Open-Source Parameterized Low-Latency Aggressive Hardware Compressors and Decompressors for Memory Compression

James C. Jearls

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Ali R. Butt, Co-chair

Xun Jian, Co-chair

Kirk W. Cameron

Changwoo Min

May 11, 2021

Blacksburg, Virginia

Keywords: compression, decompression, accelerator, hardware, parameterized, low-latency

Copyright 2021, James C. Jearls

Open-Source Parameterized Low-Latency Aggressive Hardware Compressors and Decompressors for Memory Compression

James C. Jearls

(ABSTRACT)

In recent years, memory has shown to be a constraining factor in many workloads. Memory is an expensive necessity in many situations, from embedded devices with a few kilobytes of SRAM to warehouse-scale computers with thousands of terabytes of DRAM. Memory compression has existed in all major operating systems for many years. However, while faster than swapping to a disk, memory decompression adds latency to data read operations. Companies and research groups have investigated hardware compression to mitigate these problems. Still, open-source low-latency hardware compressors and decompressors do not exist; as such, every group that studies hardware compression must re-implement. Importantly, because the devices that can benefit from memory compression vary so widely, there is no single solution to address all devices' area, latency, power, and bandwidth requirements. This work intends to address the many issues with hardware compressors and decompressors. This work implements hardware accelerators for three popular compression algorithms; LZ77, LZW, and Huffman encoding. Each implementation includes a compressor and decompressor, and all designs are entirely parameterized. There are a total of 22 parameters between the designs in this work. All of the designs are open-source under a permissive license. Finally, configurations of the work can achieve decompression latencies under 500 nanoseconds, much closer than existing works to the 255 nanoseconds required to read an uncompressed 4 KB page. The configurations of this work accomplish this while still achieving compression ratios comparable to software compression algorithms.

Open-Source Parameterized Low-Latency Aggressive Hardware Compressors and Decompressors for Memory Compression

James C. Jearls

(GENERAL AUDIENCE ABSTRACT)

Computer memory, the fast, temporary storage where programs and data are held, is expensive and limited. Compression allows for data and programs to be held in memory in a smaller format so they take up less space. This work implements a hardware design for compression and decompression accelerators to make it faster for the programs using the compressed data to access it. This work includes three hardware compressor and decompressor designs that can be easily modified and are free for anyone to use however they would like. The included designs are orders of magnitude smaller and less expensive than the existing state of the art, and they reduce the decompression time by up to 6x. These smaller areas and latencies result in a relatively small reduction in compression ratios: only 13% on average across the tested benchmarks.

Dedication

For Alfie, the smartest, happiest dog a man could ask for

Acknowledgments

I would like to thank my lovely wife, Emilie, for her support while I performed this research. I would also like to thank my family and friends for their support. I would like to thank Dr. Jian for his close work on my thesis. I would like to thank Dr. Cameron for mentoring me and guiding me through both my undergraduate and graduate research careers. I would like to thank Dr. Butt for his helpful questions that led me to think more deeply about the topics I was investigating. And I would like to thank Dr. Min for his helpful information on modern memory management algorithms. This material is based upon work supported in part by the National Science Foundation under Grant No. 1850025, 1919113, 1942590, 1838271, 1565314, and 1939076.

Contents

List of Figures ix

List of Tables x

1 Introduction 1

2 Literature Review 4

2.1 Memory Constraints 4

2.1.1 Disk-Based Swap 6

2.1.2 DRAM Alternative Technologies 7

2.1.3 Remote Memory Sharing 9

2.1.4 Memory Compression 10

2.2 Compression Algorithms 10

2.2.1 LZO 11

2.2.2 LZ4 11

2.2.3 Zstandard 12

2.2.4 Huffman Encoding 12

2.2.5 LZ77 18

2.2.6 LZW 20

2.2.7 Deflate 21

2.3 Open-Source Hardware 22

2.4 Existing Hardware Compressors and Decompressors 23

2.4.1 Existing Memory Compressors and Decompressors 23

2.4.2 Proprietary Compressor-Only Hardware 24

2.4.3 Proprietary Decompressor-Only Hardware 26

2.4.4 Open-Source Hardware Compressors and Decompressors 26

2.4.5 Open-Source Compressor-Only Hardware 27

2.4.6 Proprietary Complete Deflate-Like Hardware Compressors and Decompressors 28

3 Open-Source Hardware Solutions **30**

3.1 Overview 30

3.1.1 Open-Source 30

3.1.2 Parameterized Designs 31

3.1.3 Implementation Language 32

3.2 Huffman 33

3.2.1 High-Level Design Choices 33

3.2.2 Implementation Details 36

3.3 LZ77 42

3.3.1 High-Level Design Choices 42

3.3.2 Implementation Details 44
3.4 LZW 50
3.4.1 High-Level Design Choices 50
3.4.2 Implementation Details 52
3.5 Evaluation Methodology 54
3.5.1 Hardware Synthesis 54
3.5.2 IBM Area Estimation 55
3.5.3 Compression Ratio Comparison 55
3.5.4 Latency 56

4 Results and Analyses **57**
4.1 ASIC Area and Frequency Comparison 57
4.2 Latency Comparison 60
4.3 Compression Ratio Comparison 63

5 Limitations and Future Work **66**

6 Conclusions **69**

Bibliography **71**

List of Figures

2.1 Completed Huffman Tree Example 16

2.2 Open-Source Parameterization Comparison 27

3.1 Huffman Compressor Block Diagram 36

3.2 Huffman Decompressor Block Diagram 41

3.3 LZ77 Compressor Block Diagram 45

3.4 LZ77 Decompressor Block Diagram 49

3.5 LZW Compressor Block Diagram 52

3.6 LZW Decompressor Block Diagram 53

4.1 Open-Source Compressor and Decompressor Maximum Clock Speed Comparison 58

4.2 Parameterized Design Area Configurability Demonstration 59

4.3 Compressor and Decompressor 7 nm Design Area Comparison 60

4.4 Normalized Hardware Area Comparison 61

4.5 Compression Design Latencies 63

4.6 Decompression Design Latencies 64

4.7 Benchmark Memory Dump Compressibility 65

List of Tables

2.1 Huffman Compressed and Uncompressed Example String 14

2.2 Example Uncompressed Encoding 15

2.3 Character Frequency Count 15

2.4 Character Encodings 16

2.5 Huffman Canonical and Plain Text Tree Comparison 17

2.6 LZ77 Compressed and Uncompressed Example 20

3.1 Configurable Huffman Parameters and their typical values. 34

3.2 Example Tree Depth Annotation Output 38

3.3 Example Tree Normalization Output 39

3.4 Codeword Generator Encoding Example 40

3.5 Configurable LZ77 Parameters and their typical values. 42

3.6 CAM History Example 46

3.7 Pattern to Search For 46

3.8 Bit Array Output Example 47

3.9 Shifted Bit Arrays 48

3.10 Pattern Detection with Shifted Bit Arrays 49

3.11 Configurable LZW Parameters and their typical values. 51

4.1 Hardware Latencies of IBM Compression and Decompression Accelerators . 62

List of Abbreviations

ASIC Application-Specific Integrated Circuit

CAM Content-Addressable Memory

FPGA Field-Programmable Gate Array

HDL Hardware Description Language

IoT Internet of Things

LZ Lempel-Ziv

LZW Lempel-Ziv-Welch

ASIC designs are unconfigurable logic designed to accelerate a task.

CAMs allow the memory locations of particular bytes or byte sequences to be quickly found.

FPGAs are useful for accelerating applications without spending the resources to create an ASIC design, or for purposes where the design constraints may change over time. They are reconfigurable logic elements.

HDLs describe digital logic designs, similar to the way C++ code describes a computer program.

IoT is a class of objects that incorporate computers and electronics to allow them to network with other devices and operate or record data without human input.

LZ is a class of compression algorithms based on papers by Abraham Lempel and Jacob Ziv.

LZW is a number of modifications to the 1978 LZ compression algorithm made by Terry Welch.

Chapter 1

Introduction

From small IoT microcontrollers to warehouse-scale computers, memory is a valuable but limited resource in all modern computing devices. Memory compression is a technique that can enable significantly more effective memory usage by representing data in a more compact format. This reduces the amount of physical RAM a task needs, and it can improve performance in memory-bound applications by reducing the time spent swapping to a disk. Additionally, RAM requires a constant power draw to refresh its data, so using less physical RAM has the potential to lower power costs.

In cloud and data center configurations, where the architects often configure each CPU with the maximum amount of supported memory, memory can be more expensive than the CPUs used. The AMD EPYC 7763, the highest core count CPU currently on the market, costs $7890 [30]. This CPU can utilize up to 4 TB of memory [21]. Extrapolating from the price of the cheapest 64 GB RAM module currently available, $324, 4 TB of RAM would cost $20736, which is over 2.62x as expensive as the CPU itself [46].

Software RAM compression has existed for many years to reduce the need for physical memory. Linux, Windows 10, and MacOS all support software RAM compression [34, 49, 50]. However, software decompression adds nearly 75 microseconds of latency to memory read operations. Compared to the latency to read an uncompressed page of memory, 255 nanoseconds, decompression adds over 280x more latency. This additional latency can have a significant impact on performance. However, this latency can be significantly reduced if

compression and decompression are hardware-accelerated.

There are several existing implementations of hardware-accelerated compressors and decompressors, but these are not suitable for memory compression research [3, 20]. These designs have poor decompression latencies. They are also targeted towards file compression, which could result in lower effective memory available when using them. These designs are also exclusively proprietary and closed-source, so hardware memory compression research requires building entirely new hardware.

The closed-source nature of existing works makes it difficult to conduct memory compression research for a number of reasons. Most importantly, because there are no open-source hardware compressors and decompressors capable of state-of-the-art latencies, researchers must spend months, if not years, designing, implementing, testing, debugging, synthesizing, and optimizing the compression hardware. The researchers must do these things before they can incorporate the accelerator into a hardware design and use it for compression or decompression. The researchers must repeat this every time a new group is interested in memory compression research, adding significant unnecessary development costs to compression projects. By providing open-source tools, this work empowers the architecture research community to generate area, frequency, and power numbers for simulators, FPGA-accelerated simulations to improve the speed of simulation, and even tape-out chips in less time than ever before.

Existing designs are also very specialized. Applications with different requirements will necessitate different hardware implementations. A low-power microcontroller will require a much different compressor and decompressor than a server with hundreds of CPU cores. To address this need, either many hardware compressors and decompressors must be available, or designs should be highly parameterized. Parameterization allows for fast customization and exploration of the design space, and it allows a single hardware generator to fulfill the needs of many different use cases.

This work attempts to address the shortcomings of existing hardware compressors and decompressors for use in operating system memory compression. This work implements and tests three hardware compressor and decompressor pairs. All implementations are open-source under a permissive license, so individuals, research groups, or enterprises can use and modify them as needed. Every aspect of the design is completely parameterized, from the amount of data compressed by the accelerators to the number of bits in a byte, allowing for complete customizability of the design. Area- or power-constrained systems can choose to use individual compressors and decompressors, and high-performance designs can choose to use multiple compressors and decompressors together for higher compression ratios. The compressors and decompressors can achieve sub-microsecond decompression latency, even while demonstrating compression ratios comparable to those of software implementations. All of these characteristics mean that this work improves the state of the art for hardware-accelerated memory compression.

The compressors and decompressors in this work are capable of several times lower latency than the existing state of the art hardware compression and decompression accelerators. These hardware compressors and decompressors also use an order of magnitude less area than the existing state of the art. Finally, the compressors and decompressors in this work accomplish all of this while being fully configurable and open-source for anyone to use in the future.

The rest of this document describes this work and compares it to the existing state-of-the-art. Chapter 2 discusses the existing literature relating to memory compression. Chapter 3 discusses the designs that comprise this work and the methods used to compare it to the existing state of the art. Chapter 4 presents the results of the comparison to existing works. Chapter 5 discusses some limitations of this work and identifies potential future projects based on this work. Finally, Chapter 6 concludes with the broader implications of this work.

