16!, divided by the total possible number of chain permutations i.e. 16^{16}:

$$\frac{16!}{16^{16}} \approx 1.134 \times 10^{-6}$$

We can see that the case with no repetitions is extremely unlikely, around one in a million. This presents a challenge for hardware implementation, as a design with only one hash core per function will almost always be bottlenecked when that core is needed more than once to complete processing the X16R chain.

Figure 3.1 shows the probability of an X16R chain having a function that occurs N times, and all other functions occurring at most N times.

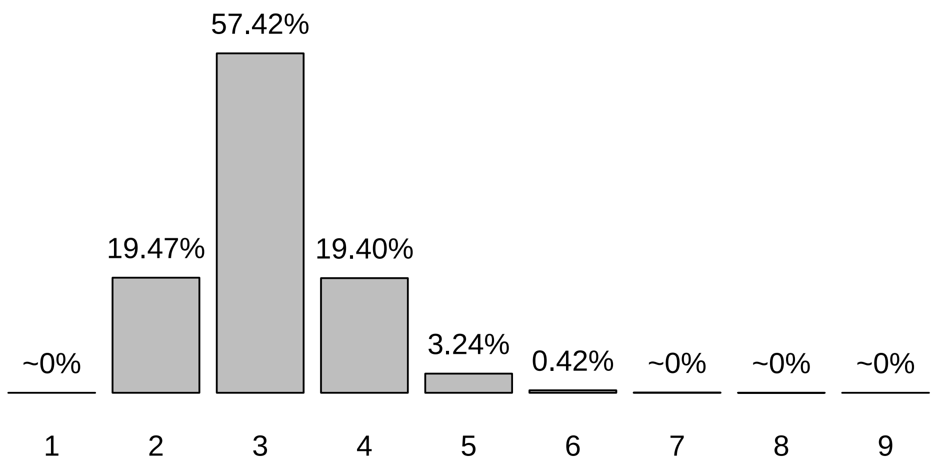

Figure 3.1: Probability of X16R chain having a function that occurs N times, and all other functions occurring at most N times

In the following sections, we will consider three different types of FPGA-based architectures for implementing the X16R algorithm.

3.2 Static design

First we consider the naive approach of a static design, in which sixteen different hash cores are instantiated on a single FPGA and are connected to each other through an in-fabric network. A suitable means of transferring the output of one core to the input of the next is therefore sought. As each hash core is fully unrolled, they can produce one result for every clock cycle. Therefore, the communication between each subsequent core in the chain is a continuous stream. One might initially consider a crossbar switch, which allows non-blocking communication between any pair of cores. However, a crossbar has a wiring complexity of $O(n^2)$ and requires 578 connections for a 17x17 bidirectional crossbar (17x17 because there are 16 hash functions and 1 input/output port). If we assume that each connection is 512-bits wide (a requirement for fully unrolled hash cores), we would require 295,936 wires, which is a significant demand on an FPGA's routing resources. Moreover, most of the switches and wires are not utilized during the processing of any particular instance of X16R. However, the network complexity is not the main concern when implementing a static design. In Figure 3.1, we see that most of the time, there are repetitions in the chain. One can calculate that, on average, a static design with only one core per hash function is bottle-necked at 34.4% of its peak throughput. This is because only a few cores are highly utilized (due to being needed multiple times in the chain), while most other cores are under-utilized (waiting for data or not needed). Another challenge is the amount of FPGA resources required to implement the fully static design. It is possible to fit rolled-up versions of each hash core in a modern FPGA. We see in [9] an implementation of X15R (X16R without Whirlpool) using a crossbar switch on a Xilinx ZU9EG device, however the performance is poor, achieving only 2MH/s. In order to achieve higher performance we need to unroll the hash cores. However, an X16R design consisting of sixteen fully unrolled hash cores will require an extremely large FPGA.

If we take the resource utilization numbers from our own hash core library, described in Section 4, we require a device with over 2.3M 6-input LUTs (not accounting for the crossbar) in order to accommodate one instance of each function. By comparison, the largest Xilinx Ultrascale+ device that is obtainable by miners, the VU13P, has 1.7M LUTs. An even larger device, the VU19P with 4M LUTs, is sold by Xilinx, however, they are likely to be too few in number and too expensive to be financially viable for miners. In order to meet the area requirements for a fully static design, one may attempt to team multiple FPGAs and spread the hash cores across them. However, it is difficult to extend the crossbar off-chip due to the lack of wires (parallel approach) or lack of ser-des bandwidth (serial approach).

3.3 Dynamic full chain

In order to avoid the shortcomings of the static design we may attempt to apply dynamic partial reconfiguration in order to modify the hardware such that only the hash functions that are needed for each block interval are included. By doing this, we are able to configure multiple instances of hash cores for functions that are repeated in the hash chain, thereby avoiding the bottleneck present in the static design. We can also avoid implementing a full crossbar by reconfiguring the connections between the cores to suit the data-flow for the current chain order.

While this method eliminates the bottleneck caused by repetitions, we still need an extremely large FPGA. In the worst case, when sixteen of the most resource expensive hash functions (e.g. cubehash) are needed by a particular problem instance, we would need an FPGA with over 5 million LUTs. Another problem is the number of permutations of the chain that must be pre-compiled. It is not possible to compile and store all permutations of the chain. A practical approach may be to break up the FPGA area into slots and break up the hash chain into sub-chains, i.e. chains of two or three hash functions. Sub-chains can be compiled into their respective slots, which are then stitched together at runtime through a runtime reconfigurable network.

3.4 Time-sliced dynamic sub-chains

In order to implement X16R efficiently on smaller devices, we consider a time-sliced approach that is capable of handling every permutation of the X16R chain. Instead of having one contiguous chain, which is configured onto the FPGA at the start of each block interval, we divide the chain into sub-chains, of which only one is configured onto the FPGA at any time. The outputs of the sub-chains are called mid-states which are buffered in memory and are subsequently fed as inputs to the next sub-chain after it has been configured. This method incurs a substantial reconfiguration overhead, as the FPGA must be reconfigured 16/(*subchain length*) times before the result can be returned. In order to lessen the impact of the reconfiguration overhead, we maximize the time spent computing hashes, by batching up as many mid-states in memory as possible before moving to compute the next sub-chain.

We consider two implementation methods, *Single hash Function per Reconfiguration* (SFR) and *Multiple hash Functions per Reconfiguration* (MFR). SFR divides the chain into 16 sub-chains with only one hash function per sub-chain. This leads to under-utilization of FPGA resources when small functions are configured and higher reconfiguration overheads per batch.

A more efficient method (MFR) is to include multiple hash functions per sub-chain. However, the number of permutations that must be compiled per sub-chain is exponential in the number of functions in the sub-chain. As such, we only consider chains where every sub-chain has exactly two hash functions, as these sub-chains take less compile time than sub-chains of greater size. We can reduce the number of permutations that need to be compiled from $16^2 = 256$ to 136 by introducing a simple multiplexer, such that the order of each function in the sub-chain can be selected at runtime. There are $\binom{16}{2} = 120$ pairs of distinct hash functions (order does not matter due to the multiplexer), and an additional 16 pairs of duplicated hash functions. Therefore there are 136 possible pairs that must be compiled.

3.5 Overheads

3.5.1 Intra-batch overhead

There is an intra-batch overhead introduced by reconfiguring the FPGA between sub-chain computations. The amount of overhead is related to the chosen batch size, the throughput of the hash cores and the time to configure the sub-chain. If we assume a constant reconfiguration time for every partial bitstream, the reconfiguration overhead, which is the proportion of time wasted during reconfiguration, can be estimated as follows:

Let t be the time taken to process an entire batch of X16R hashes, n be the number of hashes that is processed in a batch, f be the raw throughput (hashes per second) of the sub-chain cores, r be the time required to configure a sub-chain, and s be the number of sub-chains to be configured. Then:

$$t = s \times (n/f + r)$$

And the intra-batch overhead is the complement of the amount of the time spent computing hashes over the total batch time:

$$\text{intra-batch overhead} = 1 - \frac{s \times n/f}{t} = 1 - \frac{n}{n + fr} = \frac{fr}{n + fr}$$

The maximum time required to reconfigure a single SLR on a Xilinx Virtex Ultrascale+ device is 35.41ms (obtained using the maximum bitstream size as listed in [37] and assuming the maximum ICAP frequency of 200MHz). For our hash core implementations, we targeted a frequency of 625MHz and each core is capable of producing a 64-byte output every clock cycle, giving a throughput of 40GB/s. In Figure 3.2 we plot the reconfiguration overhead versus batch size in gibi-bytes.

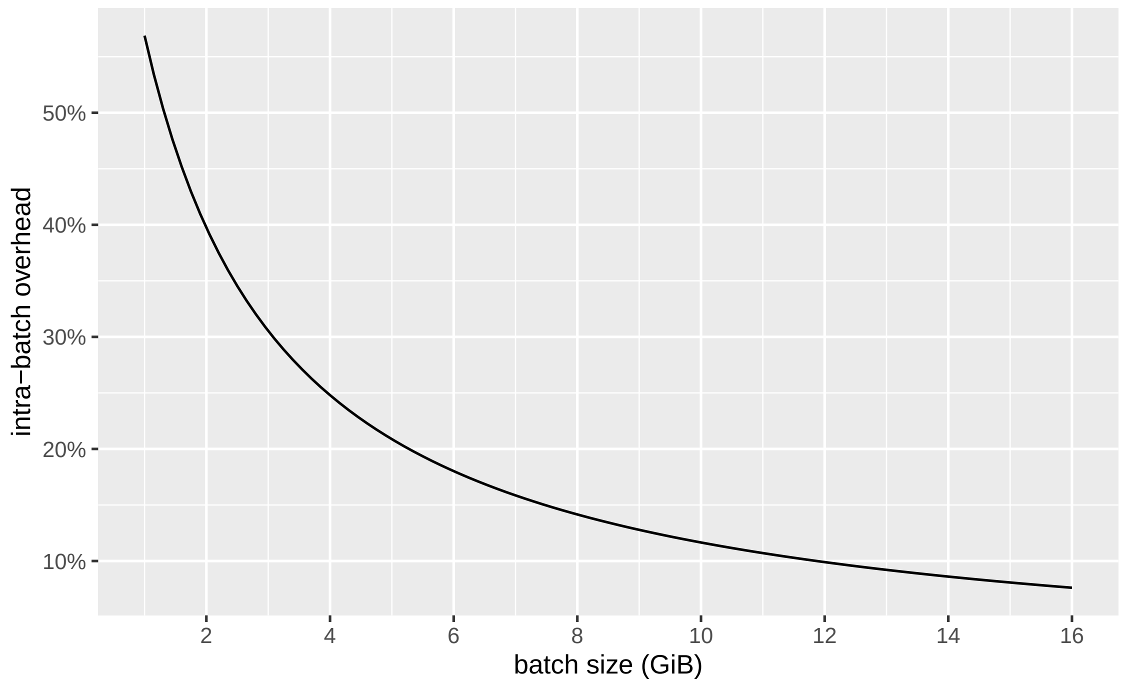

Figure 3.2: Intra-batch overhead versus batch size

Clearly, for a given throughput and reconfiguration time, the batch size should be maximized to minimize the intra-batch overhead.

3.5.2 Inter-batch overhead

As we are considering batched computation, the hashes will be produced in regular intervals, as opposed to continuously. Therefore, if a new block interval begins whilst we are processing a batch, then that entire batch of hashes is discarded. We analyse how much time on average is wasted computing hashes that must be discarded at the end of a block interval and thus find the *inter-batch overhead*.

First we need to determine the distribution of block intervals that end when a miner finds a valid proof-of-work. Recall that an individual miner is tasked with finding an input whose hash gives an output that is numerically less than a particular target L. By design, the hash outputs for X16R (and other PoW hash functions) are effectively random with each output within the range $(0, 2^{512} - 1)$ being equally likely. Thus, the probability of a candidate input giving an output that is less than L is $L/2^{512}$. It follows that the required number of hashes an individual miner needs to compute before finding one that gives an output less than L is geometrically distributed[1] with parameter $p = L/2^{512}$. Since miners perform hashes at a very fast rate, the required time for a miner to find a successful input is well-approximated by an exponential random variable, whose parameter is a function of the miner's hashing rate and the target difficulty. The time taken for a community of miners to find a successful input is the minimum among all the times taken for each individual miner to find a successful input, and in theory this time should also be exponentially distributed, with parameter proportional to the inverse of the product of the community's hash rate (also called the network hash rate) and the target. The target is adjusted at regular intervals to ensure that this parameter remains constant even if the community's hash rate changes.

Let T be the block interval and denote the average by $\mathbb{E}(T)$. T follows the exponential distribution with parameter λ where $\lambda = 1/(\mathbb{E}(T))$. Let $t = t(n, f, r, s)$ be the batch time for our chosen batch size, where n is the number of hashes that is processed in a batch, f is the raw throughput of the sub-chain cores, r is the time required to configure a sub-chain, and s is the number of sub-chains to be configured. Let N be the number of batches that we process during the block interval T, so $N = \lfloor T/t \rfloor$. Using properties of the exponential distribution, we know that $T/t \sim \text{Exp}(t\lambda)$, and we know that $N = \lfloor T/t \rfloor$ follows the geometric distribution (taking values $0, 1, 2, ...$) with parameter $p = 1 - \text{Exp}(-t\lambda)$. Thus the amount of time wasted during a block interval on a batch that we must discard is $T - Nt$, and the proportion of time wasted during a block interval, the inter-batch overhead, is given by $1 - (Nt)/T$. It is difficult to analytically express the expectation of this random variable, so we instead compute it by simulation.

In Figure 3.3, we plot the inter-batch overhead for batch times between 0.1 and 10 seconds. There are two plots, one for an average block interval of 60 seconds (which we are most concerned with) and one for 12 seconds (which is Ethereum's block interval and is the

[1]This is equivalent to flipping a biased coin with a $L/2^{512}$ chance of success. The number of trials until success follows the geometric distribution.

shortest among popular PoW blockchains).

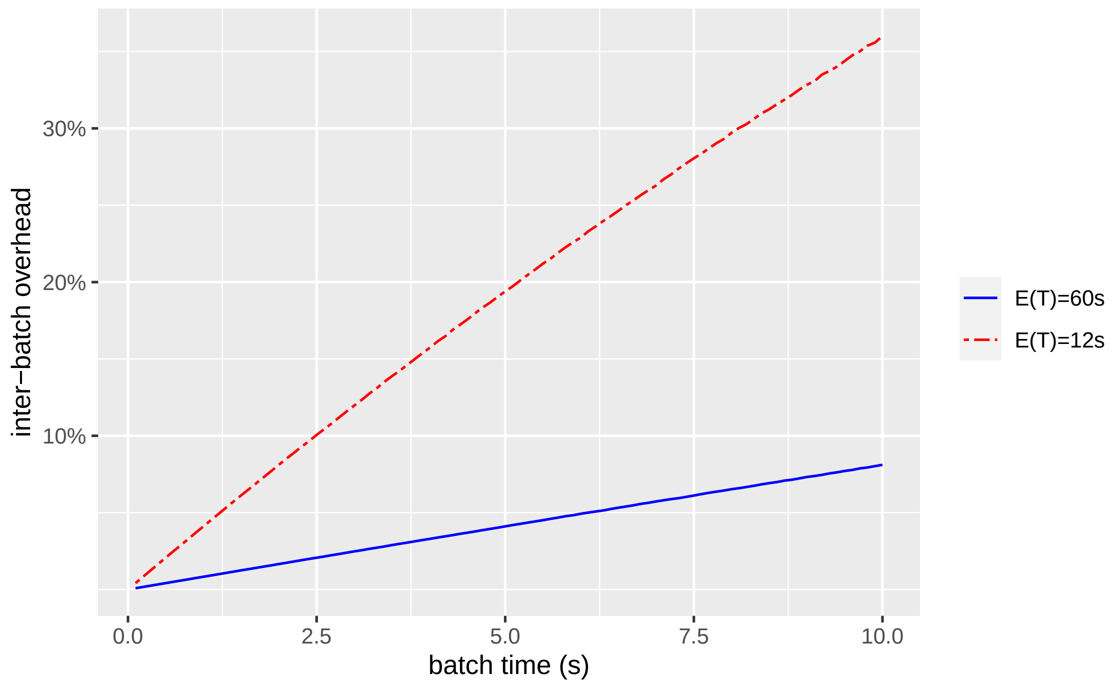

Figure 3.3: Inter-batch overhead

3.5.3 Estimated performance

We can estimate the performance of our time-sliced X16R design by taking into account both the intra-batch and inter-batch overheads. The combined overhead is given by the sum of the times wasted within each batch in addition to that wasted computing the final batch, divided by the block interval. We can express the total overhead as follows:

$$\text{total overhead} = 1 - \frac{N \times (t - rs)}{T}$$

Where $N \times (t - rs)$ is the total time spent computing undiscarded hashes and T is the block interval.

We can estimate the hash rate by multiplying the proportion of time that is used for computing hashes (excluding the discarded batch) with the raw hash throughout of a sub-chain divided by the number of reconfigurations that are required per batch.

$$\text{hash rate} = (1 - \text{total overhead}) \times \frac{f}{s} = \frac{nN}{T}$$

In Figure 3.4 we plot the estimated hash rate for various batch sizes. The blue lines are plots for average block intervals of 60 seconds, while the red lines are for average block intervals of 12 seconds. The solid and dashed lines represent plots for 8 and 16 reconfigurations per batch respectively.

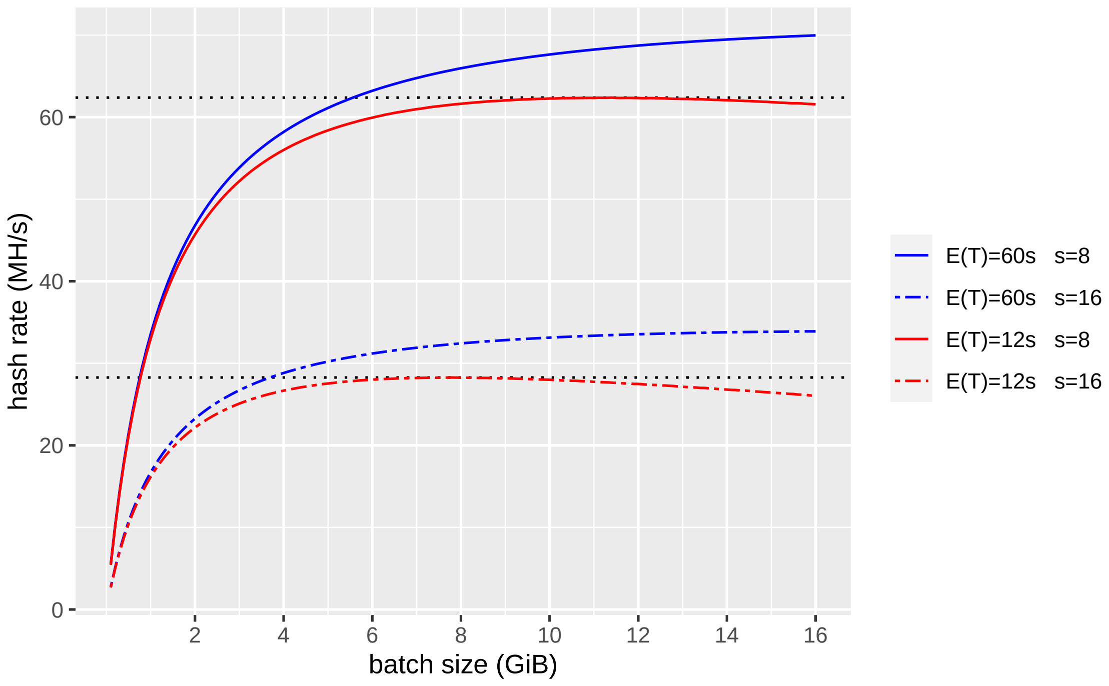

Figure 3.4: Estimated hash rate vs batch size

We see that for SFR, where there are 16 reconfigurations per batch, we should be able to achieve on average approximately 28.82MH/s, or 32.43MH/s for batch sizes of 4GiB and 8GiB respectively. Note that there are diminishing performance gains as we increase the batch size and as evidenced by the $\mathbb{E}(T) = 12s$ plot; the performance actually decreases above a certain batch size. This should be noted when selecting an FPGA-based mining platform, as one with a larger memory, which is capable of supporting a larger batch size, may not always be desirable. We also note that our time-sliced approach does not suffer greatly in performance when used with a blockchain with a very short average block interval of only 12 seconds.

Chapter 4

Hash functions implementation

In order to implement any X16R design, we must first obtain hardware implementations of the sixteen hash functions that are required. We note that in [9] the author used fourteen (out of sixteen) hash functions from [13], which were written in VHDL and optimized for FPGAs. However, for cryptocurrency mining, extra optimizations are possible that take advantage of the fixed input size (typically 640-bits or 512-bits) and allow us to fully unroll the hash function. We decided to implement our own hash functions to take advantage of these extra optimizations. To speed up the time to code these hash functions, we investigated the suitability of using high-level-synthesis (HLS) tools to automatically pipeline the designs.

4.1 Is HLS auto-pipelining competitive with handcrafted RTL?

There have been previous studies: [15], and [16], comparing HLS and hardware description languages (HDL) implementations of cryptographic functions. They concluded that the performance of the circuits generated by HLS is comparable with that of HDL designs. However, those studies concentrated on smaller designs with longer initiation intervals (i.e. designs that are not fully unrolled). We investigated whether existing HLS tools are able to compete with hand-optimized RTL when used to implement fully unrolled and pipelined hash functions. Specifically we were interested in leveraging the *auto-pipelining* capabilities of HLS in order to obtain high clock frequencies for the circuits, without sacrificing portability. For our RTL baseline we chose to use **Chisel** [1] and for the HLS candidates we chose **Vitis HLS** [35] and **PipelineC** [20].

4.1.1 Languages

4.1.1.1 Chisel

Chisel is an HDL that facilitates circuit generation within the Scala programming language. Users of Chisel are able to use modern programming abstractions to create parameterizable circuit generators that produce synthesizable Verilog. This improves the designer's productivity and the maintainability of the code. Chisel is fundamentally a HDL, as the designer describes the circuit at the register transfer level (i.e. flip-flops/registers must be explicitly defined).

On initial impressions, one may be concerned about whether the designer gives up any ability to perform low level optimizations when using Chisel. Chisel does have some limitations. One such limitation is that only synchronous circuits can be described. For example, one cannot infer an asynchronous latch using Chisel. Additionally, as of writing, some FPGA-specific primitives cannot easily be inferred, such as *true-dual-port block memories* and *DSP blocks*. However, users can easily integrate hand-crafted Verilog as black-boxes into Chisel to use these resources.

For our purposes, that is, crafting fully unrolled and pipelined hash functions with an initiation interval of one, we are not limited by Chisel's expressivity. There have been several published studies comparing the quality of results (QoR) between circuits generated with Chisel and hand-crafted Verilog. In the original paper describing Chisel [1], the authors concluded that there is no significant difference in results when comparing a *fused multiply-add unit* circuit generated in Chisel with one that was hand-crafted using Verilog. In [17] the authors compared the silicon area (for an ASIC) taken up by a RISC-V processor generated with Chisel with one that was hand-crafted using Verilog. They found that the circuit produced by Chisel was ~4% more area efficient, while only requiring ~47% of the lines of code used to describe their hand-crafted design. In a study [14], that is more relevant to our work, the authors implemented an AES core on a Xilinx Virtex Ultrascale+ FPGA. They found that both resource utilization and power consumption for the circuit generated by Chisel was very close to that of a corresponding Verilog implementation.

4.1.1.2 Vitis HLS

Vitis HLS (previously Vivado HLS), as of writing, is the officially supported HLS tool by Xilinx. Users can describe their circuit using a dialect of C/C++ or SystemC at an algorithmic level and use tool-specific pragmas to steer the HLS compiler in order to meet performance and resource utilization goals. The Vitis HLS compiler has internal timing models of Xilinx FPGA devices and is able to estimate the latency of a circuit before it is synthesised. When the pipelining option is enabled, the tool is capable of automatically inserting registers to meet a user-specified target frequency. We call this capability *auto-pipelining.* While the tool was recently partially open-sourced by Xilinx, the back-end, which handles the device-specific optimizations, is still proprietary. This means that code written in Vitis HLS is not portable to devices from other FPGA vendors.

4.1.1.3 PipelineC

PipelineC is a tool that can compile a subset of the C language to VHDL. For pure functions (i.e. a function with no pointer arguments or global variables), it is able to auto-pipeline the design. It does this by slicing the circuit and using vendor tools to synthesise and determine the latency of each sub-circuit. Registers are inserted and the design is iteratively re-synthesized until the circuit meets the designer's specified frequency requirements. It is also possible to compile the HLS code with the DO_PNR flag, which uses post-place-and-route timings to inform circuit partitioning. However, this increases the C-to-VHDL compile time substantially. Designs written in PipelineC are portable across multiple FPGA vendor tools (e.g. Xilinx, Intel and Lattice).

4.1.2 Designs evaluated

We evaluated three designs with varying expected resource usage from the cryptographic domain: an AES encryption core as a small-sized case (~10K LUTs), a Keccak hash function core as a medium-sized case (~80K LUTs) and a Groestl hash function core as a large-sized case (~300K LUTs). We chose AES as a small-sized case because it is used internally by many hash functions (e.g. inside Groestl, and Echo). Only the main datapath for each algorithm is considered (i.e. designs do not include control logic) and they are designed to have an initiation interval of one (II = 1), i.e. they are able to accept a new input every clock cycle. These designs only consist of bitwise logical operations (AND, OR, XOR, NOT, etc.), shifts, rotates, and lookup tables (ROM). Therefore no DSPs are inferred.

The Chisel designs were coded such that pipeline registers were inserted between as many logical steps as possible without sacrificing code readability. These pipeline stages could be turned on or off depending on compile time parameters. Typical pointer-free C code is used for the Vitis HLS and PipelineC designs. PipelineC compiles the code with an initiation interval of one by default. For Vitis HLS, a *Pipeline II=1* directive was used at the top level, the tool automatically unrolls loops and applies pipelining to all sub-functions. The *ap_control* setting was set to *none* to ensure no control signals were generated and the pipeline was *free-running*.