Chapter 2

Literature Review

This chapter reviews relevant literature on hardware compression and decompression for memory. Section 2.1 gives background on the limitations of DRAM and many of the attempted mitigation strategies. Section 2.2 discusses current state-of-the-art compression algorithms, their benefits, and their differentiating features. Section 2.3 provides background on the current state of open-source hardware and software for compression and motivates the open-source aspect of this work. Section 2.4 describes the current state-of-the-art in hardware compression and decompression and existing limitations.

2.1 Memory Constraints

Due to the proliferation of Big Data and AI, the available system memory constrains many workloads. In their survey of in-memory applications, Zhang et al. say, 'In spite of significant increase in memory size and sharp drop in its price, it still cannot keep pace with the rapid growth of data in the Big Data era, which makes it essential to deal with data overflow where the size of the data exceeds the size of main memory [...]' [59]. Researchers have been working hard to handle these data overflow cases for decades, so a large number of mitigation techniques exist. These techniques include swapping memory data to a disk, using new memory technologies to supplement DRAM, sharing RAM with remote computers, and compressing in-memory data to take up less space.

For all of the technologies discussed in this section, there are some important characteristics that affect performance. These are the latency, bandwidth, and capacity. Each of the technologies reviewed below has at least one characteristic that is better than DRAM.

Latency is the amount of time between memory data being requested and the transfer of data completing. DDR4 at 2133 MHz and a CAS latency of 15 will have a latency of 255 ns to read a 4 KB page of data. Any technology that is slower than this will potentially slow down applications, because the computer will need to wait longer for data than it would have if it were instead using DRAM.

Bandwidth is the amount of data that can be transferred in a given time frame. Bandwidth is significant because it can limit the number of reads and writes that can occur in a set span of time and because it affects latency. If bandwidth is very low, it will make 4 KB memory page accesses take longer than they do with DRAM, even if a technology's latency to access the first byte of data is very low. DDR4 at 2133 MHz with a CAS latency of 15 will have a bandwidth of 17 GB/s. Technologies with lower bandwidth than this could negatively affect system performance.

Capacity is the amount of data that can be stored with a given technology. While most technology capacities can be increased by purchasing more of it, it is useful to think of this as the cost-effective capacity. In other words, how much capacity a given technology can store at a reasonable price. The goal of all of the mitigation strategies is to have a higher capacity than DRAM alone. If there is not enough memory capacity, not only will the application be slower, the application will be unable to complete at all. Capacity is the motivation behind all of the following technologies.

2.1.1 Disk-Based Swap

A very simple way to increase the capacity of memory is to store infrequently-used memory pages on a disk. By doing this, the effective memory capacity can be significantly improved. Jung et al. and Lin et al. both implement flash-aware swap in the Linux kernel [25, 33]. In an attempt to benefit from the latency benefits of flash and the cost-effectiveness of hard disks, Liu et al. use a hybrid approach for improved QoS when swapping [35].

To utilize less of the disk and improve performance, Cervera et al. and Cortes et al. implement a disk-based swap that stores the data in a compressed form [12, 16]. This improves the bandwidth of the swap, makes it more accessible to devices with small disks, and reduces the latency of read accesses.

The disk-based swap approaches are some of the most cost-effective, because 1 TB of space on an HDD costs less than $100 currently. However, it can cause some of the worst performance of any of the strategies. Latency for data accesses on an HDD is many milliseconds, which is several orders of magnitude longer than the read latency of DRAM. The bandwidth is also several orders of magnitude lower as well. SATA 3 has a maximum bandwidth of 6 Gbps, and most hard drives are not even capable of that.

However, latency and bandwidth are both significantly better on PCIe-based NVMe SSDs. Although they are currently more expensive per GB, their cost is lowering over time, and many PCIe drives can achieve bandwidths of multiple gigabytes per second. Some devices, like Intel's Optane Persistent Memory drives, can also achieve latencies of as little as one microsecond, bringing performance significantly closer to that of DRAM. These faster modern devices make disk-based swap a much more attractive solution than it once was.

2.1.2 DRAM Alternative Technologies

There are several technologies designed to compete with DRAM. These technologies hope to provide lower power or higher capacities at latencies and bandwidths as close to that of DRAM as possible. The technologies discussed here are Phase-Change Memory (PCM), Resistive RAM (RRAM), and Magnetoresistive RAM (MRAM).

PCMs

PCMs use the amorphous or crystallized state of a material to store data [42, 57]. The amorphous state, due to its irregular structure, has a high resistance, and the crystalline state has a lower resistance. Heating the material in different ways can change the phase of the material. By heating the material with a quick pulse of voltage, the material cools quickly, forcing it into an amorphous state. By heating the material more slowly, the material is able to slowly crystallize into the crystalline state.

On a macroscopic scale, waiting for a physical heating and cooling process would be quite slow. Latency is a primary concern for a technology such as PCM. However, there are documented cases of PCM that can recrystallize in under 5 ns [10]. Additionally, although DRAM and other storage technologies have had poor scaling to smaller process nodes, PCM has the potential to become faster and more efficient at smaller sizes, because there is less material to heat and cool and a higher area to volume ratio to the material [10]. This means that, if manufacturing technologies are able to produce high volumes, PCM is a promising-looking technology to replace or supplement DRAM. Unfortunately, PCM will likely be expensive and time-consuming to add support for, as DRAM currently dominates.

RRAMs

RRAMs work by changing the resistance of a material with an electric current. This resistance stays until it is rewritten, so it can be used to store data. RRAM devices use two metal electrodes with an insulating oxide layer between them [60]. When a current is applied to the oxide layer, it creates or destroys a thin conductive filament in it. This reduces the resistance of the layer, allowing electrons to flow through more easily. By attempting to pass electrons through this layer, the state of the RRAM device can be found, allowing stored data to be read.

Like many of the other technologies in this section, RRAM is capable of low-voltage operation, does not require regular refreshing, and is also capable of areas, speeds, and latencies that are competitive with DRAM [58]. However, just like all of the other technologies, the largest hurdles are changing existing hardware implementations to utilize RRAM instead of DRAM.

MRAMs

MRAMs are commonly implemented with two technologies: Giant Magnetoresistance (GMR) and Magnetic Tunnel Junction (MTJ). GMR magnetoresistance MRAMs use a property of thin-film magnets to store data. This property, GMR, causes the electrical resistance to change based on the alignment of magnetic moments in stacked thin-film materials [53]. MTJ magnetoresistance MRAMs use two electrodes separated by a thin dielectric, like a capacitor [53]. If the dielectric is made thin enough, electrons can tunnel through it, but by using ferromagnetic materials for the electrodes, the orientation of the magnetizations of the electrodes compared to each other can change the resistance of the MTJ [53]. Both of these MRAM technologies use their respective magnetic orientations to store data as resistance

values.

MRAM has a number of benefits. It is nonvolatile, which means it does not require power to retain its value. MRAM does not require constant refreshing like DRAM, which also reduces its power usage. It is smaller than SRAM, and fast enough to be competitive with DRAM for program memory [23, 52]. However, like the other DRAM replacements, support for this technology will likely require extra hardware or software support when compared to the already very mature DRAM hardware stack in modern computers.

2.1.3 Remote Memory Sharing

Remote memory sharing is the idea that data can be stored in the memory of other computers on the same network. This would not be effective in a context without a fast, low-latency network connection or where all the nearby computers are memory constrained. However, in many of the contexts where large amounts of memory are necessary, like cloud computing or warehouse-scale computing, there are often imbalances of memory utilization. This can occur more than 70% of the time for certain workloads [22].

In contexts where it is applicable, remote memory sharing can be very beneficial as a supplement to DRAM. Gu et al. estimated that cluster vendors could increase revenues on the same hardware configurations by up to 24.7% by enabling Infiniswap, their remote memory sharing implementation [22]. Similarly, Liang et al. found that InfiniBand remote memory had latency within an order of magnitude of the latency of DRAM for memory copies and that quick sort only runs 1.45x slower with remote memory than with local DRAM [32]. It is clear that, in the right situations, remote memory can be very useful, but its effectiveness is limited by context.

2.1.4 Memory Compression

Memory compression, the idea that portions of memory that are not in use can be compressed to take up less space, has been around for many years. Often, memory compression works in a similar way to a normal swap device like a hard drive: RAM data that has not been recently used is stored somewhere else. However, instead of putting this data on a disk, the data is compressed, and then stored back in RAM in a compressed form. Then, if a read operation to that memory occurs, it can be decompressed and removed from the compressed virtual disk.

The benefit of compressing memory but keeping it in RAM in a compressed state is significantly reduced access latency when compared to disk accesses. Additionally, because the RAM data is compressed and still inside RAM, there is no need for support for new hardware technologies, memory compression can be used on any computer with software support. Kjelsø et al. and Roy et al. investigate the performance implications of this technique [28, 45]. Memory compression was found to speed up application performance 5-250% when compared to disk-based swap in these works. While this is significant, Kjelsø et al. estimate that hardware compression instead of the implemented software compression could yield many times higher performance [28]. These hardware compressors, the primary focus of this paper, will be discussed in Section 2.4.

2.2 Compression Algorithms

To discuss hardware compression accelerators, it is first useful to have an understanding of compression algorithms. There are two types of compression algorithms: lossy and lossless. Lossy compression algorithms are often able to achieve much higher compression ratios, but

lossy compression changes the data being compressed in some way. For memory compression, which might hold important data, pointers, or programs, the changes that result from lossy compression are not acceptable. For this reason, lossless compression is the focus of most works in the field of memory compression.

The algorithms discussed in this section are LZO, LZ4, Zstandard, Huffman, LZ77, LZW, and Deflate. This discussion is in preparation for discussion of hardware accelerators for these algorithms in Section 2.4.

2.2.1 LZO

LZO is a unique compression algorithm in that no official public specification is available for it. Only the source code and an attempt at documenting it in the Linux kernel can be used to find information about the algorithm [51]. However, there is still enough information to gain a simple understanding of its characteristics. LZO is a variation on LZ77 compression by Markus Oberhumer. Its goal was to be faster than Deflate compression software like Gzip, which uses Deflate [38]. LZO accomplishes this goal, but at the cost of lower compression ratios. This is partially due to its lack of Huffman encoding, which Deflate includes. More information about LZ77 is included in Section 2.2.5.

2.2.2 LZ4

LZ4, like LZO, is based on the LZ77 compression algorithm. It encodes data in a byte-aligned manner. This allows its outputs to be compatible as inputs to algorithms like Huffman compression, discussed in 2.2.4.

Each block of LZ4-compressed data consists of a byte of metadata, a variable number of

optional bytes of metadata based on the first byte, a number of literal bytes that are not compressed, and a compressed match [14]. The metadata at the beginning of the block informs how long the literal data is and how long the match is. Once the literal data is added to the data history, the match index, found at the end of the block, can be combined with the metadata to match a sequence of characters in a row from the data history. This allows repeated patterns of characters to be re-used without being repeated multiple times, which is where the compression benefits come from. LZ4 is essentially a specific way to encode LZ77, which is discussed in Section 2.2.5

2.2.3 Zstandard

It is one of two compression algorithms discussed in this paper that is a combination of existing algorithms. It combines LZ77 compression, Huffman encoding, and finite state entropy encoding [15]. LZ77 compression, discussed in detail in Section 2.2.5, is performed on the data first. Afterwards, literals that could not be compressed are compressed with Huffman encoding, described in Section 2.2.4. The Huffman metadata and all other data is then compressed with finite state entropy encoding, which allows it to achieve very competitive compression ratios. However, because of this complexity, it is not often implemented in hardware.

2.2.4 Huffman Encoding

Huffman encoding achieves minimum redundancy of data by giving characters of data variable lengths [24]. The lengths of each encoding are determined by how frequently it appears in the data being compressed. Frequently-occurring data are assigned shorter encodings, and infrequently-occurring data are assigned longer encodings. In this way, data can be

optimally encoded such that no two characters' representations could be swapped to achieve a shorter encoding result.

To accomplish this, Huffman encoding arranges all of the unique characters in the data into a binary tree, where each character is a leaf node. Huffman encoding uses this tree to find the most efficient encoding for each character in the data. There are two types of Huffman encoding: static and dynamic.

Static Huffman encoding uses a premade Huffman tree to encode the data. This reduces its flexibility, because the static tree will work very well for certain inputs and poorly for others. However, using a premade tree removes all of the work required to create a tree, which simplifies and speeds up the algorithm to encode the data.

Dynamic Huffman encoding is done in two passes. In the first pass, the encoder counts the number of occurrences of each data character to determine their relative frequencies. Afterwards, the encoder uses these frequencies to construct a binary tree where each data character is a leaf node. The construction of this tree starts from the two least frequent characters. Every time two nodes are combined under a new parent node, the parent node's frequency becomes the sum of the frequencies of its children. This continues until a single root node encompasses the entire tree.

Once the tree is constructed, the lengths and encodings of the characters can be determined. The number of bits used to represent a character is equal to its depth in the tree from the root node. The encoding of each character is determined by the traversal steps required to reach it from the root node. Every left path in the tree concatenates a '0' to the binary encoding of the character, and each right path concatenates a '1' to the binary encoding.

Finally, once each of the leaf nodes has been given an encoding, the second pass can encode the data. For each character of the input data, the second pass looks up its new encoding

and outputs this new encoding. This continues until every character of the input has been encoded.

To decode Huffman-encoded data, only the original Huffman tree is needed. Then, the decompressor can simply traverse the Huffman tree for each bit in the encoded data, stopping and starting from the root node every time it reaches a leaf. Including the original Huffman tree with the compressed data can have a significant impact on the compression ratio. To combat this impact, most Huffman compression implementations represent the tree in a canonical form. A canonical Huffman tree represents the data as the number of characters of each depth, then lists the characters in order from the lowest depth to the highest depth. Canonical form is one of the most efficient ways of representing the Huffman tree, and it can be several times smaller than an equivalent tree represented in 'plain-text' form, listing each character, the length of its encoding, and the encoding itself.

Example

For the purposes of demonstration, the string of characters being compressed for Huffman are shown in Table 2.1.

Table 2.1: Huffman Compressed and Uncompressed Example String

Uncompressed	A	A	A	B	A	A	C	B	B	A	A	B	D	A	A	B
Compressed	0	0	0	10	0	0	110	10	10	0	0	10	111	0	0	10

Without using Huffman encoding, each of these characters could be encoded with only two bits. An example of the encodings that could be used to do this is shown in Table 2.2. Because there are 16 characters in the example, it would require a total of 32 bits to encode this data.

The first step in dynamic Huffman encoding is to count and sort the frequency that each

Table 2.2: Example Uncompressed Encoding

Character	Binary Encoding
A	00
B	01
C	10
D	11

character occurs. The result of the character frequency counting step is shown with the frequencies sorted in Table 2.3.

Table 2.3: Character Frequency Count

Character	Number of Occurrences
A	9
B	5
C	1
D	1

Once the sorted list of frequencies is complete, the compressor can build the Huffman tree. To build the Huffman binary tree, create a parent node and set its children to be the two least frequent nodes in the list of existing nodes. Remove the two nodes that were set as children from the list of characters remaining, add their summed frequency values as a new parent node to the list, and re-sort the list by frequency. This step repeats until there is only a single node remaining in the list. This is the root node. Figure 2.1 shows a completed Huffman tree.

Once the Huffman tree is completed, the standard Huffman algorithm is nearly finished. A codeword can be generated for each character in the tree based on its position in the tree. Starting with the root node, the path of each sequence of left and right children reveals the encoding of a character. Each left appends a 0 to the encoding, and each right appends a 1.

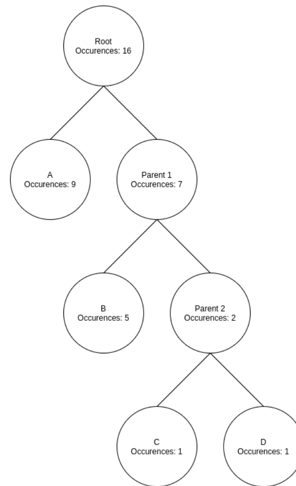

Figure 2.1: Completed Huffman Tree Example

It is useful to note here that each character's depth in the tree is equal to the number of bits in its encoding. A character encoding based on the tree in Figure 2.1 is shown in Table 2.4.

Table 2.4: Character Encodings

Character	Character Encoding	Character Encoding Length
A	0	1
B	10	2
C	110	3
D	111	3

Now, the only task remaining is to encode each character using the codewords shown in Table 2.4. The result of the final character encoding step is shown below in Table 2.1.

The resulting bit sequence shows a significant number of bits have been saved. The original encoding would have required 32 bits, but this version only needs 25 bits. This simple example results in a compression ratio ($\frac{initialdatasize}{compresseddatasize}$) of 1.28. This is unusually short compared to how Huffman compression would normally be used, so adding the storage overhead from the Huffman tree will remove any compression benefits from Huffman compression. However, for completeness' sake, both a canonical and a plain text Huffman tree example are

demonstrated below.

For a canonical Huffman tree, there are many methods of encoding the data. The simple way that is shown here first lists the number of each character at each depth, then lists each character in order from shallowest to deepest. Because this example uses only four characters, the deepest a leaf node could go in a Huffman tree is a depth of 3. For each depth of the Huffman tree, the number of bits needed to represent the number of characters at that depth is $1 + depth$. Because the Huffman tree can have characters at a depth of up to three, to represent all the depths, the first part of the canonical Huffman tree is as follows: two bits for the number of 1-bit characters, three bits for the number of 2-bit characters, and four bits for the number of 3-bit characters. This sums to a subtotal of nine bits for the depths of the canonical Huffman tree. After the depths, each character is listed. Because the characters can be represented uncompressed as 2-bit values, the character section of the canonical Huffman tree can be anywhere from two to eight bits, depending on how many characters are in the Huffman tree. Table 2.5 below is the binary representation of the canonical Huffman tree of the encodings in Table 2.4. Note, this encoding is seventeen bits, while the compressed version of the data only saved seven bits, resulting in an increase in size of ten bits.

Table 2.5: Huffman Canonical and Plain Text Tree Comparison

HC	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	1	1											
HPT	0	1	0	0	0	0	0	1	0	0	1	0	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1

A plain text Huffman tree is very simple to implement. The most basic method is to represent each character as the number of bits in its encoding, the encoding itself, and the uncompressed character. The current example requires two, three, and two bits for the number of bits, the encoding, and the uncompressed character, respectively. For four characters, that means the plain text form will take up 28 bits, significantly more than the

canonical encoding. If the Huffman tree didn't use a character, this is demonstrated by setting its encoding length value to zero for the plain text representation. Table 2.5 below is the plain text representation of the encodings in Table 2.4.

More details on Huffman encoding and canonical Huffman trees can be found in [18, 24].

2.2.5 LZ77

While Huffman compression is effective at encoding individual characters, it has limitations that motivate the use of other compression algorithms. Huffman compression with a dynamic tree requires two passes, which requires buffering and knowledge of the total data length for uses like file compression. Additionally, because Huffman only takes advantage of individual characters that occur frequently, its utility is limited by the size of the Huffman tree and the original size of each symbol. For instance, using 8-bit characters results in a manageable tree size of 256 characters, but the maximum achievable compression ratio is 8x when a character can be represented with a 1-bit code. Using 16-bit characters doubles the maximum achievable compression ratio to 16x, but increases the size of the Huffman tree to 65,536 characters. Using symbol sizes that are unaligned to the sizes of characters, such as 9- or 10-bit symbols, results in poor compression ratios due to misalignment of symbols to byte boundaries.

LZ77-based algorithms provide a solution to the shortcomings of Huffman encoding. LZ77-based compression techniques use an algorithm pioneered by Abraham Lempel and Jacob Ziv in 1977 [61]. When patterns of characters are repeated in data, LZ77 techniques use references to previous instances of the patterns to represent the data more efficiently. These references are bounded in a sliding window of the most recent data to pass through the compressor. By compressing repeated patterns of characters, maximum compression ratios

can be much higher. Additionally, by compressing based on the history of the data, LZ77 avoids needing to send metadata like Huffman does.

For LZ77 compression, a sliding window of a fixed size stores the past history of the compressor. As characters are fed into the compressor, the compressor checks to see if each sequence of characters it receives already exists in the sliding window. If it does, instead of representing the characters as literal characters, it represents them as a pointer-length pair. This pair contains both a pointer to the starting position of a character sequence in the sliding window and the length of the matching character sequence. In this way, even very large sequences of repeated data can be compressed down to only a few bytes.

When LZ77 data is decompressed, the decompressor also keeps track of a sliding window of the data that has been decompressed. Whenever the decompressor receives a pointer-length pair, the decompressor looks up the pointer and reads a number of characters after the pointer equal to the length of the pair. This encoding does not require sharing any metadata between the compressor and decompressor, they both automatically build up identical histories as they process the data.

Example

The LZ77 example will use the string shown in Table 2.6 to demonstrate the result of LZ77 compression. As with the Huffman example, the characteristics of the example like the compression ratio are not necessarily indicative of the average compression ratio that is achieved by LZ77 on most inputs. However, the example is useful for demonstrating the general format of an LZ77 implementation.

Table 2.6 also shows an example of compressing the example text with LZ77. Each bracketed entry represents a character pattern that has already occurred in the history and that is three

characters or longer. Three characters was chosen because this work requires at least three characters in a pattern for the pattern encoding to be less than or equal to the length of the pattern itself. Other variants of LZ77, like LZ4, choose four characters for this cutoff, but the same principles still apply [14].

Each pattern shown in the compressed data of Table 2.6 follows the form {pattern starting index, number of characters}. So, for example, {3,6} represents the pattern starting at index three that is six characters long: ' quick'.

Table 2.6: LZ77 Compressed and Uncompressed Example

Uncompressed	The quick brown fox jumps over the lazy dog. The brown fox is very quick, and the lazy dog's nap is over.
Compressed	The quick brown fox jumps over the lazy dog. {0,4}{10,10}is very{3,6}, and{30,13}'s nap is{24,5}.

2.2.6 LZW

LZ78-based compression techniques such as LZW instead use an algorithm published by Lempel and Ziv in 1978, after their first seminal paper on compression [62]. Like LZ77, LZ78 works by representing sequences of characters as references, but instead of representing them as pointer-length pairs, LZ78 represents references as dictionary entries. These dictionary entries are built by the compressor and rebuilt by the decompressor as the data is read, so there is no need to transmit a dictionary separately.

LZW is a modification of LZ78 published by Terry Welch in 1984 [55]. LZW was developed as a simpler, faster algorithm for compression than existing LZ77 or LZ78 algorithms, but with the goal of still taking advantage of many of the redundancy opportunities that LZ77 and LZ78 take advantage of [55]. LZW is able to take advantage of the same character

repeated multiple times in a row and of sequences of characters that occur multiple times in the data.

When compressing with LZW, every sequence of inputs is added to a dictionary. Initially, the dictionary contains as many entries as there are possible unique characters. Then, as data is streamed through the LZW compressor, it adds entries to the dictionary. Anytime a sequence of two or more characters is no longer able to be found in the dictionary, it is added to the dictionary, and the sequence of characters starts from the final character in the sequence that caused it to not match any existing entries. This continues for all of the input characters, and then each character output from the dictionary must be the number of bits required to represent the number of entries in the dictionary. This means that as the dictionary gets longer, the length of each output of the compressor also gets larger to allow it to represent all the valid indices in the dictionary. More information and examples of LZW can be found in Welch [55].

LZW was implemented in this work as an alternative to LZ77 for combining with Huffman to make a hardware Deflate accelerator. However, this idea was not utilized, because LZW + Huffman performance was worse than either in isolation for all of the tested memory compression workloads. Despite this, because LZW was already implemented, it will be included with the other compressors and decompressors in this work due to its area efficiency and simplicity.

2.2.7 Deflate

Deflate is a combination of LZ77 and Huffman encoding [17]. First, byte-aligned LZ77 compression is performed on the input data. Next, this result is compressed with Huffman. To decompress, Huffman decompression is first applied, followed by LZ77 decompression.

By combining these two algorithms, Deflate is able to take advantage of both the pattern matching of LZ77 and the improvements in character representation of Huffman encoding. Because it takes advantage of both, its compression ratios are higher than either individually. These high compression ratios make it a good candidate for memory compression.