4.1.3 How results were collected

The designs were compiled on a PC with an Intel i9-9900KS with 8 cores (16 threads) clocked at 5GHz, 64GB of DDR4 3200MT/s RAM and 128GB of SSD swap. We used Vivado/Vitis 2020.2 running on Ubuntu 20.04. We targeted the **xcvu33p-fsvh2104-2-e** device as it is the largest (440K LUTs) single SLR device in the Xilinx Virtex UltraScale+ family. Synthesis was done *out-of-context* (i.e. IO pins were not mapped) to ensure that IO did not constrain the design at high frequencies, and with all other settings left at their defaults. We compiled each design ten times with target clock periods ranging from 2ns (500MHz) to 1ns (1GHz), decreasing in 0.1ns intervals. Target periods were set in Vitis HLS during the *C Synthesis* stage and were carried forward to RTL synthesis and place-and-route. For PipelineC, the target period was set as a pragma in the source code. Once the final VHDL was generated for the set of designs and target periods, the clock constraint was carried forward to Vivado for RTL compilation. For the Chisel designs, we used the same number of pipeline stages until the designs no longer met the target period post-implementation, at which point we manually increased the number of pipeline stages in the source code and recompiled to meet the target period.

4.1.4 AES results

As seen in Figure 4.1, the circuits produced by all three languages were able to achieve nearly the same maximum frequency (i.e. maximum throughput). Chisel and PipelineC achieved 939.85MHz, while Vitis HLS was able to achieve 932.24MHz. Upon inspection of the timing reports, it was noticed that the designs were limited by a pulse-width timing violation corresponding to the maximum frequency of the SRL (Shift Register Lookup Table) primitive. While it may be possible to clock higher by forcing the synthesizer to map shift registers as flip-flops, this is a reasonable frequency for high performance designs implemented on Virtex Ultrascale+ devices.

When observing throughput over area in Figure 4.2, we see that Vitis HLS consistently outperforms the hand-crafted HDL at every target period past 1.9ns and is 13.8% more area efficient at the maximum frequency. There was a large range in throughput/LUT for the PipelineC designs (18.74 to 112.10), however it was 5.7% more efficient than the Chisel solution at the maximum frequency. PipelineC under-performs significantly at the 2ns and 1.9ns targets. However when compiled with the *DO_PNR* flag (which uses post-routing timings to inform the partitioning algorithm), the result falls into line with the Vitis and Chisel solutions.

The throughput over area and achieved frequencies are generally increasing as we decrease the target periods. However, we note that there are dips in the plots due to performance regressions. This is because the tools do not know the optimal way to insert pipeline registers in order to achieve the target frequency while also minimizing area. This leads to some partitions that increase the area of the circuit (e.g. due to high fan-out), which in turn lower the achievable frequency (e.g. due to routing-delay).

4.1.5 Keccak results

The results for Keccak are shown in Figures 4.3 and 4.4. Vitis HLS was able to achieve the best maximum frequency of 922MHz, an improvement over the maximum frequency achieved by Chisel and PipelineC (*DO_PNR*) of 3% and 9% respectively. The throughput/LUT of the Vitis HLS solution is also better than that of Chisel at high frequencies. The throughput and throughput/LUT of the PipelineC designs are consistently below the

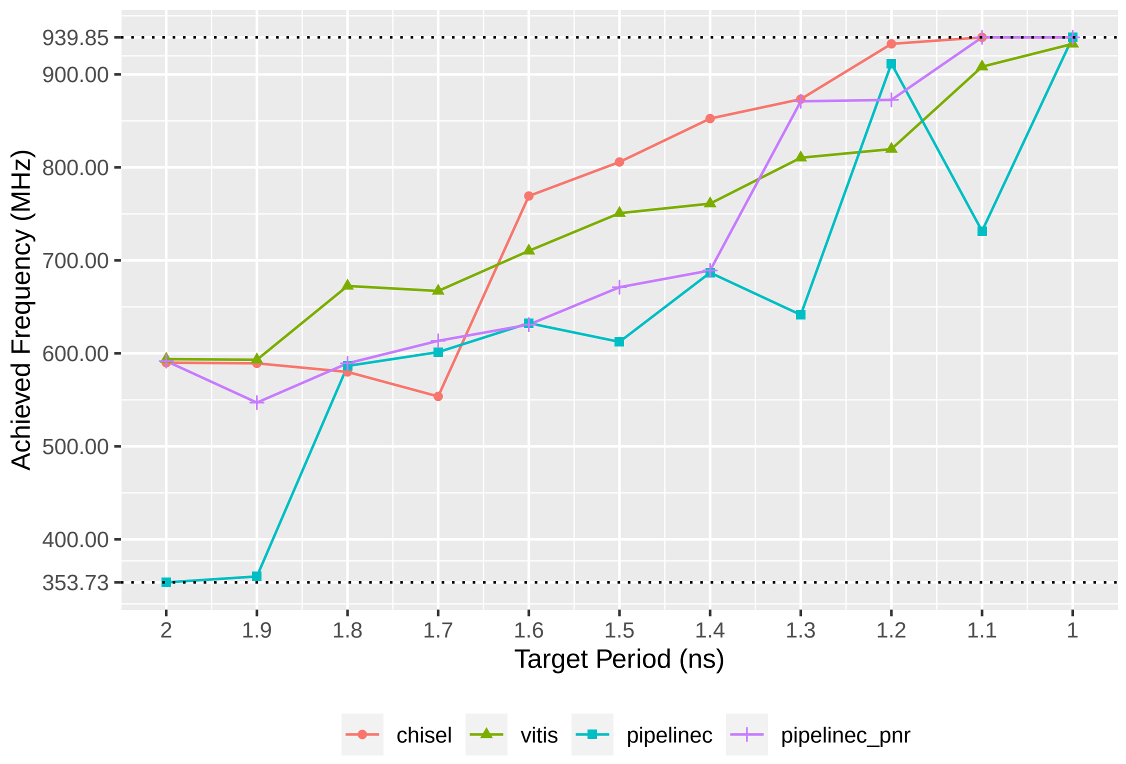

Figure 4.1: AES: achieved frequency

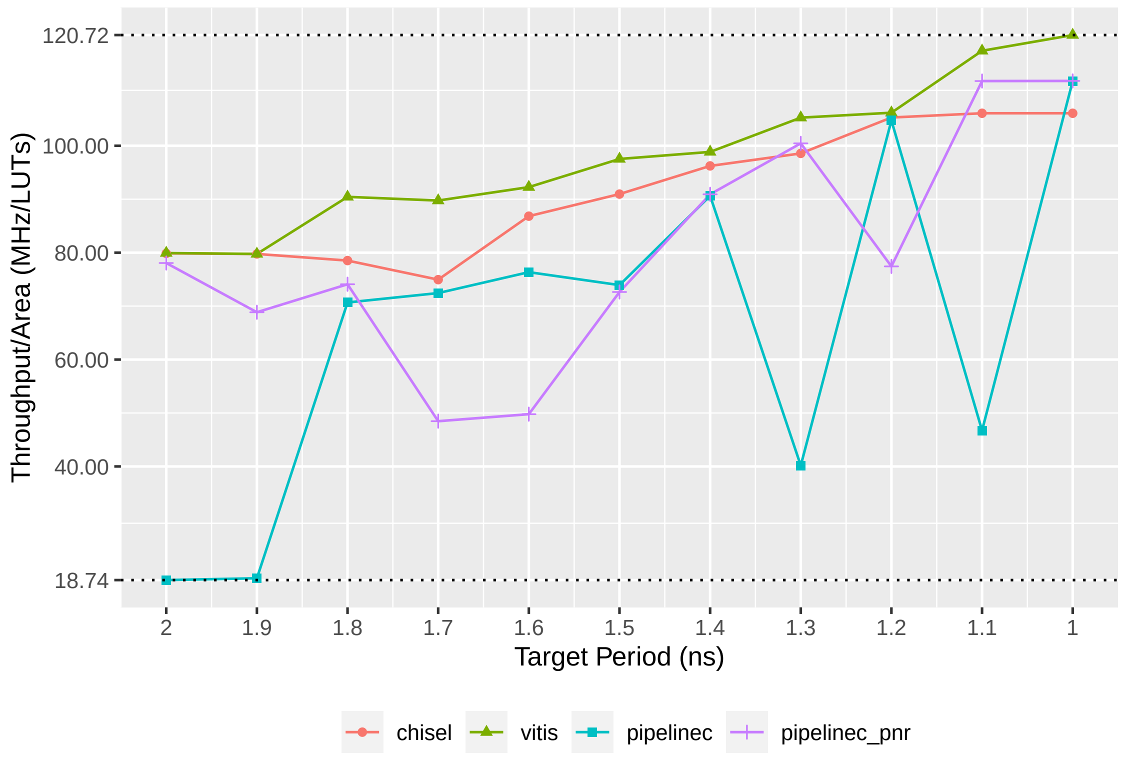

Figure 4.2: AES: throughput/area

others, however if the *DO_PNR* flag is turned on, the total throughput becomes very similar to that of the others, while the best throughput/LUT outperformed Chisel by 56% and is similar to the best Vitis HLS circuit. However, we found that PipelineC (*DO_PNR*) was unable to converge on a circuit that met timing past the 1.2ns target.

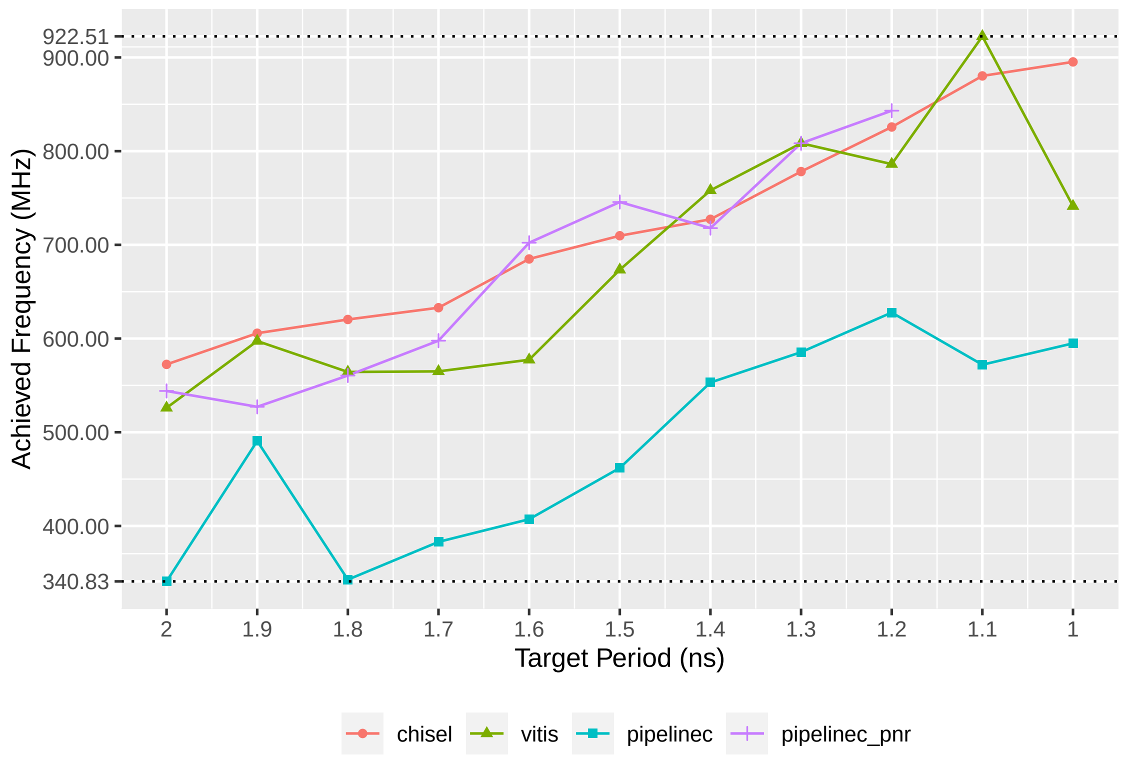

Figure 4.3: KECCAK: achieved frequency

4.1.6 Groestl results

The results for Groestl are shown in Figures 4.5 and 4.6. We see a clear difference, with the hand-crafted Chisel solution outperforming both Vitis HLS and PipelineC at high frequencies. Vitis HLS performed well when the target period was varied between 2ns and 1.5ns. However, we found that performance dropped significantly below the 1.5ns target. The Chisel HDL solution had 20% higher maximum throughput when compared to Vitis HLS. However, Vitis HLS outperformed Chisel in throughput/LUT at lower target frequencies. The PipelineC design consistently lagged behind both Chisel and Vitis HLS and was unable to produce a design that fit into the target device past the 1.3ns target period. We were unable to obtain Groestl data for PipelineC with the DO PNR flag due to the long run time ($>$ 48 hours per run).