Deflate's high compression ratios and Deflate's use of LZ77 and Huffman compression are the motivation for this work implementing LZ77 and Huffman hardware accelerators. The LZ77 and Huffman hardware accelerators can be combined to implement a Deflate accelerator, improving compression ratios significantly.

2.3 Open-Source Hardware

Some of the most successful, widely-used software in the world is open-source. In listing some of the most popular open-source projects, Fitzgerald lists some of the largest ever software projects: the Linux kernel, the Apache web server, the Firefox and Chromium browsers, the GNU C compiler, the Perl and Python scripting languages, and MySQL [19]. These each revolutionized their fields, becoming some of the most popular tools in existence, even to this day.

Open-source software has been a world-wide phenomenon. SourceForge, a website for open-source software projects, lists over 199,000 open-source software projects. Some of the largest software companies, Google, Facebook, Microsoft, and Amazon, have released dozens of open-source projects between them, and they provide financial support to dozens more.

Open-source hardware has a huge potential for success. The open-source RISC-V instruction set has resulted in an explosion of open-source CPU cores, including offerings from Western Digital, Berkeley, Princeton, and dozens more [8, 11, 40]. Nvidia open-sourced the design

for a deep-learning accelerator, NVDLA. FPGA tools, which for a long time have been proprietary, are being replaced with open-source synthesis suites for Xilinx, Microsemi, and Lattice FPGAs. It is now possible to build entire SOCs with fully open-source components [26].

Unfortunately, though, hardware has not seen the same proliferation of open-source designs. Between OpenCores.org and LibreCores.org, two open-source hardware hosting websites, there are less than 1000 projects. There are less than 2000 Verilog projects currently listed on GitHub. Despite the many academic hardware compression works found in Section 2.4, none of them are open-source.

2.4 Existing Hardware Compressors and Decompressors

This section contains discussion of both proprietary and open-source implementations of compressors and decompressors. The proliferation of proprietary implementations and the dearth of open-source frameworks for compression/decompression further motivate our approach.

2.4.1 Existing Memory Compressors and Decompressors

The goals of Compresso are similar to the goals of this work: enabling low-latency memory compression to increase effective memory size [13]. However, because Compresso only compresses blocks of 64 bytes, it is only able to achieve compression ratios of 1.85. In this work's testing, even excluding outliers with compression ratios of 60x, Deflate compression ratios are 3.3 on average. Compresso's mediocre compression ratios make it unsuitable for

aggressive compression of memory data.

Like Compresso, IBM's MXT technology attempts to implement a low-latency memory compressor in hardware [2]. Although MXT succeeds at this, it is only capable of compression ratios up to 2.68 in the best case shown, which severely limits its usefulness.

Benini et al. utilize hardware acceleration for memory compression, but makes no mention of latency. However, Benini et al. mention that a future work is to implement LZ-like compressors due to their higher compression ratios [9]. Although Benini et al. do not publish the compression ratios their design achieves, this suggests that it is significantly worse than LZ-based compression.

Pekhimenko et al. implements a low-latency hardware compression accelerator for main memory, but compression results are significantly worse than either IBM's MXT hardware or LZ compression, both of which give significantly lower compression ratios than that of software Deflate [39].

2.4.2 Proprietary Compressor-Only Hardware

This section discusses proprietary hardware accelerators that only implement the compressor, relying on software to perform the decompression.

ASIC Designs

Lee et al. verify an LZ4 compressor design in an FPGA, then fabricate it on a 65 nm process node [31]. However, this design can only compress patterns as large as 31 bytes, which limits the compression ratios that it can achieve.

FPGA Designs

Rigler et al. implements separate hardware compressors that use Huffman and LZ77 [44]. The designs are separate, but the authors claim that it is possible to merge the designs together to implement GZIP. Unlike many designs, the Huffman compressor of this design supports dynamic Huffman encoding, which significantly complicates the hardware.

Abdelfattah et al. use OpenCL to implement GZIP on an Altera FPGA [4]. This GZIP compressor supports dynamic Huffman encoding and achieves compression ratios over 2x. Abdelfattah et al. used high-level synthesis tools to implement this design. High-level synthesis tools have a reputation for producing slow, inefficient designs. However, the OpenCL implementation is shown to be very close to a hand-programmed Verilog design in area, speed, and compression ratio.

Luo et al. implement a multi-core GZIP compressor for use with the Hadoop Distributed File System [36]. This work uses static Huffman encoding. The hardware implementation enables a speedup of 70x compared to software compression, which doubles the performance of HDFS.

To address CPU-FPGA communication speed limits on FPGA compressors, Qiao et al. implement a high-throughput Deflate compression accelerator [41]. Their design uses the extremely high-speed communication of Intel's HARPv2 platform to overcome existing communication bottlenecks for FPGAs. Like many other compressors, they modify the hash table to allow it to work in parallel and use a static Huffman tree to allow Huffman to easily pipeline with the LZ77 compressor.

Ribeiro implements an LZ77 compressor and a static Huffman encoder on an FPGA with a general-purpose CPU that passes data between them and manages them [43]. The CPU of the Zynq devices that the design is implemented on limits the work to 123 MiB/s, and

achieves a speedup of 1.5x over the Zynq ARM processor alone.

2.4.3 Proprietary Decompressor-Only Hardware

To the best of my knowledge, Ledwon et al. implement the only relevant work to include a Deflate decompressor without a matching compressor [29]. Ledwon et al. implement their work in C++ and synthesize it with Xilinx's high-level synthesis tools. Because their work is implemented in software, their decompressor supports the full Deflate specification. This is uncommon in other hardware designs due to the complexity of implementing the full specification in hardware.

2.4.4 Open-Source Hardware Compressors and Decompressors

This section discusses open-source hardware accelerators that have a matching compressor and decompressor.

FPGA Designs

Nunez-yanez created the open-source X-MatchPROvw, implementing the XMatchPro compression algorithm [37]. This hardware compressor and decompressor is a dictionary-based pattern compression technique, and the design is capable of four bytes per cycle compression and decompression throughput.

Vijlbrief implements a configurable Deflate compressor and decompressor designed for FPGAs [54]. However, this design requires multiple cycles per compressed or decompressed byte, so it is not suitable for low-latency compression and decompression of memory. Additionally, although the HDL-deflate design is partially parameterized, it has many fewer 'knobs' for

parameterization than this work, as can be seen in Figure 2.2. Note that, because this work does not implement a Deflate design by combining the LZ77 designs and Huffman designs, the hypothetical Deflate shown in Figure 2.2 uses the sums of the parameters of the Huffman and LZ77 designs.

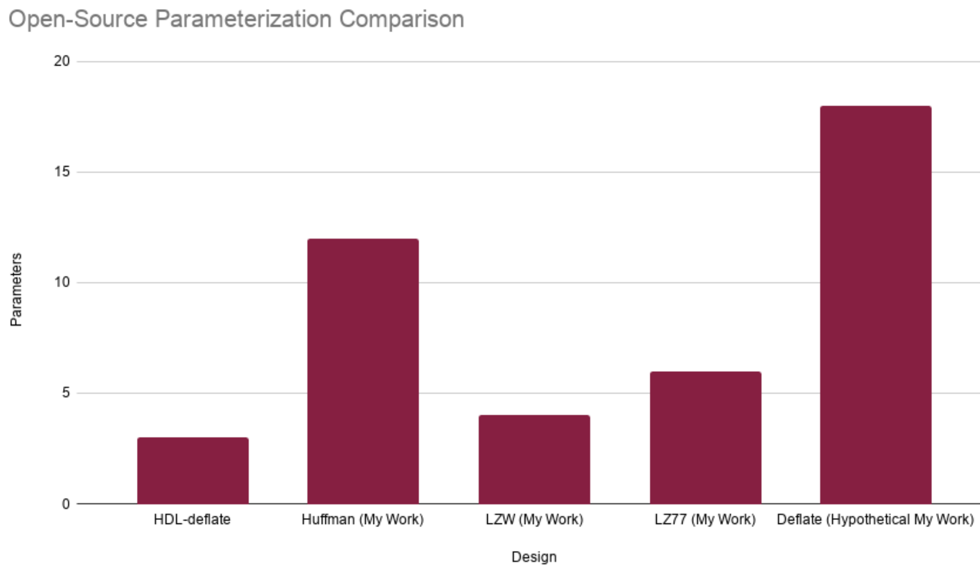

Figure 2.2: Open-Source Parameterization Comparison

2.4.5 Open-Source Compressor-Only Hardware

This section discusses open-source hardware accelerators that only implement the compressor. These rely on a software implementation of the decompression algorithm.

FPGA Designs

Schrittwieser implements a LZRW1 compressor core [48]. This core requires two cycles per byte and does not have a matching decompressor, so it is not suitable for low-latency compression and decompression of memory.

2.4.6 Proprietary Complete Deflate-Like Hardware Compressors and Decompressors

This section discusses proprietary hardware compressors and decompressors that implement Deflate-like algorithms. For the purposes of this discussion, a Deflate-like algorithm is one that uses LZ77 compression followed by Huffman compression.

ASIC Designs

IBM created a Deflate compression and decompression engine that is included in both their POWER9 and z15 processors [3]. IBM's implementation is capable of handling data at a rate of 8 bytes/clock, allowing for very high compression and decompression bandwidth [3]. The IBM design uses completely accurate near-history CAM and a pseudo-CAM for the older sliding window history to limit CAM area. Additionally, IBM's Huffman encoder builds dynamic trees, unlike many other works that build in a static Huffman encoding to remove the required tree-building process.

The AHA Products Group sells multiple PCIe cards that implement GZIP, which uses Deflate [5]. While there is not much published information about these cards, they support Deflate, GZIP, ZLIB, and LZS compression, and they scale from 10 Gbps to 80 Gbps compression throughput.

FPGA Designs

Microsoft implements a compressor and decompressor capable of Deflate [20]. The LZ77 compression of this Deflate accelerator is slightly modified to allow it to execute LZ77 in parallel, and the Huffman encoder and decoder uses a static Huffman tree. This design also

utilizes over 20 pipeline stages to enable its high clock speeds on an FPGA.

Microsoft also implements its own Xpress9 compression algorithm [27]. Xpress9 is very similar to Deflate, with an LZ77 stage and a Huffman stage. This design is intended to accelerate file compression and decompression in large-scale storage servers. Like Microsoft's other compressor and decompressor, this design processes multiple bytes in parallel to improve the bandwidth of the design. However, unlike Microsoft's Deflate implementation, the Xpress9 design uses a Nios II FPGA soft-core to compute the Huffman encoding. 7 of this design on an FPGA are able to support up to 128 threads of data at 2.4Gbps.

Chapter 3

Open-Source Hardware Solutions

This chapter reviews the contributions of this work. It contains arguments for what makes the work valuable and details about how the work was implemented. It also discusses the high-level design choices and the reasons they were made.

3.1 Overview

The proposed work consists of three separate pairs of open-source, parameterized compressors and decompressors. The whole work is open-source under the MIT license, so it can be reused. All designs in the work are also entirely parameterized to allow for easy configuration. The first set of compressors and decompressors implements a heavily modified Huffman encoding, the second implements a very slightly modified LZW, and the last implements a modified LZ77.

3.1.1 Open-Source

As Section 2.3 discussed, there is a clear need for more open-source hardware compressor and decompressor designs. This is especially true for low-latency designs. For this reason, all of the designs, scripts, and workflow for this work will be open-sourced under the MIT license. The MIT license gives nearly unlimited freedom to any group or indi-

vidual to use, modify, or sell the designs in any way they may desire. This work can be found at https://github.com/cjearls/Open-Source-Parameterized-Low-Latency-Aggressive-Hardware-Compressors-and-Decompressors.

In the spirit of open-source, all of the tools used to debug, test, and simulate the work are fully open-source as well. This means that anyone can recreate nearly every aspect of this work for free, without paying for any expensive licenses. The only proprietary software used by this work is the Synopsys Design Compiler, which was used for synthesis of the design to find frequency and area estimates. Although there are currently open-source projects targeting synthesis, they were still in development when this work began, so they were not considered.