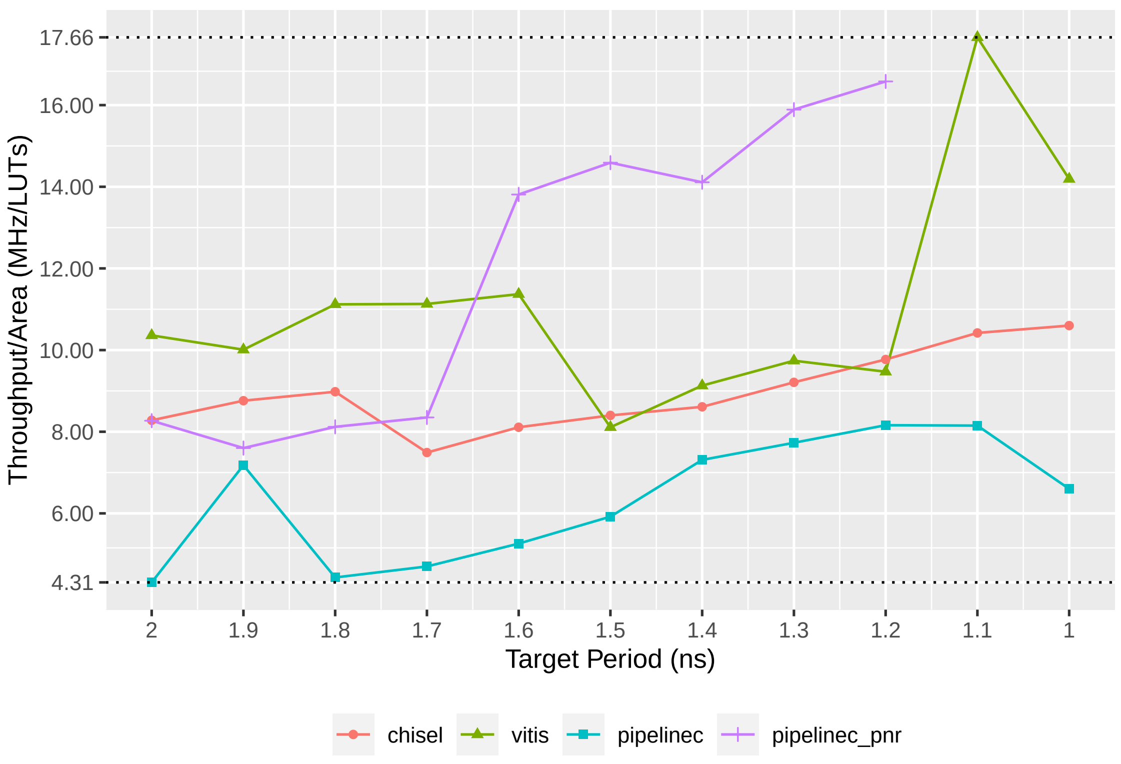

Figure 4.4: KECCAK: throughput/area

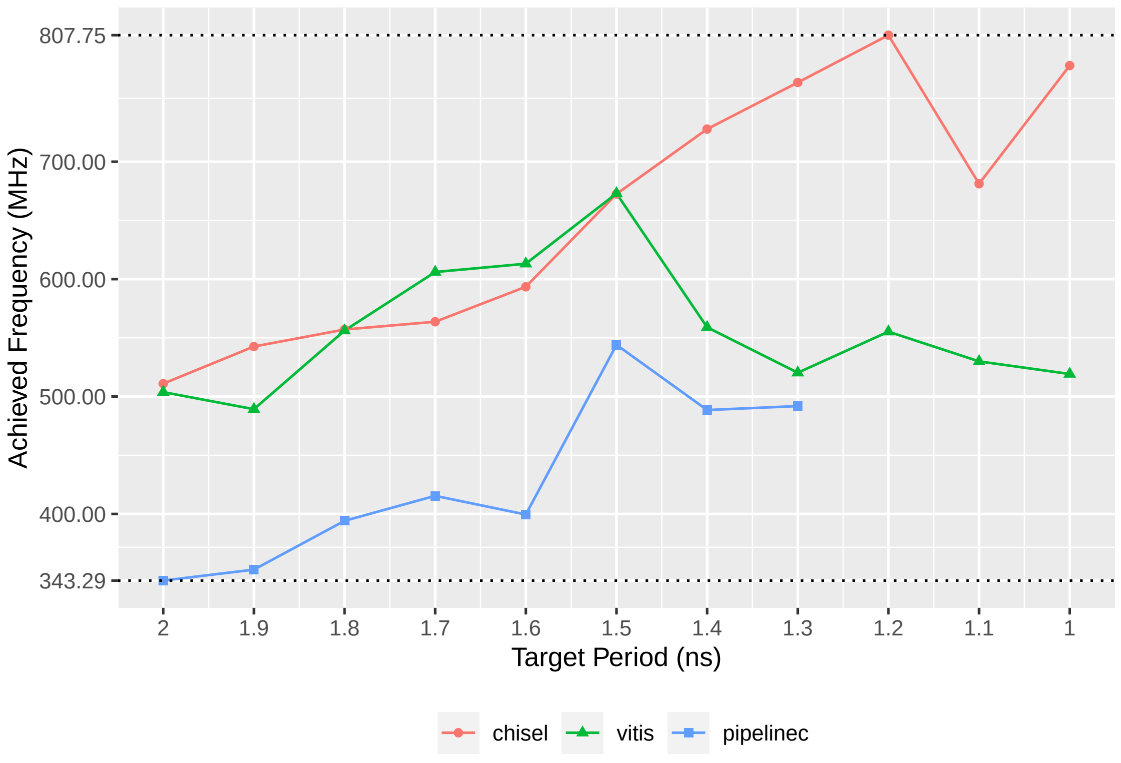

Figure 4.5: GROESTL: achieved frequency

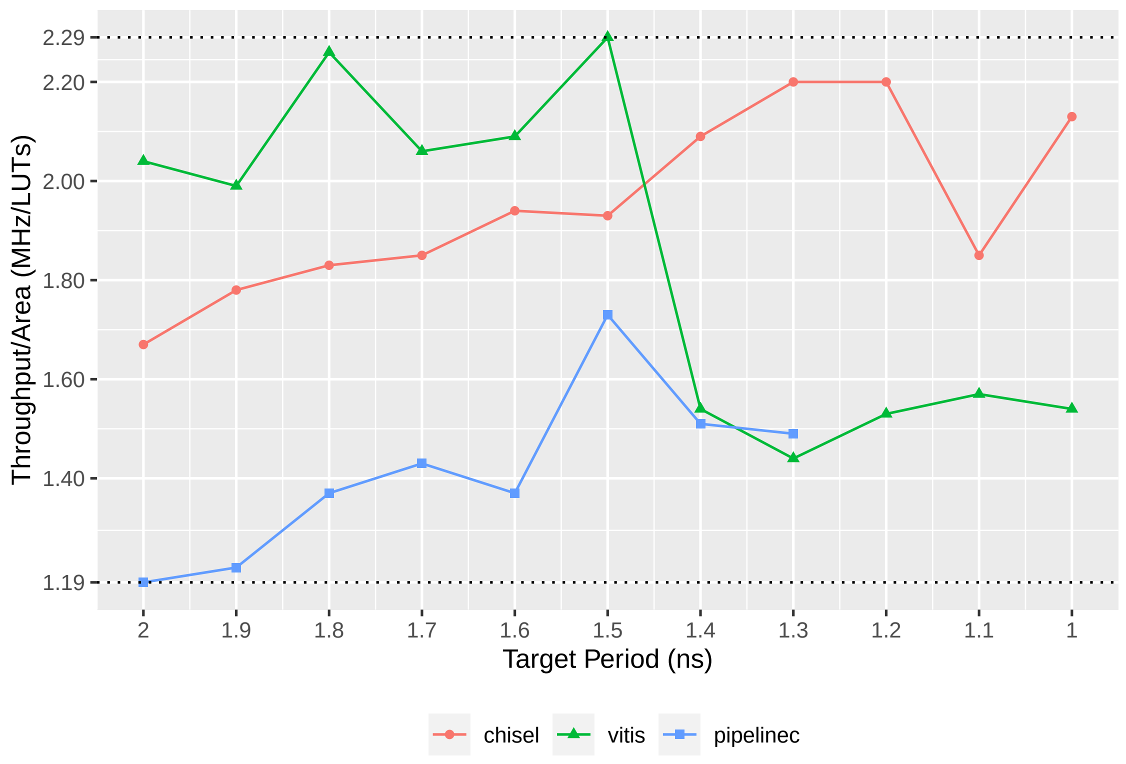

Figure 4.6: GROESTL: throughput/area

4.1.7 HLS vs RTL conclusion

The HLS tools performed well when implementing the AES and Keccak algorithms. They were able to match the HDL solution in terms of total throughput and had superior area efficiency at the highest frequencies. However, for the larger Groestl design, the hand-crafted solution was clearly the better choice when targeting frequencies above 666MHz. This is reasonable to expect, as the auto-pipelining tools do not have an accurate estimate of where the LUTs will be placed on the device (especially when there is significant routing congestion), and Groestl uses far more of the available LUTs relative to AES and Keccak.

Through this comparative study, we conclude that both PipelineC and Vitis HLS offer comparable or better performance than hand-crafted RTL when implementing fully unrolled small- to medium-sized circuits and when targeting frequencies between 500MHz and 666MHz.

4.2 Hash core library implementation

We implemented the sixteen hash functions: (0) Blake, (1) BMW, (2) Groestl, (3) JH, (4) Keccak, (5) Skein, (6) Luffa, (7) Cubehash, (8) Shavite, (9) SIMD, (A) Echo, (B) Hamsi, (C) Fugue, (D) Shabal, (E) Whirlpool, and (F) SHA512. For our X16R implementation, we were targeting frequencies between 450MHz (which is the native frequency of our memory controller) to 625MHz (above which the Xilinx AXI-based IP ecosystem has trouble meeting timing). Therefore, we chose to implement the hash functions using Vitis HLS, which performed well at our target frequencies, as determined in the previous section. Using Vitis HLS, we were able to produce a reusable library of hash cores, which can be re-targeted to different Xilinx FPGA architectures. We chose not to use PipelineC, which would have allowed us to target FPGAs from other vendors, because of the significant compile time required for larger circuits, especially when using the *DO_PNR* flag.

4.2.1 Input width optimization

The input to the X16R function is the 80-byte (640-bit) block header. However, internally the mid-states are 64-bytes wide as each hash function has a 64-byte output digest. This provides an opportunity for optimization, as we can create two input-width-optimized versions of every hash function: an 80-byte version, and a 64-byte version. We note for the 80-byte version, only the last 4 bytes, the nonce, will vary for each input value within a batch. Therefore, for hash functions that ingest less than 76-bytes of data at a time, we can pre-compute a portion of the computation on the host CPU and load it as an *initial mid-state* at runtime. This reduces the hardware resources required by some of the 80-byte hash cores. Hash functions that benefit from this optimization are: JH, Keccak, Skein, Luffa, Cubehash, Hamsi, and Whirlpool.

4.2.2 HLS coding

There are three different hash core interfaces, 80-byte, 80-byte with mid-state, and 64-byte:

```
using header_t = struct header_t {
    bool vld;
    ap_uint<2> id;
    ap_uint<640> data;
};

using hash_t = struct hash_t {
    bool vld;
    ap_uint<2> id;
    ap_uint<512> data;
};

hash_t hash_80(header_t input, ap_uint<1600> midstate);
hash_t hash_80(header_t input);
hash_t hash_64(hash_t input);
```

The inputs to each function consist of a valid signal, an ID and the input data (either 640-bits or 512-bits). The output is always a valid signal and the 512-bit digest. Note that there is no capability for back pressure (i.e. no ready signal), the cores are non-blocking and always ready to accept data. The valid signal is simply passed through to the output in sync with the data. The 80-byte interface has an alternate version that accepts a 1600-bit initial mid-state, which is sized according to the largest initial mid-state required (Keccak). There is an accompanying HLS core that de-serializes a 32-bit stream and presents the full 1600-bits to the hash core.

In the top-level HLS function for each hash core, we insert the `HLS pipeline` pragma. This tells the HLS compiler to unroll and pipeline the function such that the initiation interval is one clock cycle. This means that the top level function will accept a new input, while outputting a result every clock cycle. The compiler is supposed to automatically apply the pipeline directive down to sub-functions as necessary to meet the initiation interval for the top level. However, we found that sometimes the tool fails to meet II=1 and we have to apply the `HLS pipeline` pragma to sub-functions explicitly. We also apply the `HLS interface ap_ctrl_none` and `HLS interface ap_none` pragmas to prevent HLS from generating control signals for the input and output ports (we explicitly define a valid signal for *header_t* and *hash_t*).

```
hash_t hash_64(hash_t input) {
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface ap_none port=input
#pragma HLS pipeline II=1
}
```

Generally, for cryptocurrency mining, the latency of a hash core is not important, as ideally the pipeline should be almost always filled. We do not put any latency constraints on the hash cores. Finally we use the `config_rtl -reset none` directive in the TCL compile script to prevent HLS from generating resets for the core. We do not use resets as this adds additional routing and may affect timing closure. For our use case, that is, dynamic reconfiguration, a global reset will reset all registers in the dynamic region every time partial reconfiguration occurs.

4.2.3 Results

Table 4.1 shows the resource utilization for the hash cores when compiled for a 625MHz clock target (passes timing after place and route). We see that the largest core takes up 327,106 LUTs.

Core	LUTs	FFs	Core	LUTs	FFs
blake_80	90,189	133,269	shavite_80	199,605	64,504
blake_64	88,826	130,809	shavite_64	146,975	54,813
bmw_80	55,178	81,891	simd_80	324,806	428,267
bmw_64	56,632	81,524	simd_64	321,783	427,834
groestl_80	299,630	84,539	echo_80	201,244	107,892
groestl_64	297,938	84,196	echo_64	199,781	89,305
jh_80	97,794	173,738	hamsi_80	47,904	52,613
jh_64	97,832	171,900	hamsi_64	139,335	153,609
keccak_80	51,054	78,445	fugue_80	175,441	84,290
keccak_64	49,293	74,112	fugue_64	154,279	72,674
skein_80	93,959	113,926	shabal_80	84,471	74,398
skein_64	87,075	103,894	shabal_64	76,664	68,531
luffa_80	66,946	63,541	whirlpool_80	50,045	76,336
luffa_64	107,251	99,022	whirlpool_64	74,140	115,393
cubehash_80	270,925	369,668	sha512_80	79,371	128,062
cubehash_64	327,106	437,639	sha512_64	79,807	127,182

Table 4.1: Hash core resource utilization

Chapter 5

X16R implementation

5.1 Target hardware

In order to implement our time-sliced approach to the X16R algorithm, we must first determine the hardware requirements for the FPGA, and the host computer. Our FPGA device must have enough logic resources (ideally in one SLR) to fit the largest hash function in our hash library, which takes up 327K LUTs. The FPGA board must have sufficiently large DRAM capacity to buffer mid-states, and be able to support high-speed dynamic partial reconfiguration.

The typical mining machine used by GPU miners consists of many GPUs connected to an x86-based host over PCIe. Due to its relatively low cost, the host machine is typically constructed from consumer grade hardware (e.g. Core-i7 instead of Xeon CPUs), which have limited PCIe lanes and slots[1]. Miners commonly bifurcate the available PCIe lanes into many x1 lanes, and use PCIe riser cables to adapt them into x16 slots (see Figure 5.1). Popular mining motherboards include: ASUS B250 Mining Expert[2], and ASRock H110 Pro BTC+[3]. These high slot density boards usually only support the PCIE Gen2 standard. In order to take advantage of the existing infrastructure of GPU miners, we should consider backward compatibility for PCIe Gen2 x1 bandwidth (i.e. 500MB/s).

We chose to target the Xilinx Virtex Ultrascale+ HBM family of devices as they con-

[1]PCIe lanes refer to the electrical connections, whereas PCIe slots refer to the physical connector. GPUs can only plug into a x16 slot, however, they can operate using just an x1 lane in reduced bandwidth mode.

[2]https://www.asus.com/au/Motherboards-Components/Motherboards/Others/B250-MINING-EXPERT/

[3]https://www.asrock.com/mb/intel/h110%20pro%20btc+/

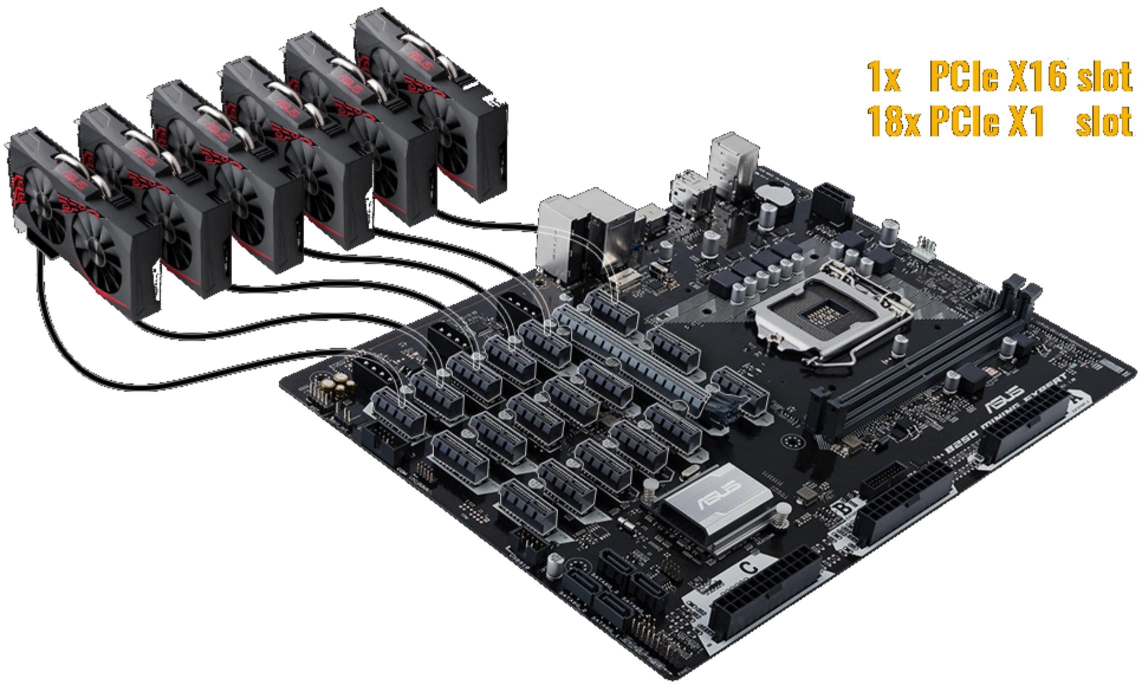

Figure 5.1: ASUS B250 Mining Expert

tain approximately 440K LUTs per SLR, have up to 16GiB of High-Bandwidth-Memory (HBM), and support DPR via the ICAP at a maximum bandwidth of 800MB/s. Specifically, we implemented our design on two different boards; the SQRL FK33, which is based around the VU33P chip, and the Xilinx Varium C1100, which is based around the U55N chip. The VU33P is a one-SLR device, while the U55N is a two-SLR device. They both have 8GiB of HBM with a memory bandwidth of 460.8GB/s. Our host is an x86-based system with an Intel Core i7-8700 and 64GiB of DDR4 ram.

5.2 System design

To realize the design, the FPGA resources are split between a static shell and a dynamic hash region. The static shell is responsible for communication between the host and the FPGA, loading and storing mid-states to and from the HBM, and orchestrating reconfiguration events that swap hash functions in and out. The dynamic region is where hash functions are swapped in and out over time and is allocated the majority of the FPGA area.

We chose the largest possible batch size of 62.9 million (i.e. considering each mid-state is 64-bytes, we need 7.5GiB) that can fit in the 8GB HBM, while reserving 512MB of space for storing partial bitstreams on chip.

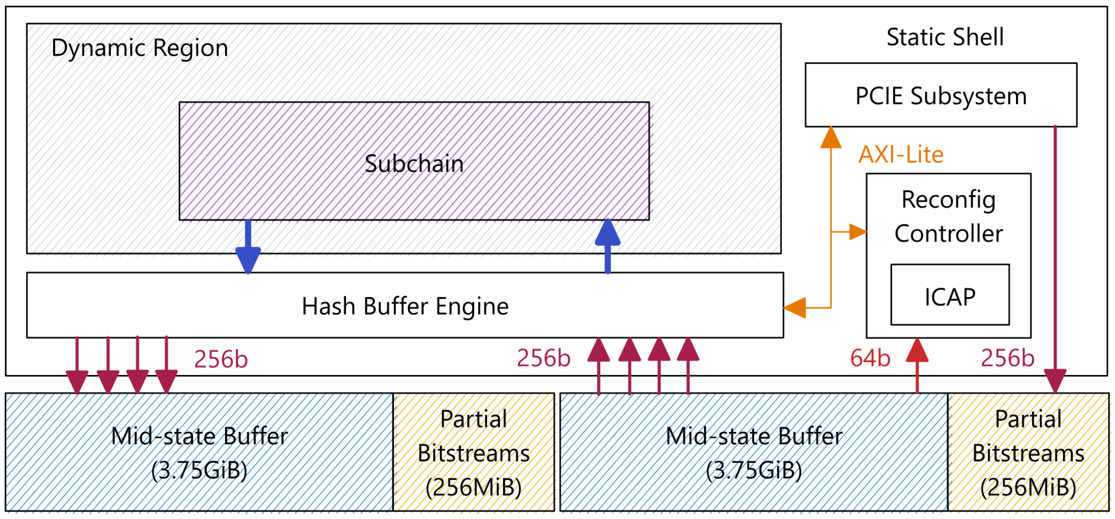

Figure 5.2: System overview

5.2.1 PCIe subsystem

The FPGA communicates with the host over PCI Express. We use the Xilinx DMA-over-PCIe IP (XDMA) core included in Vivado [34]. This allows the host PC to transfer large amounts of data from main memory onto an AXI4 bus on the FPGA with minimal interaction from the CPU. This core also allows the host to peek and poke an AXI-Lite bus at low speed. The AXI4 port is 256-bits wide and is connected to the HBM in order to load partial bitstreams, while the AXI-Lite port is 32-bits wide and is used for low-speed control signals. Both ports are clocked at 250MHz.

We configured the XDMA with a link speed of PCIE Gen3 x4 for a maximum bandwidth of 4GB/s. When plugged into a PCIe Gen3 x1 riser or PCIe Gen2 x1 riser, the link speed drops to 1GB/s and 500MB/s respectively. We note that the PCIe Gen2 x1 bandwidth is insufficient to saturate the ICAP bandwidth. We should therefore store as many partial bitstreams as possible in local memory (in the HBM) for maximum compatibility.

5.2.2 Reconfiguration controller

The reconfiguration controller is a custom HLS-based block that is responsible for loading the partial bitstreams from the HBM to the ICAP. The controller has a 64-bit wide AXI4 master port, which connects to the HBM, and a standard 32-bit wide AXI-Lite slave port, which connects to the PCIE DMA subsystem. There are sixteen registers, which hold the addresses in HBM where the partial bitstreams are stored. The addresses are loaded in order, such that register zero always holds the address of the partial bitstream for the first sub-chain (always an 80-byte version)[4], and register fifteen always hold the address of the partial bitstream for the last sub-chain.

An internal FSM, upon receiving a start signal from the host, fetches the first partial bitstream and loads it into the ICAP. On completion, a signal is sent to the hash buffer engine to start hashing. The FSM then waits for a done signal from the hash buffer engine indicating that all mid-states have been processed. Then the next partial bitstream is loaded. This process repeats until all mid-states for the last sub-chain in the chain have been processed. The controller then wraps back to loading the first partial bitstream again. The host can also issue a reset (indicating the start of a new block interval or a change in the working block header), which brings the FSM back to idle.

The ICAPE3 primitive is inserted into the HLS code as a Verilog black box. A JSON companion file is required that specifies the signal interface, initiation interval and pipeline. As the ICAPE3 is a primitive, we treat it as a combinational module, with `II=0` and `pipeline=0`. A 32-bit wide, 1024 entry deep FIFO is inserted between the AXI4 input and the ICAP. This adapts the 64-bit input to 32-bits and hides the AXI4 burst and DRAM refresh overheads, so that there is no ICAP idle time when loading partial bitstreams. The FIFO is sized at 1024x32-bit entries as this takes up one 36K BRAM block.

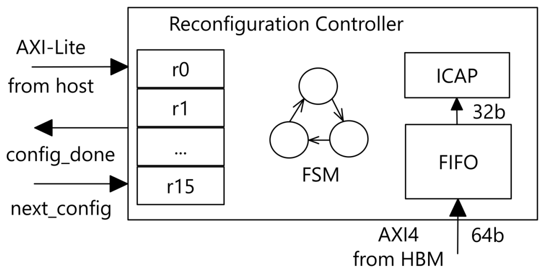

Figure 5.3: Reconfiguration controller

The reconfiguration controller logic and the AXI4 interface operate at 200MHz, which is the maximum frequency for the ICAPE3. However, the AXI-Lite interface operates at 250MHz.

[4]In the current design, the zeroth register is always 0x0_F000_0000.

The partial bitstreams are stored in a 512MiB reserved region in the HBM. The address range is split into two regions, one on the left HBM stack starting at `0x0_F000_0000` and one on the right HBM stack starting at `0x1_F000_0000`. The maximum possible full bitstream size for a single Virtex Ultrascale+ SLR is 27.02MiB, so this reserved region can hold at least eighteen full bitstreams, and it is worth noting that in practice partial bitstreams are smaller than full bitstreams.

5.2.3 Hash buffer engine

The hash buffer engine is responsible for reading the mid-states from HBM, feeding them to the sub-chain that resides within the dynamic region and writing the resulting mid-states back to HBM. The engine is connected to the HBM by eight AXI3 ports, four for writing and four for reading. On the first pass, it initializes the first sub-chain with the initial mid-state provided by the host (if needed) and feeds the sub-chain with the block header, while incrementing the nonce field. On the last pass, instead of writing the mid-states back to the HBM, the returning stream of hashes is filtered for hashes that meet the target (golden nonces). Golden nonces are inserted into the golden nonce FIFO.