3.1.2 Parameterized Designs

Each of the designs in this work are fully parameterized. This means that every aspect of the design is controlled by a parameter. Every single number in the source code of the design is either a configurable parameter or derived from one or more configurable parameters. Even the number of bits in a byte is configurable in this design. This provides a number of benefits.

It is important to note that, unlike with software, fully parameterizing the design is very uncommon in the world of hardware. If hardware is parameterized at all, it is normally limited to a small number of macros that control the sizes of buffers or other structures. Vijlbrief is a good example of this, as the design does have three parameters, but these parameters only control the sizes of buffers and a search window [54]. The three designs that comprise this work have over 20 parameters in total, and this allows design configurations with frequencies spanning several hundred MHz and areas that are over an order of magnitude

different. Not only are the sizes of buffers different between configurations, so are state machine characteristics, numbers and sizes of arithmetic operators, and even the numbers of many modules that are instantiated. This new level of parameterization separates this work from any existing hardware compressors or decompressors.

The first, most important benefit of fully parameterized designs is that the designs are easily configurable. This allows the designs to more easily fit area, frequency, power, and latency constraints without additional tweaking. This means that these designs can potentially be used for tiny microcontrollers and massive data centers without modifying the source code.

Another benefit of the parameterized nature of the designs is the speed of design space exploration. Because everything is parameterized, multiple configurations of the designs can be simultaneously tested for compression ratio, frequency, area, and power. This speeds up the development cycle significantly, reducing the human effort needed to make decisions on design trade-offs.

3.1.3 Implementation Language

This work was initially started in Verilog. However, after a brief trial run switching a module over from Verilog to Chisel, every component of the design was implemented in Chisel. Chisel has a number of benefits over Verilog. Verilog required 2-3x more lines of code on average compared to Chisel. Chisel was significantly faster to write, and it generates well-formatted Verilog, so there are never any issues with synthesizing a Chisel design. Chisel error codes are easy to understand and fix. Most importantly, Chisel language features make it very easy to parameterize the design. Parameterizability was critical for this work, as fast design space exploration was very important to optimizing the hardware.

3.2 Huffman

3.2.1 High-Level Design Choices

This section of the paper describes the high-level design of each of the compressors and decompressors, the modifications to the algorithms used, and the motivations for the changes. Information about the low-level implementation details of the design is included in a later section.

Compressor

The Huffman compressor is a fully parameterized hardware compressor implementation. The parameters of the design, along with the typical values chosen for them, are shown in 3.1. The Huffman compressor supports dynamic Huffman encoding, so instead of simply translating input data with a static lookup table, it performs significant computation on the input data. This allows it to have a higher compression ratio for most workloads than would be possible with a static table.

Another benefit of the Huffman compressor design is its modularity. Because Huffman compression requires many different steps, every stage of the process has been separated into its own module. Thanks to this modularity, single components of the design can be modified or replaced without affecting the rest of the design.

The first, most significant difference between this hardware Huffman compressor and a standard implementation is the size of the Huffman tree. This design uses a truncated Huffman tree, meaning only the most frequently occurring characters are used in the generated Huffman tree. The number of characters allowed in the Huffman tree is controlled by the 'characters in the Huffman tree' parameter. The remainder of the characters are represented

Table 3.1: Configurable Huffman Parameters and their typical values.

Parameter	Typical Value
characters to compress	4096
bits in a character	8
characters in the Huffman tree	16
maximum allowed bits in a codeword	10
character counting frequency parallelism	8
compressor output encoding parallelism	8
include statistics on compressed data	false
allow for inputs smaller than the ‘characters to compress’ size	false
parallel compressor inputs	false
parallel compressor outputs	false
parallel decompressor inputs	false
parallel decompressor outputs	false

with an escape character that signals that a literal character, not character encodings, follow. This escape character adds a small amount of compression overhead, but because it is only used for the least frequently occurring characters, the overhead is not significant in most cases.

Similarly, this compression implementation is able to normalize Huffman trees to a chosen height. While normally, a Huffman tree would have a worst-case depth equal to the number of characters in the tree, this implementation is able to further reduce this worst-case depth to be equal to the ‘maximum allowed bits in a codeword’ parameter. In this way, the amount of hardware can be reduced by limiting the total number of bits of a codeword in the encoding. This normalization has a price. Because there is a limited encoding space, sometimes, there is not room to reduce the depth of a character to the depth desired. In these cases, characters less than this depth are pushed deeper, opening up additional room

in the encoding space, until all characters can be represented correctly. This also has a small impact on the compression ratio of the design, because the characters that are moved in the tree to generate more encoding space are still some of the least frequently occurring characters, so slightly lengthening them often does not make much of a difference.

Additionally, most software Huffman compressors represent the Huffman tree in canonical Huffman tree form. This is a special format that reduces the size of the tree. However, this design uses a plain text Huffman tree. In this tree, each character of the tree has its length, its encoding, and the actual character data. This design uses a plain text Huffman tree for two reasons: first, the design uses fewer characters than most designs, and second, canonical Huffman trees are more complicated to parse than plain text. The first reason means that not using a canonical form for the tree does not affect this design as much as it would affect a design that used the standard Huffman algorithm. The second reason means that the decompressor can be significantly faster, because it needs to do much less parsing and computation to determine what the Huffman tree looks like.

Finally, this Huffman compressor has two configurable options for exploiting parallelism: the character frequency counting and the final character encoding. Both of these can be configured with parameters, allowing the design to be optimized for extreme bandwidth and low-latency or to be optimized for area at the cost of latency and bandwidth.

Decompressor

Compared to the compressor, the Huffman decompressor is very simple. It uses all of the same parameters as the Huffman compressor. It contains a simple state machine that controls the reading of Huffman tree data into internal lookup tables. Once this is completed, for each parallel encoder in the compressor, there is a parallel decoder that searches for its input

data in the internal lookup tables. Each decoder is capable of a steady one byte per clock cycle, so its throughput is nearly perfectly scalable with the compression parallelism.

3.2.2 Implementation Details

This section of the paper describes the low-level implementation details of each of the compressors and decompressors and the reasoning for those implementations.

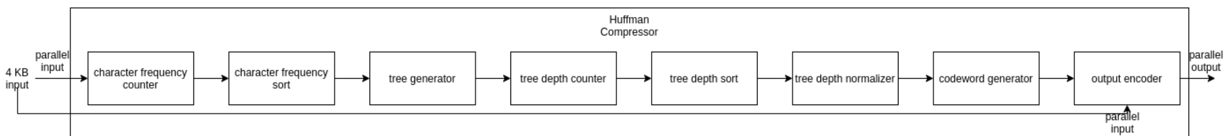

Figure 3.1: Huffman Compressor Block Diagram

Compressor

Huffman encoding is the most complicated of the hardware accelerators implemented in this work. The compressor consists of multiple modules (see Figure 3.1), each representing one stage of Huffman compressor, all connected with ready/valid interfaces. The design consists of a character frequency counter, a character frequency sort, a tree generator, a tree depth annotation traversal, a sort for the depths of characters in the tree, a character depth normalizer, a codeword generator, and a compressed data output.

The character frequency counter takes as input a number of characters determined by the 'character counting frequency parallelism' parameter. This module iterates through the entire input data, counting how many occurrences of each possible character value there are. Once this is complete, it passes an array of the total number of occurrences for each character to the character frequency sort.

The character frequency sort module sorts the frequencies it receives, but requires two passes

to perform the sort. The first pass sorts the 'characters in the Huffman tree' most frequent characters and sums the number of occurrences of characters that are excluded. Then, so all the characters can be represented in some way, the second pass uses this sum of occurrences for the escape character. The compressor sorts the escape character into the existing character data, then passes the characters and their frequencies to the tree generator. The end of this module matches up with the state of the compression example shown in Table 2.3, but with the addition of the escape character.

The tree generator builds a Huffman tree with hardware registers that can be configured as pointers or as character and frequency values. Just like with a standard Huffman tree, the tree generator iteratively builds a binary tree, adding nodes from the two least frequent members of the node list until the entire tree has a single root node. In this tree, all of the characters are leaf nodes. Each pointer from the root node can be eventually traversed to reach a character. The tree generator passes this tree of pointers and characters to the next module. The output data of this module corresponds with Figure 2.1, but the hardware adds some additional steps after this stage.

The benefit of the Huffman tree is in knowing the depths of characters in the tree, so once the tree is built, it must be traversed to find the depth of each individual character. The tree depth annotation module performs an in-order traversal, only recording information about the leaf nodes, as these are the nodes that correspond to characters. As an example of how this works, the tree shown in Figure 2.1 would result in the tree depth annotation module output shown in Table 3.2. Once the entire tree has been traversed, the tree depth annotation module passes the depth of each character recorded to the next module.

The characters in the tree must be sorted based on their depth, so the next module sorts them accordingly. Unlike the character frequency sort module, there are no other complicating features in this module, just a simple hardware sort. Once the data is sorted by depth, it is

Table 3.2: Example Tree Depth Annotation Output

Character	Depth
A	1
B	2
C	3
D	3

passed to the next module.

After sorting, it is possible that the characters in the Huffman tree could have been deeper than the chosen maximum depth, which determines the number of bits needed to encode a character. To remedy this, the tree normalization module iterates down the sorted list of characters from the lowest depths to the highest depths, then back up to the lowest depth characters once it reaches the character at the highest depth.

On the way down the list in the tree normalization module, the number of unused spaces in the encoding space is calculated. The number of available encodings in the encoding space is $2^{maximum_bits}$, where $maximum_bits$ is the maximum number of bits allowed to represent a character encoding, determined by the 'maximum allowed bits in a codeword' parameter. The number of unused encodings in the encoding space is $AvailableEncodings - \sum_{n=1}^{numberofcharacters} 2^{maximum_bits - character_bits}$, where $character_bits$ is the depth of character n in the Huffman tree. For characters at a depth deeper than 'maximum allowed bits in a codeword', their depth is set equal to 'maximum allowed bits in a codeword' during this iteration down the sorted list of characters. This method of representing the encoding space is due to the properties of the Huffman tree. Because each character in the tree is a leaf node, the less deep in the tree a leaf node is, the more encoding it is taking up in the total encoding space, because it prevents any encodings that would need to be children of its current position in the tree from being used. This method of representing the encodings is

helpful for truncating and normalizing the Huffman tree.

Once the tree normalization module has gone through each character in the list, if the number of available encodings in the encoding space is non-negative, it sends its output to the next module. Otherwise, it iterates back up the tree from the deepest characters to the shallowest characters in the tree. If a character is at the maximum possible depth, it is ignored. However, if it is not the maximum possible depth, it tries to alter the characters depth, making it deeper in the tree until it reaches the 'maximum allowed bits in a codeword' depth or the number of available encodings in the encoding space is no longer negative. As long as the number of encodings available in the encoding space is still non-negative, it continues to iterate up the list of characters and their depths.

An example of the tree normalization step with a 'maximum allowed bits in a codeword' of 2 performed on the results of the tree depth annotation in Table 3.2 is shown in Table 3.3. Note that, to make room in the encoding space for C and D to become 2-bit encodings, A had to go from a 1-bit depth to a 2-bit depth. Although this would completely nullify the compression benefit for this oversimplified example, with 8-bit characters at a 10-bit maximum character depth with an escape character, this effect is often very minor.

Table 3.3: Example Tree Normalization Output

Character	Depth
A	2
B	2
C	2
D	2

The codeword generator is the final step in the design before the data can be encoded. This module uses the sorted, normalized data that it receives to generate codewords for each character. It starts with the data at a depth of one bit and an encoding of zero. Every time

the next data item is deeper, the encoding is left-shifted by the difference in depth, and every time a data item is processed, the encoding is incremented by one to get the encoding of the next character. In this way, every character and the escape character can be given unique encodings. Once this is complete for the characters in the Huffman tree, these characters, encodings, and the encodings for characters that utilize the escape character are written to registers that are visible to the final module of the design. An example of the result of the codeword generator module with the tree depths of Table 3.2 as inputs is shown in Table 3.4. Note that the depths in Table 3.3 were not used for this example because, being all the same depth, they would all be the same value. However, normally the codeword generators input would come from the tree normalization module.