In Figure 5.4, we see that internally, the hash buffer engine consists of four FIFOs (one for each AXI3 write port), a 4-to-1 multiplexer and a 1-to-4 de-multiplexer. Due to the architecture of the hardened AXI HBM switch, the mid-states are read from HBM over four 256-bit AXI3 channels using four load units. For each AXI3 channel, every two consecutive 256-bit words read are recombined to form a 512-bit mid-state every two clock cycles, which is then merged through a multiplexer to form one 512-bit stream that is fed to the sub-chain.

Within each load unit, there is a counter that keeps track of how many mid-states it has loaded. When a 512-bit mid-state is reconstructed, it is tagged with the ID (0-3) that represents the load/store unit pair, which is carried with the mid-state through the dynamic region. The output of the sub-chain is de-multiplexed to the appropriate write channel, according to the ID. This way, we can guarantee that the mid-states return to memory in order and therefore we can implicitly encode the nonce as the buffer index plus some initial offset.

As the HBM consists of DRAM, bank refreshes can stall the read and write channels for

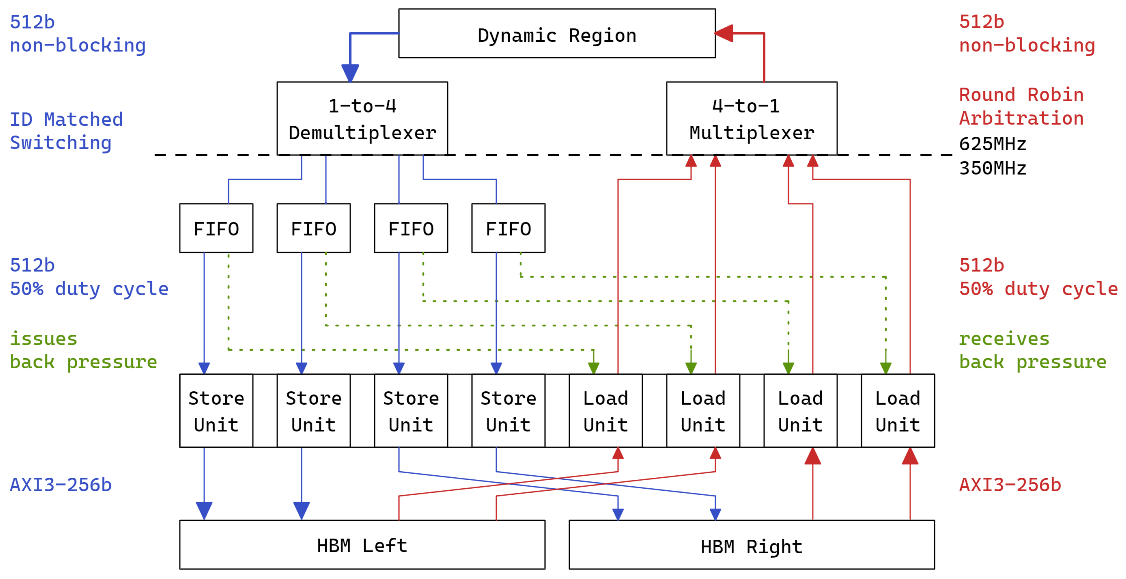

Figure 5.4: Hash buffer engine

up to 260ns every 3.9µs, therefore the FIFOs are needed to prevent mid-states from being dropped. The reason we multiplex between four HBM ports instead of two is because the hash buffer engine runs at a lower frequency (350MHz), while the sub-chains/hash-cores are running at a higher the frequency (625MHz). We do not run the hash buffer engine at the native HBM interfacing frequency (450MHz), to achieve timing closure as it is area constrained (we want to allocate as much FPGA area to the dynamic region as possible).

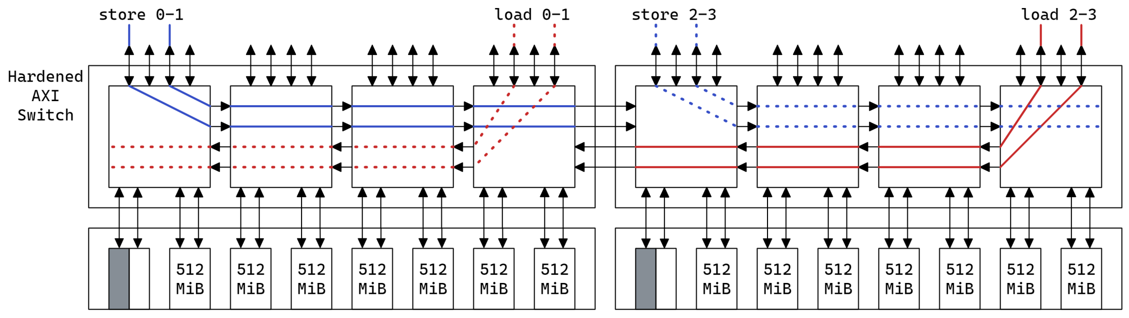

Figure 5.5: HBM channel configuration

In Figure 5.4, we see that we split the load and store channels between the left and right HBM stacks. This is due to a bandwidth limitation within the hardened HBM AXI switch. In Figure 5.5, we see that between each AXI sub-switch there are two simplex channels running right and two running left. As we use eight channels in total, we must carefully arrange them such that no bottlenecks arise. As such, we split the memory into four contiguous buffers, given IDs: 0, 1, 2, and 3. Buffers 0 and 1 reside in the left HBM stack and buffers 2 and 3 reside in the right HBM stack. During hash computation, while buffers 0 and 2 are being read from, buffers 1 and 3 are being written to. After each reconfiguration, the buffers being read from and written to are swapped. We connect the channels to the load and store units so that mid-states being stored always generate right-bound memory transactions and mid-states being loaded always generate left-bound memory transactions. This way there is no contention for lateral memory bandwidth, and there is no contention between loads and stores within a buffer.

5.3 Single function per reconfiguration

In the single function per reconfiguration scheme (SFR), we only configure sub-chains containing one hash function at a time. Thus there are thirty-two possible sub-chains, sixteen 80-byte input sub-chains and sixteen 64-byte input sub-chains. We do not have enough space to load all thirty-two partial bitstreams into the 512MiB HBM region, therefore, we opt to load the sixteen 64-byte sub-chains upon power-on of the FPGA via PCIe DMA from the host. Then at the start of every block interval, we only load the one 80-byte sub-chain that is needed. The partial bitstreams for the 80-byte sub-chains are kept in the host DRAM to allow for low-latency copy. On our system, with a PCIe link speed of Gen3 x4, we found that the worse-case total transfer time for a 27.02MiB bitstream is 7.82ms and is 6.4ms in the average case. This adds an additional ~0.02% overhead to the system which is negligible. For PCIe Gen2 x1 speeds, the overhead increases to ~0.13% which is also negligible.

At the start of every block interval, the host software (written in Python) receives a new block header from a simulated mining pool. It extracts the hash ordering and copies the first 80-byte sub-chain, to the FPGA HBM. It then loads the addresses in HBM for each partial bitstream, in order, into the sixteen registers in the reconfiguration controller over the AXI-Lite bus. It also loads the block header along with the starting nonce and the target to the hash buffer engine. If required, the initial mid-state is computed on the CPU (this is required for JH, Keccak, Skein, Luffa, Cubehash, Hamsi, and Whirlpool) and loaded into the hash buffer engine. Then a start signal is sent to the reconfiguration controller to start the system.

The reconfiguration controller starts by loading the first partial bitstream, and notifying the hash buffer engine when done. The hash buffer engine loads the initial mid-state (if required) and the first 608-bits of the block header to the newly configured sub-chain by shifting the data in 32-bit chunks. Once complete, the four load units generate new data by incrementing the nonce by one and adding an offset equal to the load/store unit ID (0-3) multiplied by 15,728,640. Thus the load units generate non-overlapping nonces (memory addresses). The nonces are fed through the pipeline as illustrated in Figure 5.4 and saved into the HBM. Once all nonces have been processed, the hash buffer engine sends a done signal to the reconfiguration controller to start configuring the next sub-chain.

For subsequent sub-chains, the mid-states are loaded from memory instead of being generated. On the last sub-chain, the outputs are not saved to memory, but are filtered for ones that meet the target. Outputs that meet the target have their nonces (the memory address they were stored in) placed into a 32-bit wide FIFO. The host continuously polls this FIFO for *golden nonces.*

5.4 Multiple functions per reconfiguration

In the multiple functions per reconfiguration scheme (MFR), we configure up to two hash functions at a time. The hardware components in the system stay the same, however the software is changed. Instead of loading all sixteen 64-byte sub-chains at power on, we load the sub-chains that are needed for the specific chain ordering at the start of every block interval. The host software, upon receiving the block header, extracts the hash chain ordering and divides it into sub-chains of at most two hash functions according to a list of available pre-compiled sub-chains. The required partial bitstreams are loaded into the FPGA's HBM via DMA. The transfer is completed in ~97.5ms on average, which adds an overhead of ~0.16%, as opposed to 0.02% for SFR. On PCIe Gen2 x1 systems, this overhead increases to ~1.28%.

Each sub-chain contains up to two hash functions that can be selected via a switch, that connects the sub-chain input to either of the hash function inputs and allows the output of one hash function to be the input of the other. This cuts down on the number of permutations that are required to be pre-compiled.

Not every combination of hash functions fits within one Virtex Ultrascale+ SLR, e.g. combinations such as Groestl-Groestl are too large. We were able to successfully compile 96, of the 64-byte to 64-byte combinations, out of the possible 136. In total these partial bitstreams take up 2.43GiB of memory on the host.

5.5 Multi-SLR scaling

Both SFR and MFR can easily be scaled to multiple SLRs, by duplicating another dynamic region onto a second SLR and by extending the static region to encompass the second ICAP. In this scheme, there is a separate reconfiguration controller for each SLR. The bus that connects the reconfiguration controllers to the HBM is widened to 256-bits, thus allowing up to seven (leaving room for DRAM overhead) reconfiguration controllers to share a single 256-bit AXI3 port on the HBM. A round-robin arbiter is used to multiplex between the reconfiguration controllers. This way, each reconfiguration controller is able to reconfigure its local dynamic region independently and in parallel. As long as each ICAP does not load a bitstream destined for a different SLR, they are able to operate at the maximum 200MHz frequency. Therefore, there is no additional reconfiguration overhead related to multi-SLR scaling.

We implemented a two SLR system on the Xilinx C1100 (U55N) board, where sub-chains have only a single hash function. We call this scheme SFR2. The mid-states are loaded and stored by the hash buffer engine as in the single SLR design. However, after they flow through the first dynamic region, they cross the SLR boundary and are processed by the next hash function configured in the dynamic region of the second SLR.

In this scheme, there are now two sets of sixteen 64-byte partial bitstreams and one set of 80-byte partial bitstreams. However, we only need to load at most eight partial bitstreams for each dynamic region. This is because in the worse case, where all sub-chains are one hash function long, we will need to configure each dynamic region eight times.

5.6 Results

We physically implemented our SFR, and MFR designs on the SQRL FK33 board and our SFR2 design on the Xilinx C1100 board. We ran an 8-hour long mining session,

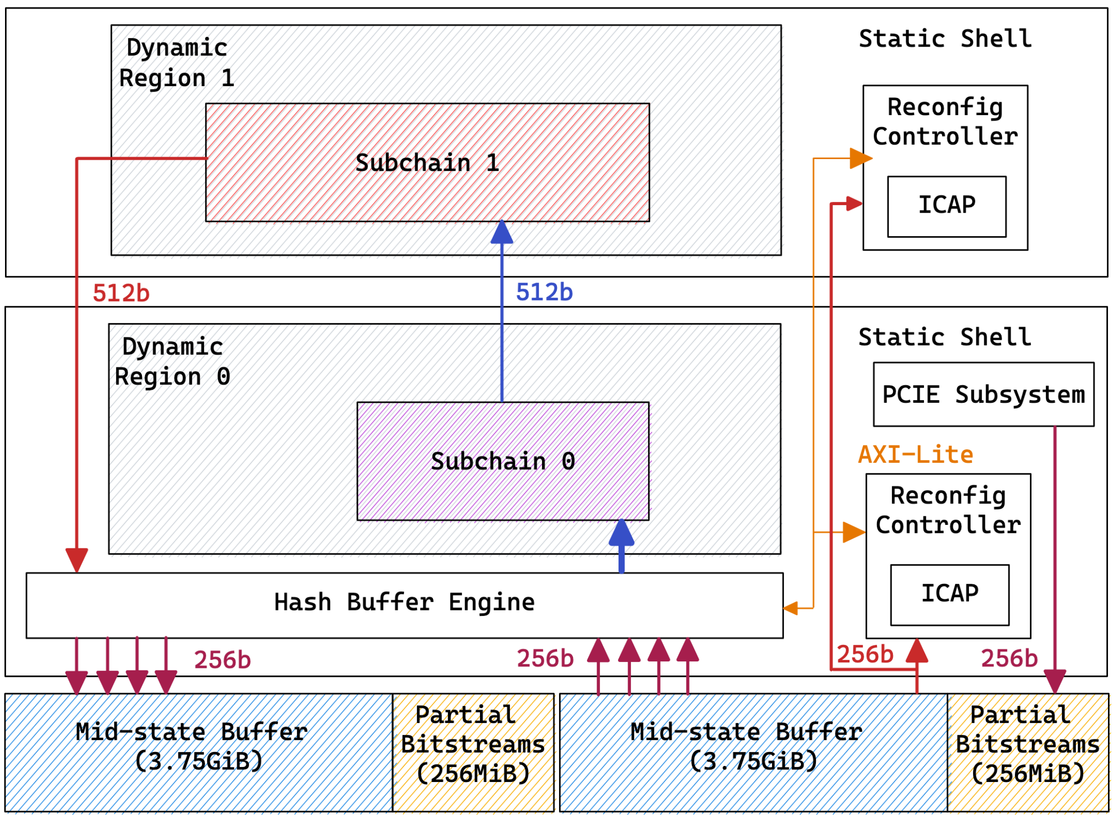

Figure 5.6: Two SLR system

with a simulated mining pool, with the block interval set to one minute. We used an Elmor Labs Power Measurement Device[5], to measure the average power consumption. We measured the 12V rails on the PCIE slot and the 8-pin auxiliary power connector. Table 5.1 shows a comparison of performance measured, as mega-hashes per second, as well as power consumption in watts, and power-efficiency in mega-hashes per joule, between various NVIDIA GPUs (GTX and RTX), a previous FPGA-based solution from Altered Silicon[6], a commercial X16R ASIC (OW1) and our solutions (SFR, MFR, and SFR2). We obtained the X16R performance for the *RTX 2070 Super,* and *RTX 3090* GPUs by benchmarking with commercial mining software, *T-Rex version 0.19.14* [27]. All other GPU performance figures were obtained from *nicehash.com*[7], which maintains a database of GPU mining performance for various PoW algorithms.