Table 3.4: Codeword Generator Encoding Example

Character	Generated Encoding
A	0
B	10
C	110
D	111

The final module of the design uses the encodings from the codeword generator module to compress the input data. This module performs the final character encoding as discussed in the Huffman compression example in Section 2.2.4. First, this module outputs the plain text form of the Huffman tree as specified in Section 3.2.1. Once this is completed, the module accepts input data compress. This module can be configured to compress multiple streams from the input data in parallel. For each byte of each input stream, the corresponding encoding and encoding length is found from the codeword generator. Once this is found, the encoding and its length are output for each stream, resulting in a fully-encoded output.

Decompressor

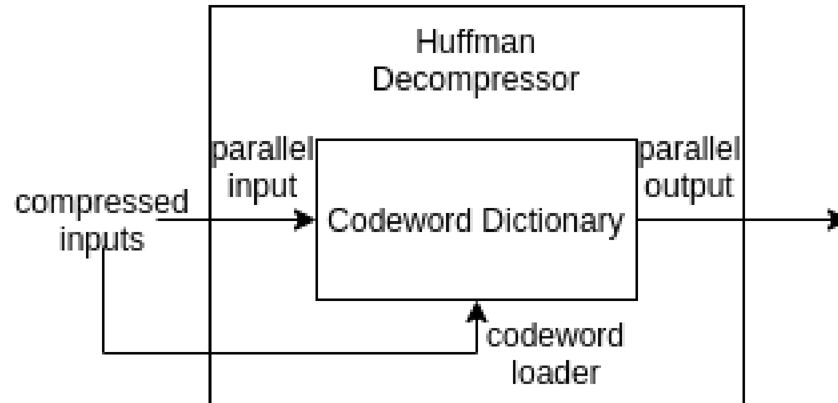

Figure 3.2: Huffman Decompressor Block Diagram

The decompressor is much simpler than the compressor. First, the decompressor reads the plaintext tree data in from the first of the parallel character inputs. In the plaintext tree, each character has a static bit width, and all possible characters in the Huffman tree are always transmitted, even if there are fewer characters than that used. The data is always transmitted as the unencoded character first, followed by the static-length character encoding, followed by the length of the encoding. Once this data is processed for each character, decompression can begin.

With the tree stored in the dictionary, the decompressor then decodes each configured character input in parallel. If a matched codeword is the escape codeword, the decompressor reads out the bits after the escape word to get the literal character being referred to. Otherwise, the decompressor reads the correct value from the dictionary. This continues until every character input has decoded the correct number of characters.

3.3 LZ77

3.3.1 High-Level Design Choices

This section of the paper describes the high-level design of each of the compressors and decompressors, the modifications to the algorithms used, and the motivations for the changes. Information about the low-level implementation details of the design is included in a later section.

Compressor

The LZ77 compressor and decompressor hardware mostly implements its algorithm unchanged. As with the other designs, it is fully parameterized, and the design parameters and typical values can be found in 3.5. The major difference between this LZ77 implementation and the normal LZ77 algorithm is that this implementation always selects the first, longest CAM match, and can only match up to 'CAM max pattern length' characters in the CAM.

Table 3.5: Configurable LZ77 Parameters and their typical values.

Parameter	Typical Value
Character bits	8
CAM characters	4096
CAM max pattern length	16
Max pattern length	266
Escape character	103
Decompressor max characters out	8

The LZ77 compressor reads each byte into a buffer of size 'CAM max pattern length'. Once this buffer is full, the pattern is searched in a CAM which returns the longest, first occurrence

and the length of the occurrence. If the length of the occurrence is less than ‘CAM max pattern length’ but equal to or greater than the number of bytes needed to encode a pattern, the CAM outputs a pattern instead of the literal bytes. If the length is the maximum, then the state of the state machine changes, and each byte after the pattern is checked against the input bytes. This continues until the bytes no longer match or the length reaches ‘Max pattern length’. Otherwise, the first character in the buffer is output as a literal character, and the pattern buffer shifts in a new value from the input.

Compressed data always follows the same format. First, the escape character value appears, then the negated most significant bit of the escape character, then the pointer to where in the CAM the data is, then the length of the data. If the length of the data exceeds the value that can be shown by one character of ‘Character bits’ width, then another character of the same width is added to the encoding after that, until the ‘Max pattern length’ is reached or this is terminated by a value that is less than the maximum possible value. However, this means that escape characters cannot be encoded properly, so two escape characters in a row encodes a single literal value of the escape character. The negated escape character bit in the encoded data format ensures that encoded data is not mistaken for two escape characters in a row.

The above information about the encoding for patterns does not make any assumptions on the size of a character or the size of a pattern. This is because the design does not either. The design can be generated for any configuration, and the encoding of patterns in the design will be adjusted to reflect the configuration.

Decompressor

The decompressor for the LZ77 hardware, like the other decompressors, is much simpler than the compressor. It reads in multiple bytes at a time and the first byte is a literal, it outputs a single byte corresponding to that literal. If the input is a pattern encoding, the decompressor transitions to the pattern decoding state. In the pattern decoding state, the decompressor outputs 'Decompressor max characters out' characters from the history every cycle. Once the decompressor has output a number of history characters equal to the length of the pattern encoding, the decompressor goes back to the initial reading state, ready to read more literal or pattern encoding data.

3.3.2 Implementation Details

This section of the paper describes the low-level implementation details of each of the compressors and decompressors and the reasoning for those implementations.

Compressor

The LZ77 compressor consists of two primary components: a pattern search module that uses a CAM to search for multi-character patterns in the byte history combinationally, and the state machine and logic that utilize the pattern search module to perform LZ77 compression.

The pattern search module uses an internal multi-character CAM. This CAM receives data and writes it to a history and receives patterns of data to search for in the history. The CAM provides an output that gives access to all of its stored data so that, when matches longer than 'CAM max pattern length' are found, the state machine of the LZ77 compressor can check their true length. The CAM also outputs multiple large arrays of wires. Each

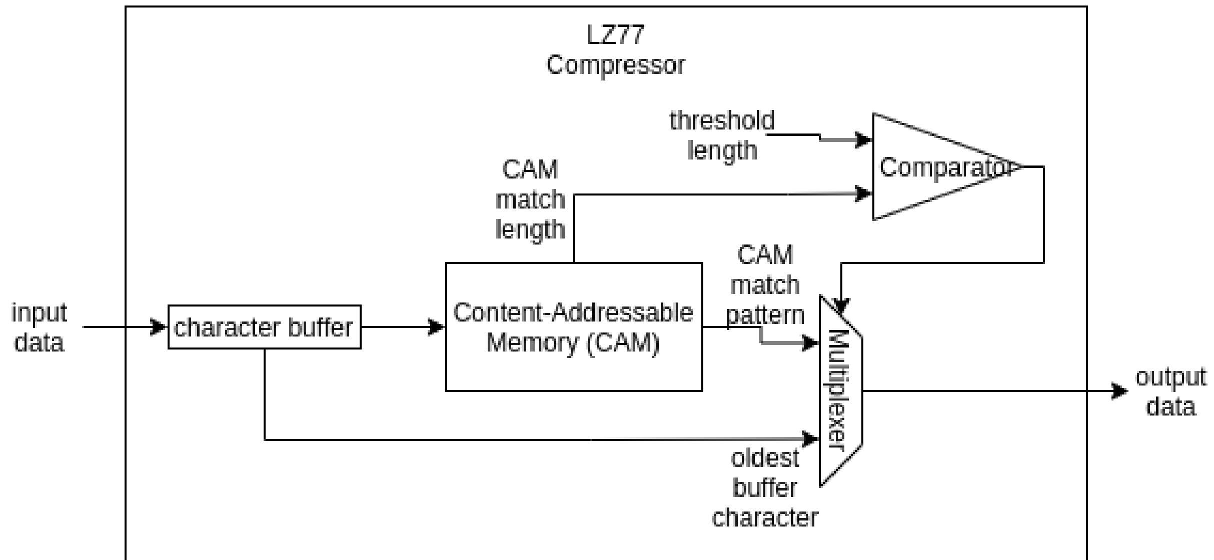

Figure 3.3: LZ77 Compressor Block Diagram

array corresponds to one of the characters in the pattern being searched. Each bit in an array shows whether or not its corresponding pattern character matches a character in the CAM's history. Table 3.6 shows an example CAM history, and Table 3.7 shows an example pattern search input. Table 3.8 demonstrates each bit array, its corresponding character in the pattern search input, and the value of each bit in each array with the CAM history included for easy reference.

The pattern search module uses the output of the internal CAM to find patterns up to length 'CAM max pattern length'. It does this by submitting a pattern to the CAM. The pattern search module performs a combinational left-shift on each array of matches from the CAM by its index, e.g. array 0 is shifted 0 bits, array 1 is shifted 1 bit, array 2 is shifted 2 bits, etc. Then, the pattern search module performs a bitwise AND on the results of the array shifts. As each additional array AND is performed, the remaining true values in the array correspond to the starting indices of patterns that match a number of characters. So, for instance, the bitwise AND of the shifted results of arrays 0, 1, 2, and 3 result in an array where each true value index corresponds to the starting index of a pattern that matches the

Table 3.6: CAM History Example

History Index	Stored Character
0	A
1	D
2	B
3	C
4	D
5	C
6	C
7	D
8	B
9	C
10	A

Table 3.7: Pattern to Search For

Index of Character in Pattern	Character to Match
0	D
1	B
2	C
3	A

first four characters of the pattern search data. Once all ANDs have been performed, the only remaining true values in the array are the indices of the starting points for patterns that matched the CAM input entirely. By reversing the resulting arrays of wires and using the Chisel3 lastIndexWhere() function, the first matching index of patterns of each length can be found [6, 7]. The results of each of these lastIndexWhere() functions can be chosen by performing an AND reduction on each resulting array of wires in the bitwise AND operation previously described, and picking the longest pattern that returns a true value from this reduction. Table 3.9 shows the result of the shifting operation for each bit array shown in

Table 3.8: Bit Array Output Example

Byte History Characters	Pattern Index 0 (D) Bit Array	Pattern Index 1 (B) Bit Array	Pattern Index 2 (C) Bit Array	Array Pattern Index 3 (A) Bit Array
A	0	0	0	1
D	1	0	0	0
B	0	1	0	0
C	0	0	1	0
D	1	0	0	0
C	0	0	1	0
C	0	0	1	0
D	1	0	0	0
B	0	1	0	0
C	0	0	1	0
A	0	0	0	1

Table 3.8. Table 3.10 shows how the hardware AND's the shifted bit arrays together to detect a pattern. Table 3.10 shows that a pattern up to three characters long was found at index 1, and a pattern up to four characters long was found at index 7. Because the longest pattern is chosen, the pattern search module would output the index 7 and the length 4 as the results of the pattern search.

The state machine that controls compression stores the input data in a buffer. The buffer must be filled, and then each clock cycle, the buffer is checked against the pattern search module. If a pattern is of a length such that the number of characters to encode the pattern is less than or equal to the number of characters in the pattern, the pattern is compressed. Otherwise, the buffer shifts out the oldest character and shifts in a new character, then adds the old character to the byte history, and the old character is output as a literal character.

Table 3.9: Shifted Bit Arrays

Bit Array 0 Shifted 0 Bits	Bit Array 1 Shifted 1 Bit	Bit Array 2 Shifted 2 Bits	Bit Array 3 Shifted 3 Bits
0	0	0	0
1	1	1	0
0	0	0	0
0	0	1	0
1	0	1	0
0	0	0	0
0	0	0	0
1	1	1	1
0	0	0	0
0	0	0	0
0	0	0	0

When a pattern is long enough to be compressed, the encoding starts off with an escape character determined by the 'escape character' parameter. If the escape character is a literal representation of that character value, the escape character occurs twice in a row. If not, the escape character is followed by the negation of the most significant bit in the escape character, then the starting index of the pattern in the byte history, then the number of characters in the pattern. This encoding is character-aligned, which causes it to work very well with Huffman encoding. However, to keep the encoding character-aligned while allowing long patterns, the encoding can have a variable length. The smallest length of the encoding occurs when the number of characters in the pattern can be added into the remaining bits needed to reach character-alignment after the escape character, negation bit, and index are all used. Because a pattern always has a minimum length, the pattern's length is always incremented by the pattern's break-even length to increase the efficiency of the encoding. If

Table 3.10: Pattern Detection with Shifted Bit Arrays

Index	Array 0	Arrays 0&1	Arrays 0&1&2	Arrays 0&1&2&3
0	0	0	0	0
1	1	1	1	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	1	1	1	1
8	0	0	0	0
9	0	0	0	0
10	0	0	0	0

the pattern is too long to be represented this way, the maximum length value is set, and the next character's value is added to the pattern length. This can continue arbitrarily until the maximum number of characters in a pattern is reached.

Decompressor

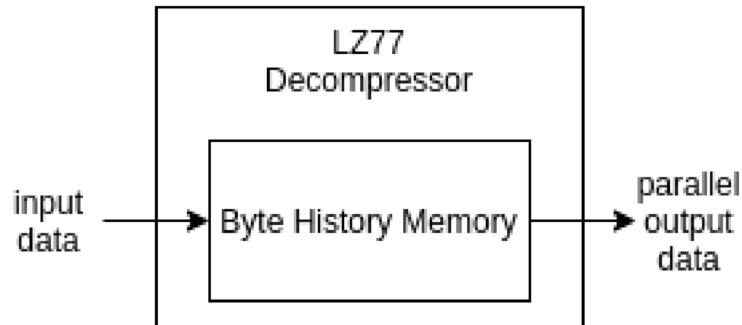

Figure 3.4: LZ77 Decompressor Block Diagram

The LZ77 decompressor takes enough characters to allow the largest possible pattern encod-

ing from the LZ77 compressor to be read in a single cycle. If the first character in the input data is not an escape character or is two escape characters in a row, it is counted as a literal that is output and added to the character history. Otherwise, a state machine is used to read bytes from the character history in parallel. The LZ77 decompressor continues to take inputs until the configured number of bytes is decompressed, then finishes.

3.4 LZW

This section discusses the LZW compressor and decompressor. First, it introduces the high-level design choices of this hardware, then it describes the details of the implementation.

3.4.1 High-Level Design Choices

This section of the paper describes the high-level design of each of the compressors and decompressors, the modifications to the algorithms used, and the motivations for the changes. Information about the low-level implementation details of the design is included in a later section.

Compressor

The LZW compressor and decompressor are very close to functioning exactly like LZW. Like the other designs, the LZW compressor and decompressor are fully parameterized. Parameters and their typical values are listed in 3.11. The LZW compressor streams in one character per cycle and adds it into a buffer to match against existing patterns in the dictionary. Once a pattern no longer matches anything in the dictionary, the last pattern match is sent to the output, and all but the last byte of the pattern are shifted out of the

pattern buffer. The pattern buffer is built up again, repeating this cycle until there are no more characters to be compressed.

Table 3.11: Configurable LZW Parameters and their typical values.

Parameter	Typical Value
character bits	8
maximum characters in a sequence	16
maximum elements in the dictionary	768
enable debugging statistics	false

The only differences compared to a software LZW implementation are limitations of hardware: there is a limit to the size of the pattern buffer and to the number of entries in the dictionary. If the dictionary is full of patterns, no new patterns will be added, but the hardware can continue to compress as normal. And if a pattern fills the entire pattern buffer, further instances of it will not continue to be added to the dictionary. Other than that, the LZW hardware compressor is normal.

Decompressor

The decompressor is even simpler than the compressor. Every time it reads an input, it outputs the resulting dictionary entry, and adds a new item to the dictionary based on the last pattern and the first byte of the current pattern. Like the compressor, it only avoids adding an element to the dictionary if the dictionary is full or the pattern is already as long as it can become.

3.4.2 Implementation Details

This section of the paper describes the low-level implementation details of each of the compressors and decompressors and the reasoning for those implementations.

Compressor

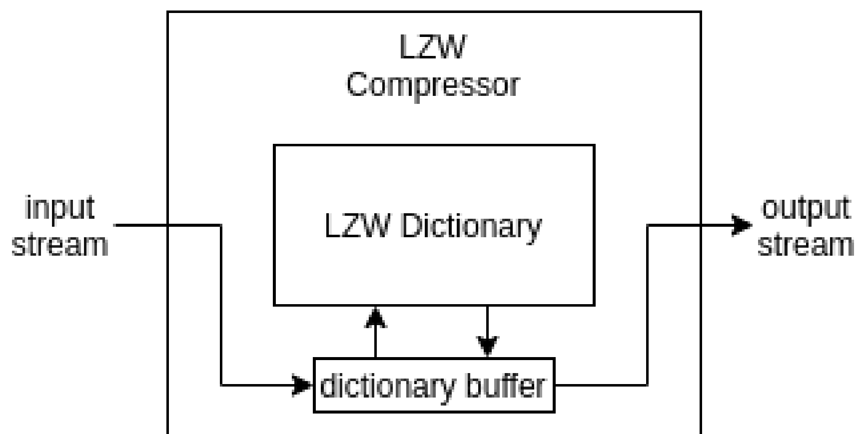

Figure 3.5: LZW Compressor Block Diagram

The LZW compressor requires fewer steps to work than the Huffman compressor, so it is implemented in a single monolithic module. The most important part of the LZW compressor design is the dictionary. Each entry in the dictionary can hold a number of values equal to 'maximum characters in a sequence' Additionally, each entry in the dictionary stores the number of character values it holds. The dictionary starts with a list of each literal character where the character's index in the dictionary is equal to its encoded value and its length is one character.

The LZW compressor reads in characters one-at-a-time to a character buffer. The size of this buffer is determined by the 'maximum characters in a sequence' parameter. Each time a new character is added to the buffer, the character is checked against the existing LZW dictionary and the length of matching entries is checked to be sure any matches are valid.

Every match is held until the next clock cycle, and once the buffer is full or there is no match in the dictionary, the last match is output and the buffer is cleared. If a match was output because the buffer was full, the buffer is completely emptied, and if not, the buffer is emptied except for the most recently added character. Every time the buffer is cleared because there was no existing match, a new entry is added to the dictionary representing the character sequence in the buffer.

When it transmits an output, the LZW compressor also sends the number of bits of the data, so the design using the compressor is able to pack the output values correctly. As the LZW dictionary grows, the number of bits required to represent an index in the dictionary grows as well.

This sequence of filling and emptying the buffer and adding characters to the dictionary is repeated until there are no more remaining characters. If there are no more remaining characters but there is still data left in the character buffer, the 'stop' signal can be asserted to force the compressor to output a match corresponding to the data still held in the character buffer.

Decompressor

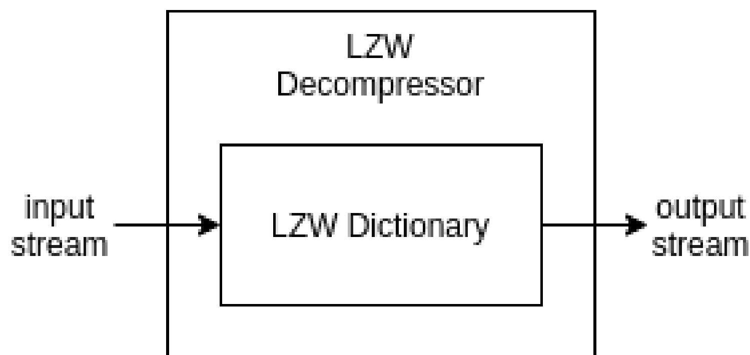

Figure 3.6: LZW Decompressor Block Diagram

The LZW decompressor works in much the same way as the compressor, minus the character buffer. The decompressor also starts with the dictionary containing an entry for each literal character. Each of these entries still have a character length of one, and the index of each entry still matches its character value. The LZW decompressor takes outputs from the compressor and uses them as indices to look up sequences in the dictionary. It outputs each of the sequences in parallel. It also outputs the number of bits each input index currently consists of so that the hardware sending the data can correctly increment whatever counters are being used to send the unaligned data, because the number of bits in an input changes as the size of the dictionary increases.

For each sequence that is not the maximum length, it and the first character of the following sequence are added to the dictionary. This makes sure that the LZW decompressor's dictionary stays in sync with the dictionary of the compressor. Unlike the LZW compressor, there is no need for a stop signal to send the final bytes of data, so once the input data is exhausted, all the output data will be transmitted as well.

3.5 Evaluation Methodology

This chapter discusses the methodology that was used to generate data about the designs in this work. It also contains details about the ways existing designs can be compared to this work.

3.5.1 Hardware Synthesis

All designs were implemented with the ASAP7 7nm Predictive PDK with the Synopsys Design Compiler. First, designs were synthesized with a target clock period of 10 picoseconds.

The synthesis tools will fail to meet this unreasonable expectation, but the negative slack from this synthesis run can be used to find the fastest possible clock speed and the area usage of the design at this speed. A second run at the correct maximum clock speed can then be used to find power information, if desired.

3.5.2 IBM Area Estimation

Unfortunately, IBM does not provide exact area numbers for the compression and decompression acceleration engine [3]. However, die photographs enable area estimation for a POWER9 CPU [56]. Using this and the claim that the acceleration engine took up less than 0.5% of the entire chip's area, the area of the acceleration engine can be approximated [3].

3.5.3 Compression Ratio Comparison

To compare memory compression ratios of the hardware designs with existing software implementations, memory dumps were taken randomly during benchmarks from SPEC, Sparkbench, Parsec, and other benchmarks. These memory dumps were then parsed into 4 KB pages of memory. These pages were randomly sampled and compressed and decompressed with Gzip (which uses Deflate), and the hardware designs from this work. Although no hardware Deflate compressor or decompressor exists as part of this work, the pages were also compressed and decompressed with LZ77 and Huffman sequentially to demonstrate the difference between software and hardware implementations of the same algorithm.

The goal of this compression ratio comparison was to determine how much effective RAM could be achieved with memory compression using each design. Because of this, all pages that compressed to larger than 4 KB were treated as being exactly 4 KB for the average compression ratio calculation, as an incompressible page would remain in an uncompressed

form during normal OS memory compression.

3.5.4 Latency

To calculate software compression latency, Gzip was timed compressing several thousand 4 KB memory dump pages in a row, then the time was divided by the number of pages. This was done because a single compression often occurs too quickly to accurately time, so multiple sequential compression operations were needed.

For hardware latency calculations, the latency can be found by multiplying the fastest possible clock period of each design with the number of cycles required to complete compression or decompression with the design.

Chapter 4

Results and Analyses

This chapter delivers the results of the investigations detailed in Section 3.5 and analyzes them.

4.1 ASIC Area and Frequency Comparison

Figure 4.1 shows the highest clock speeds achievable by each design configuration. As can be seen, most of the designs are capable of achieving at least 2 GHz, except for the LZ77 compressor and the LZW decompressor. The reason for the LZ77 compressor's lower than expected clock speed is not currently known. Initial investigations into the clock speed by dividing the design up and synthesizing it as separate pieces suggest that the reduction in clock speed is not due to the CAM or pattern search module, so must be due to the state machine. However, the state machine does not contain much additional logic, so it is currently unclear what components cause the slowdown.

As for the LZW decompressor, although its hardware is simpler than the LZW compressor, the compressor takes two cycles per byte compressed. The LZW decompressor only uses one cycle per input pattern. It seems that the synthesis tools are able to perform register retiming on the LZW compressor to make it faster, but that this retiming is not possible on the decompressor because fewer cycles are available for each additional pattern to the dictionary. This also must be investigated further to be determined certainly.

IBM does not discuss the specifics of the clock speeds attainable by the hardware [3]. However, IBM mentions that the target clock speeds are 2.0-2.5 GHz. The IBM POWER9 and z15 compressors are implemented on a 14 nm manufacturing node. Although there is no way to make a direct comparison between the GlobalFoundries node used by IBM and the ASAP7 PDK, a TSMC press release review by Schor provides a way to approximate [47]. If it is assumed that the difference between the GlobalFoundries 14 nm node and ASAP7 is the same as the difference between TSMC's 16 nm node and TSMC's 7 nm node described by Schor, then IBM would be able to achieve speeds 35-40% faster than the POWER9 compression engine, meaning it would be in the range of 2.7-3.5 GHz [47]. This would mean that its clock speeds are faster than most of the designs of this work. However, it is important to note that IBM's design is heavily optimized to achieve these clock speeds, and the most important aspect of the design is not clock frequency, but total memory latency [3].