The times to complete a single batch for SFR, MFR, and SFR2 are on average 2.28s, 1.77s, and 1.19s respectively. The time to reconfigure each bitstream on average is: 28.8ms for SFR, 32.6ms for MFR, and 28.8ms for SFR2 (i.e. when configuring two regions in parallel).

[5]https://elmorlabs.com/product/elmorlabs-pmd-power-measurement-device/
[6]Product information is no longer available on the web.
[7]https://www.nicehash.com/profitability-calculator

Device	MH/s	Watts	MH/J	Node
GTX 1070	17.8	125	0.14	TSMC 16nm
GTX 1080	17.71	150	0.12	TSMC 16nm
GTX 1080 Ti	19.33	170	0.11	TSMC 16nm
RTX 2060	19.88	90	0.22	TSMC 12nm
RTX 2070	25.5	110	0.23	TSMC 12nm
RTX 2070 Super *	34	110	0.31	TSMC 12nm
RTX 2080	34	108	0.31	TSMC 12nm
RTX 2080 Ti	43	145	0.30	TSMC 12nm
RTX 3090 *	63.99	330	0.19	Samsung 8nm
OW1 ASIC	182	1400	0.13	Unkown
AS VCU1525	120	220	0.55	TSMC 16nm
ZU9EG	2	Unkown	Unkown	TSMC 16nm
VU33P (SFR)	27.07	26	1.04	TSMC 16nm
VU33P (MFR)	35.02	31	1.13	TSMC 16nm
U55N (SFR2)	52.35	48	1.09	TSMC 16nm

Table 5.1: X16R performance and efficiency

For MFR, we found that an average of 10.04 reconfigurations per batch is required. This gives a reconfiguration overhead of 20.2%, 18.5% and 10.1% respectively.

The raw performance, without accounting for inter-batch overhead, for SFR, MFR, and SFR2 is; 27.59MH/s, 35.54MH/s, and 52.87MH/s respectively. However, when we collect the average hash rate over a longer period of time (i.e. 8 hours) we obtain slightly decreased performance due to the inter-batch overhead.

We see that our SFR design achieves 27.07MH/s while consuming only 26W and our MFR design increased the hash rate to 35.02MH/s while consuming 31W. We see that MFR has 8.7% better power efficiency than SFR, which can be attributed to the fact that there is a baseline idle power usage when the FPGA is power on, and as we do more computations on the FPGA, this idle power usage becomes a smaller proportion of the total power consumption. Our SFR2 design achieves 52.35MH/s, which is 93% higher than SFR, while consuming 48W.

When comparing our designs with GPUs with similar hashing performance, we see that our SFR hash rate is similar but a bit higher than that of the RTX 2070. However, our SFR design is 4.52x more power efficient. Our MFR design has a slightly higher hash rate than the RTX 2080, but is 3.65x more power efficient. Our SFR2 design falls between the RTX 2080 Ti and RTX 3090 in terms of hash rate, but is respectively 3.63x and 5.74x more power efficient. It is likely that lower-end GPUs within the RTX 30-series family will be more power efficient than the RTX 3090. However, we did not have any on hand to test, nor could we find the performance numbers on *nicehash.com*.

We should also note that the FPGAs used in our experiments were manufactured on the TSMC 16nm technology node, while the GTX 10-series, RTX 20-series, and RTX 30-series were manufactured on TSMC 16nm, TSMC 12nm, and Samsung 8nm respectively. The largest GPUs from each family are the GTX 1080 Ti with 12 billion transistors, the RTX 2080 Ti with 18.6 billion transistors, and the RTX 3090 with 28.3 billion transistors. We could not find the exact transistor count for the FPGAs we used, however, the largest FPGA within the Xilinx 16nm Virtex Ultrascale+ family is the VU19P which is manufactured with 39 billion transistors. The VU19P device contains 4.086 million LUTs, which is 9.29x the number of LUTs we used for SFR and MFR, and 4.64x the number of LUTs we used for SFR2. Therefore it may be fairer from a manufacturing technology point of view to compare our FPGA-based designs to the GTX 10-series. When we compared to the most efficient GPU in the GTX 10-series that we tested, the GTX 1070, our MFR design is 8.07x more power efficient.

Our designs are also more power efficient than a proprietary FPGA-based solution, listed as AS VCU1525, from Altered-Silicon. Their system requires two Xilinx VCU1525 boards, which are teamed together using two QSFP28 direct attach copper cables. However, in Table 5.1 we reported the hash rate and power consumption for one board (half the total system). We also note that our designs are more power efficient than the OW1 "ASIC" mining unit, which uses 72 separate chips to run X16R. We suspect that the chips in the OW1 are most likely manufactured on an older technology node. We warn readers to take the AS VCU1525 and OW1 performance numbers with a grain of salt, as publicly available information is scarce, aside from claims from third party blog sites[8] and anonymous sources[9].

[8]https://en.cryptonomist.ch/2019/09/17/mining-ravencoin-hashrate/

[9]Users who bought the AS VCU1525 solution reported the results shown in Table 5.1.

Chapter 6

Conclusion

6.1 Summary

The aim of this research was to demonstrate that FPGA runtime reconfiguration could be used to speed up multi-hash proof-of-work mining, and in particular, to produce a more competitive implementation of the X16R algorithm. We examined why a static, ASIC-like architecture would be inefficient when implementing multi-hash PoW. We considered two strategies, *Dynamic Full Chain*, and *Time-sliced Dynamic Sub-chains*, which use dynamic partial reconfiguration to address the bottlenecks that appear in the static architecture. For dynamic full chain, we considered reconfiguring the FPGA at the start of every block interval with a full chain of sixteen hash cores. While this method would in theory be the most efficient, as it has very little overhead, it is impractical to implement with current FPGAs due to the large amount of logic resources required, not to mention the challenge of producing the bitstreams needed. We chose to implement our X16R algorithm on an FPGA using the time-sliced dynamic sub-chains method, which divides the X16R chain into sub-chains, of which only one sub-chain at a time is required to be configured. This is a more practical approach as it lowers the minimum resource requirements when choosing an FPGA mining platform, and therefore lowers the barrier to entry for miners.

To obtain a fair comparison with commercial implementations of X16R, we required a reasonably performant implementation ourselves. To this end we required sixteen high performance hash cores that are optimized for cryptocurrency mining. We were unable to find hash cores that fit our requirements within the public domain, therefore we chose to implement our own library of hash cores. During this process, we investigated whether the auto-pipelining feature provided by various higher-level tools were good enough to use instead of manual pipelining at the RTL level. We compared our hand-optimized designs for AES, Keccak, and Groestl, which were written in the Chisel HDL, to an auto-pipelining tool called PipelineC, as well as Xilinx's officially supported high-level synthesis tool, Vitis HLS. We found that the circuits generated by Vitis HLS offered comparable and sometimes better performance than our hand-optimized RTL at the target frequencies we care about. This informed our decision to implement all sixteen hash functions using Vitis HLS.

We implemented three different configurations of our time-sliced dynamic sub-chains architecture: SFR — where sub-chains are of length one, MFR — where sub-chains are of length one or two, and SFR2 — where we extended our SFR design to use two SLRs. Our designs achieved much better power efficiency than any GPU solution that we were able to obtain performance metrics for, and in terms of hash throughput, our SFR2 solution outperforms an RTX 2080 Ti but falls short of an RTX 3090. The latter comparison could be considered unfair since we compared our 16nm FPGA against a theoretically superior 8nm GPU.

6.2 Discussion

We successfully demonstrated that by using the unique feature of FPGAs to reconfigure at runtime, one can speed up certain classes of algorithms, in particular multi-hash PoW algorithms. Thus, FPGAs are a practical platform for mining cryptocurrencies that employ this kind of PoW, and are more suitable than GPUs, especially when considering power efficiency. The power efficiency is an important factor for miners as it factors into the total-cost-of-ownership (TCO) for a device. For example, using the May 2022 New South Wales electricity reference price[1] of $0.3782 AUD/kWh, a miner would be expected to pay $1093.30 AUD per year in electricity cost when mining with an RTX 3090, as opposed to only $159.03 AUD per year when mining with a Xilinx C1100 FPGA (U55N) card with our SFR2 solution.

[1]As given by the AER DMO price for 2022-2023: https://www.aer.gov.au/news-release/aer-sets-energy-price-cap-to-protect-consumers .

We hope that the cryptocurrency community will consider FPGAs as a separate class of computing device from ASICs. Runtime reconfiguration gives FPGAs the flexibility to optimize an algorithm in both the space and time dimensions. PoW algorithm designers should therefore also consider whether they want properties of FPGA-resistance, separate from ASIC-resistance.

We note that there is still room for improvement upon the SFR, MFR, and SFR2 solutions we presented, while using the same hardware. During our experiments with auto-pipelining, we observed that it is possible to pipeline the Keccak and Groestl cores to obtain a maximum clock frequency of 922MHz and 807MHz respectively. For Keccak, this can be done using auto-pipelining. However for Groestl, hand optimization would be required. If we had set our target clock frequency to 800MHz, as opposed to 625MHz, and hand-optimized those designs that underperformed when auto-pipelined, this would lead to an approximate 28% increase in performance. Alternatively, we can compile each sub-chain such that they operate at their own maximum clock frequency, and have the clock generator to be dynamically adjusted after each reconfiguration. We did not implement MFR2 (i.e. MFR that spans two SLRs, similar to SFR2 but with up to two hash functions per sub-chain) due to the complexity of compiling 96 additional partial bitstreams for each of the two SLRs.

In this work we implemented a time-sliced X16R algorithm on an FPGA manufactured using TSMC's 16nm technology node. We hypothesise that by re-implementing our method on the latest Xilinx Versal family of devices (TSMC 7nm) we will be able to achieve at least a two-fold increase in performance when using devices with the same number of SLRs. This is due to a doubling in the number of LUTs available in an SLR, coupled with an eight-fold improvement in configuration bandwidth[12].

6.3 Future work

In order to reduce the number of partial bitstreams required for MFR, we need to investigate the use of techniques such as *nested reconfiguration* [36] where the dynamic region is configured in multiple stages and *module relocation* where modules can be moved from one part of the FPGA to another without the need for recompilation.

Applying nested reconfiguration to our X16R design would entail splitting the reconfiguration process into two stages. The first stage configures a sub-chain *template* containing a number of *slots* that can be further reconfigured with our hash cores. This way, we can have templates that allow for a single slot, dual slots, three slots, etcetera. We could have templates where the slots are equal in size or unbalanced such that slots are of different sizes. The hash cores can then be compiled into these slots independently, such that the scheduler is able to mix and match hash cores at run time. By using nested reconfiguration we can significantly reduce the compilation time and reduce the partial bitstream storage requirements.

Module relocation allows us to move our hash cores in a more fine-grained manner, in an X-Y grid. This is done by modifying an existing partial bitstream at runtime. By introducing module relocation, we can further reduce the number of partial bitstreams that must be pre-compiled, thus allowing us to combine more hash cores into a sub-chain and improving performance. Module relocation is a field in which there have been many previous studies. However, there have been few practical applications of the method, due to the need for very low level and precise floor-planning. However, due to recent advancements in both FPGA hardware and compiler technologies, we believe it is time to revisit the idea of practical module relocation.

The new generation of FPGA hardware, specifically the Xilinx Versal ACAP (built on TSMC 7nm) family have significantly improved runtime reconfiguration capabilities [12], allowing for eight times faster loading of partial bitstreams compared to the devices we used in our experiments. The FPGA fabric in Versal is more regular and thus more amenable to module relocation. Additionally, a dedicated network-on-chip (NoC) that is embedded in the fabric, allows user modules to communicate with each other and with memory without needing a soft-network. A dedicated reconfiguration controller and ARM cores eliminates the need for a static shell, which allows the designer to concentrate on the compute architecture rather than the supporting infrastructure.

References

[1] Jonathan Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *DAC Design Automation Conference 2012*. DAC Design Automation Conference 2012. June 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[2] Christian Beckhoff, Dirk Koch, and Jim Torresen. "Portable Module Relocation and Bitstream Compression for Xilinx FPGAs". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014 24th International Conference on Field Programmable Logic and Applications (FPL). Sept. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927480.

[3] Michael Bedford Taylor. "The Evolution of Bitcoin Hardware". In: *Computer* 50.9 (2017), pp. 58–66. ISSN: 1558-0814. DOI: 10.1109/MC.2017.3571056.

[4] Sheetal U. Bhandari et al. "Real Time Video Processing on FPGA Using on the Fly Partial Reconfiguration". In: *2009 International Conference on Signal Processing Systems*. 2009 International Conference on Signal Processing Systems. May 2009, pp. 244–247. DOI: 10.1109/ICSPS.2009.32.

[5] Alex Biryukov and Dmitry Khovratovich. "Equihash: Asymmetric Proof-of-Work Based on the Generalized Birthday Problem". In: Jan. 1, 2016. DOI: 10.14722/ndss.2016.23108.

[6] Tron Black. *Ravencoin — ASIC Thoughts Round Two*. Medium. Apr. 6, 2020. URL: https://tronblack.medium.com/ravencoin-asic-thoughts-round-two-f4f743942656 (visited on 05/08/2022).

[7] Tron Black and Joel Weight. *X16R-Whitepaper.Pdf*. 2018. URL: https://ravencoin.org/assets/documents/X16R-Whitepaper.pdf (visited on 05/28/2022).

[8] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.*

[9] Hyungmin Cho. “ASIC-Resistance of Multi-Hash Proof-of-Work Mechanisms for Blockchain Consensus Protocols”. In: *IEEE Access* 6 (2018), pp. 66210–66222. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2878895.

[10] Evan Duffield and Daniel Diaz. *Dash: A PrivacyCentric CryptoCurrency.*

[11] Bruce Fenton and Tron Black. *Ravencoin: A Peer to Peer Electronic System for the Creation and Transfer of Assets.*

[12] Brian Gaide et al. “Xilinx Adaptive Compute Acceleration Platform: Versal ™ Architecture”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* FPGA ’19: The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Seaside CA USA: ACM, Feb. 20, 2019, pp. 84–93. ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293906. URL: https://dl.acm.org/doi/10.1145/3289602.3293906 (visited on 05/28/2022).

[13] Kris Gaj, Ekawat Homsirikamol, and Marcin Rogawski. “Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs”. In: Aug. 17, 2010, pp. 264–278. ISBN: 978-3-642-15030-2. DOI: 10.1007/978-3-642-15031-9_18.

[14] Xinfei Guo et al. “Agile-AES: Implementation of Configurable AES Primitive with Agile Design Approach”. In: *Integration* 85 (July 1, 2022), pp. 87–96. ISSN: 0167-9260. DOI: 10.1016/j.vlsi.2022.04.005. URL: https://www.sciencedirect.com/science/article/pii/S0167926022000426 (visited on 05/05/2022).

[15] Ekawat Homsirikamol and Kris Gaj. “Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study”. In: *Applied Reconfigurable Computing.* Ed. by Kentaro Sano et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 217–228. ISBN: 978-3-319-16214-0. DOI: 10.1007/978-3-319-16214-0_18.

[16] Ekawat Homsirikamol and Kris Gaj. “Toward a New HLS-based Methodology for FPGA Benchmarking of Candidates in Cryptographic Competitions: The CAESAR Contest Case Study”. In: *2017 International Conference on Field Programmable*

Technology (ICFPT). 2017 International Conference on Field Programmable Technology (ICFPT). Dec. 2017, pp. 120–127. DOI: 10.1109/FPT.2017.8280129.

[17] Jaekyung Im and Seokhyeong Kang. "Comparative Analysis between Verilog and Chisel in RISC-V Core Design and Verification". In: *2021 18th International SoC Design Conference (ISOCC)*. 2021 18th International SoC Design Conference (ISOCC). Oct. 2021, pp. 59–60. DOI: 10.1109/ISOCC53507.2021.9614007.

[18] Intel. *Intel Stratix 10 Configuration User Guide.*

[19] *Kawpowminer (Ethminer Fork with ProgPoW Implementation)*. RavenCommunity, May 23, 2022. URL: https://github.com/RavenCommunity/kawpowminer (visited on 05/28/2022).

[20] Julian Kemmerer. *PipelineC*. May 2, 2022. URL: https : / / github . com / JulianKemmerer/PipelineC (visited on 05/05/2022).

[21] Rakan Khraisha and Jooheung Lee. "A Scalable H.264/AVC Deblocking Filter Architecture Using Dynamic Partial Reconfiguration". In: ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings. Apr. 19, 2010, pp. 1566–1569. DOI: 10.1109/ICASSP.2010.5495525.

[22] Dirk Koch et al. "Partial Reconfiguration on FPGAs in Practice — Tools and Applications". In: *ARCS 2012*. ARCS 2012. Feb. 2012, pp. 1–12.

[23] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.*

[24] Marie Nguyen and James C. Hoe. "Time-Shared Execution of Realtime Computer Vision Pipelines by Dynamic Partial Reconfiguration". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018 28th International Conference on Field Programmable Logic and Applications (FPL). Aug. 2018, pp. 230–2304. DOI: 10.1109/FPL.2018.00046.

[25] C. Patterson. "High Performance DES Encryption in Virtex/Sup TM/ FPGAs Using JBits/Sup TM/". In: *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871)*. Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871). Apr. 2000, pp. 113–121. DOI: 10.1109/FPGA.2000.903398.

[26] Thinh Hung Pham, Suhaib A. Fahmy, and Ian Vince McLoughlin. “An End-to-End Multi-Standard OFDM Transceiver Architecture Using FPGA Partial Reconfiguration”. In: *IEEE Access* 5 (2017), pp. 21002–21015. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2756914.

[27] *Release T-Rex 0.19.14 · Trexminer/T-Rex.* GitHub. URL: https://github.com/trexminer/T-Rex/releases/tag/0.19.14 (visited on 05/28/2022).

[28] Ashish Rajendra Sai et al. “Taxonomy of Centralization in Public Blockchain Systems: A Systematic Literature Review”. In: *Information Processing & Management* 58.4 (July 1, 2021), p. 102584. ISSN: 0306-4573. DOI: 10.1016/j.ipm.2021.102584. URL: https://www.sciencedirect.com/science/article/pii/S0306457321000844 (visited on 05/28/2022).

[29] National Institute of Standards and Technology. *FIPS PUB 180-2.* URL: https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf (visited on 05/28/2022).

[30] tevador. *RandomX.* May 27, 2022. URL: https://github.com/tevador/RandomX (visited on 05/28/2022).

[31] *The Whirlpool Hash Function.* Nov. 29, 2017. URL: https://web.archive.org/web/20171129084214/http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html (visited on 05/28/2022).

[32] John Tromp. *Cuckoo Cycle: A Memory Bound Graph-Theoretic Proof-of-Work.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 49–62. DOI: 10.1007/978-3-662-48051-9_4. URL: http://link.springer.com/10.1007/978-3-662-48051-9_4 (visited on 05/28/2022).

[33] Meltem Sonmez Turan et al. *Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition.* NIST IR 7764. Gaithersburg, MD: National Institute of Standards and Technology, 2011, NIST IR 7764. DOI: 10.6028/NIST.IR.7764. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7764.pdf (visited on 05/28/2022).

[34] Xilinx. *DMA/Bridge Subsystem for PCI Express v4.1 Product Guide.* 2021.

[35] Xilinx. *Getting Started with Vitis HLS • Vitis High-Level Synthesis User Guide (UG1399) • Reader • Documentation Portal.* URL: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls (visited on 05/08/2022).

[36] Xilinx. *Technology Advancements for Dynamic Function eXchange in Vivado ML Edition.* 2021, p. 13.

[37] Xilinx. *UltraScale Architecture Configuration User Guide.* 2022.

[38] Xilinx. *Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics.* 2021, p. 81.

Appendix A

Appendix A

A.1 Ethash

Ethash, which is sometimes referred to as Dagger-Hashimoto, is a memory-hard algorithm that was developed for the Ethereum blockchain. It consists of the following high-level steps:

Preparation:

1. A new *seed* is computed every *epoch,* which is 30,000 blocks. The seed for the current epoch is the Keccak-256 hash of the previous epoch.
2. From the seed, 16MiB pseudo-random data, called the *cache* is computed.
3. A pseudo-random N GiB *dataset* is generated from the cache. N starts at 1GiB for the first epoch, and grows linearly over time, increasing for each new epoch. The exact values of N for each epoch can be found on the Ethereum wiki[1].

Mining:

1. The block header and a 64-bit nonce is first hashed using Keccak-256 to produce a 256-bit digest.
2. A *mix* variable is used to index and fetch a 128-byte *page* from the dataset, and is initially set to the output of step 1.

[1]https://eth.wiki/en/concepts/ethash/ethash

3. The mix variable is then updated by combining it with the page which was fetched from the dataset, using a *mixing function*.
4. Steps 2-3 are repeated 64 times.
5. The resulting mix value is post-processed into a final 256-bit digest.
6. This final digest is compared with the target to determine if this is a winning hash.

We can see that each iteration of Ethash requires 64 memory reads of 128-bytes each, totalling 8KiB. Meaning that a chip with a maximum memory bandwidth of 500GB/s is limited to a maximum hash rate of 61MH/s. Currently, as of May 2022, the size of the dataset N for the Ethereum blockchain is 4.8672GiB.

A.2 RandomX

While most proof-of-work algorithms operate by executing a fixed algorithm on random data, RandomX makes the algorithm pseudo-randomly determined. The current block header, and a nonce, generates the current algorithm. By iterating the nonce, a miner can generate many valid programs to prevent overlapping work. However, to prevent miners from cherry-picking easy programs by iterating through nonces and performing static analysis, a chain of programs is used. The generation of each subsequent program is dependent on the result of the former. Therefore, a hypothetical ASIC designed to mine a programmatic PoW must be fully reprogrammable. The RandomX developers take this concept further by making their algorithm take advantage of the following features of modern CPUs:

- super-scalar execution, the ability to execute multiple instructions in one clock cycle,
- out-of-order execution, the ability to re-order instructions to minimize stalls due to data availability, and
- multi-level caching, the ability to store small amounts of data close to the CPU cores to reduce latency incurred by data movement.

Thus, an ASIC designed for mining RandomX will likely resemble a modern desktop or server CPU. A RandomX program executes on a virtual machine with the following specifications:

- 64-bit integer arithmetic unit with:
  - 8 x 64-bit registers
- 64-bit floating-point arithmetic unit with:
  - 4 x 128-bit read-only registers (constant for a particular instance of a program)
  - 4 x 128-bit additive registers (can only store results after additions or subtractions)
  - 4 x 128-bit multiplicative registers (can only store results after multiplications, divisions or, square roots)
  - where each 128-bit register stores two 64-bit floating-point numbers (the two are not independently addressable)
- 2MB scratchpad consisting of:
  - 2MB L3 cache (inclusive of L1 and L2)
  - 256KB L2 cache (inclusive of L1)
  - 16KB L1 cache
- 2GB RAM (read-only dataset)

Any random program will only modify the state of the register file and scratchpad. The 2GB dataset only changes once every 2048 blocks (~2 days for Monero) with a 64-block forewarning. The dataset is generated using a SuperscalarHash function specifically designed to burn as much power as possible. This is done to prevent ASICs from computing parts of the dataset on the fly, bypassing the 2GB memory requirement.

At a high level, RandomX does the following. It takes the input (header and nonce), hashes it, and uses the hash as a seed to generate a random program. It then executes this program a large number of times (such that the time spent running the random program dominates). We obtain an output that is to be used as a seed to generate a new random program (also to be executed a large number of times). The series of steps is done several times to prevent cherry-picking as referred to earlier.

One iteration of RandomX is computed as follows:

1. The input to RandomX (block header and nonce) is hashed using Blake2b-512 to produce an initial seed.
2. This initial seed is used to initialize the scratchpad using an AES-based pseudo-number generator (PRNG), with the last 64 bytes of the scratchpad used as the seed for the next part.
3. The seed is used to generate (also using an AES-based PRNG) a random 256 instruction program.
4. The program is executed 2048 times in a loop, producing a 256-byte result (the state of the register file).
5. Do steps 3-4 eight times, using the Blake2b-512 hash of the result as the seed.
6. The scratchpad is hashed using an AES-based hash function producing a 64-byte digest.
7. The Blake2b-256 hash of the scratchpad digest, concatenated with the final state of the register file, is considered the output for one iteration of RandomX.