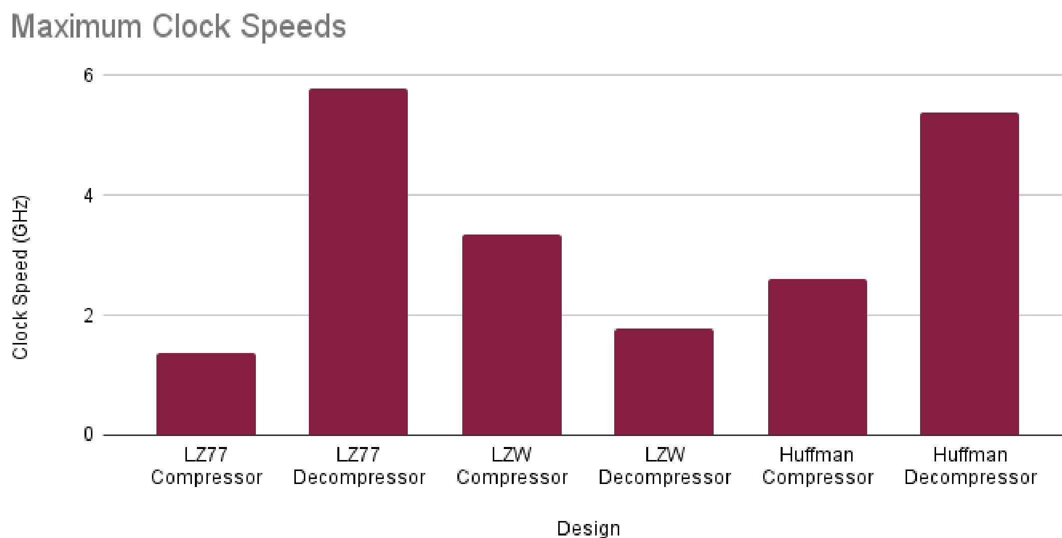

Figure 4.1: Open-Source Compressor and Decompressor Maximum Clock Speed Comparison

Although not all of the designs can achieve similar clock speeds to the IBM compression accelerator, they are certainly much more configurable due to their parameterization. As

can be seen from Figure 4.2, all of the areas of the designs in this work can be significantly changed based on the configuration. The compressor and decompressor with the largest capability for change, LZW, can change in size more than an order of magnitude, depending on what values the parameters are set to.

The designs in this work also achieve much lower areas than the IBM work. Figure 4.3 shows the area results of the design configurations chosen to be compared to the IBM implementation. Although IBM does not publish the exact area of the POWER9 compression engine, IBM mentions that the compression engine takes less than 0.5% of the total chip area [3]. Additionally, estimates of the die area from photographs can be used to indicate the approximate area of the IBM design [56]. For a direct comparison to the IBM area estimate, Figure 4.3 shows the area of the designs from this work and IBM, normalized to the smallest design in this work. As is evident, the smallest design in this work is 866x smaller than IBM's accelerator, and even the largest design in this work is still 18.6x smaller than IBM's design.

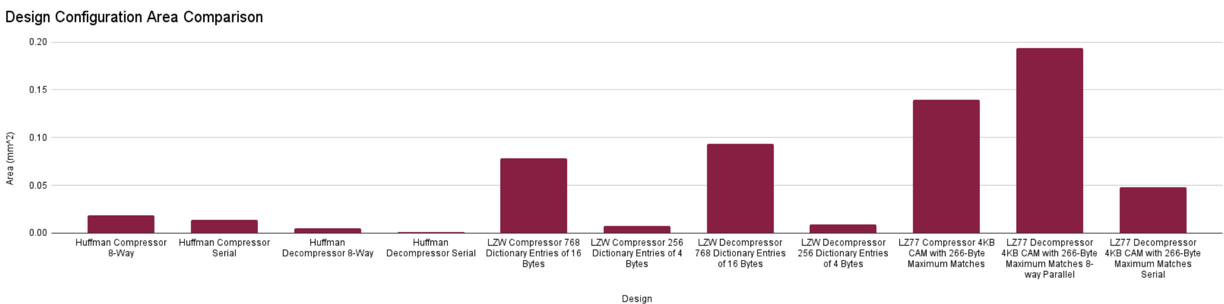

Figure 4.2: Parameterized Design Area Configurability Demonstration

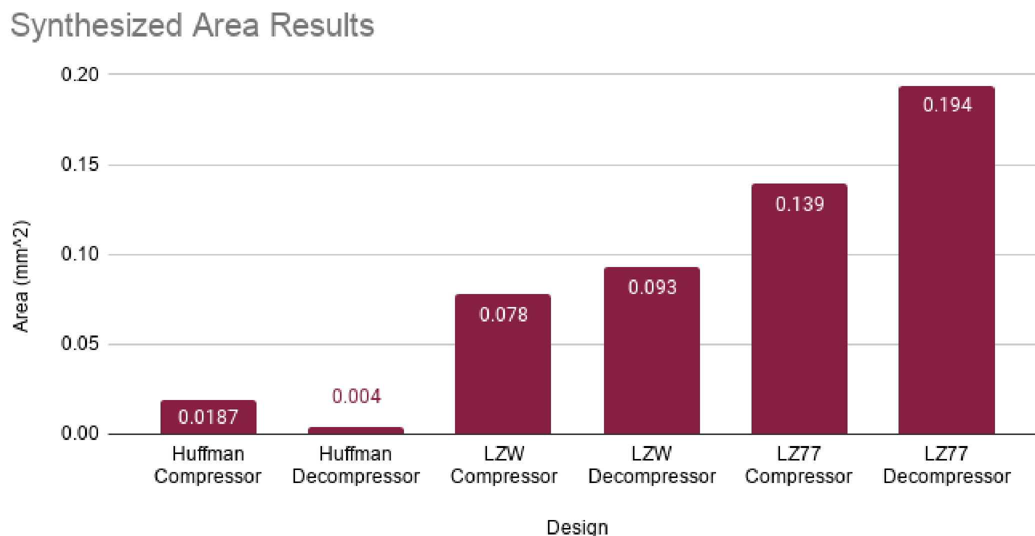

Figure 4.3: Compressor and Decompressor 7 nm Design Area Comparison

4.2 Latency Comparison

To compare latencies to the IBM design, first it is important to calculate the latencies of the IBM design. For the purposes of comparing against this work's designs, it is most fair to compare IBM's latency to compress or decompress a 4 KB page of memory. The IBM paper states, 'We model the accelerator execution time T with $T = T_0 + Data_Size/PeakBandwidth$ where constant T_0 is the setup time which is the software and hardware initialization delays before the accelerator starts processing data.' [3]. To get the overall hardware latency of the IBM design, the hardware latency portion of T_0 needs to be known, and the result of $Data_Size/PeakBandwidth$ needs to be known for compression and decompression. The sum of these will be the hardware latency for IBM's implementations.

The hardware latency portion of T_0 can be found without difficulty. Later in the paper, Abali et. al state, 'Therefore, with multithreading software delays nearly disappear in the total delay calculations and the 8-thread latencies in Table II reveal the actual NXU hardware

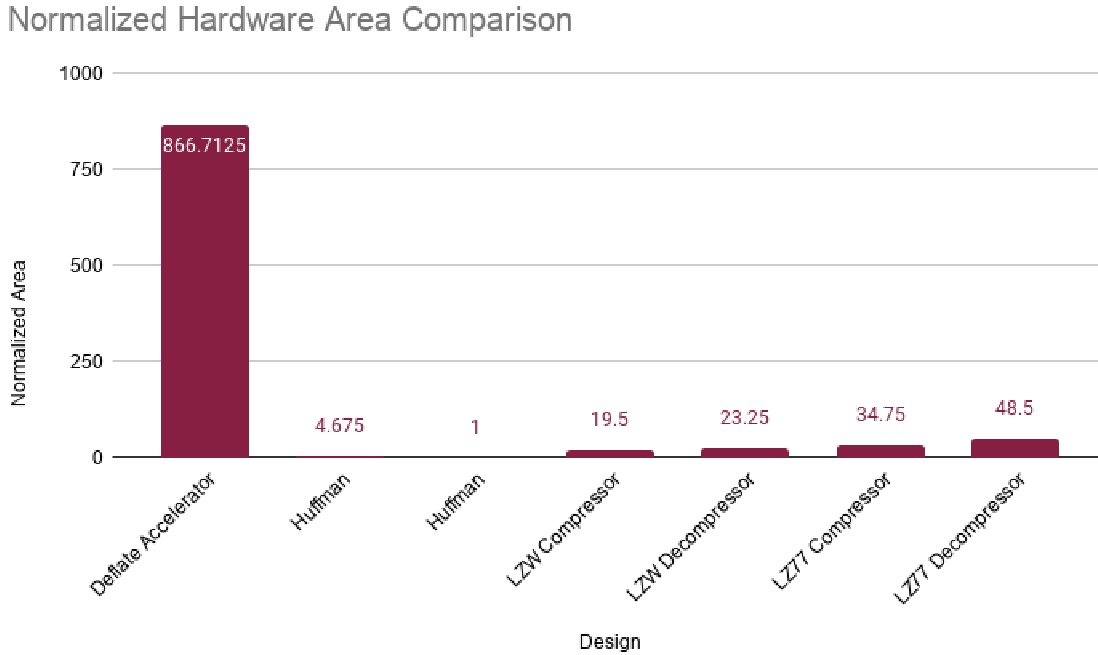

Figure 4.4: Normalized Hardware Area Comparison

latencies in the 0.55-0.73μ range' [3]. The table the authors referred to can then be used to find the hardware latency for compression and decompression for both the z15 and POWER9 accelerator engines.

The latency once the hardware is initialized requires some additional math, but can be found from IBM's description as well. The IBM paper says that, 'When the data size is $Half_Peak_Data_Size = T_0 \times PeakBandwidth$ we get $T = 2T_0$, which means that half the time is spent in the setup and the other half in the accelerator doing actual processing.' [3]. From this, we can deduce that $PeakBandwidth = Half_Peak_Data_Size \div T_0$. The peak data size of 4 KB is already known, so the final equation for this portion of the latency is $4KB/(Half_Peak_Data_Size \div T_0)$. These remaining variables can both be found in the IBM paper's Table II, so it is possible to calculate the hardware latency of each accelerator. The latency of each accelerator is shown in Table 4.1.

Table 4.1: Hardware Latencies of IBM Compression and Decompression Accelerators

Hardware Design	Total Hardware Latency (μs)
POWER9 Compressor	1.1
z15 Compressor	0.88
POWER9 Decompressor	1.21
z15 Decompressor	1.205

Because I have all of the necessary information for my designs, it is much simpler to calculate the total latencies. The minimum and maximum compression and decompression latencies are simply the most and fewest possible cycles to compress or decompress a page multiplied by the clock period of a single cycle.

Figure 4.5 and Figure 4.6 show the latency comparison between the IBM accelerator and this work. As these graphs show, the minimum decompression latencies from the designs in this work have significantly lower decompression latencies than the IBM designs. However, although the designs in this work demonstrate significantly improved decompression latencies, it is important to note that the compression latencies are often higher with the designs from this work, and the Deflate hardware design is hypothetical.

The maximum decompression latencies are significantly faster for my design than the IBM design, but these maximum decompression latencies can only be reached when the data is completely incompressible. If data is incompressible, then there is no need for it to be stored in memory in a compressed form, so no need for it to be decompressed. In all cases with reasonable compression ratios for the compressed memory, the latency of my hardware designs is significantly better than the maximum decompression latency, moving the performance of this design closer to an order of magnitude lower than IBM's.

The compression latencies are often higher with my designs. However, this is not as much of a concern with memory compression as the decompression latencies. This is because,

while the decompression is on the critical path to completing computations, the compression latency is not. Once a compression operation has been submitted, the CPU could move on while the memory write was buffered and compressed in the background.

The final note on these figures is that the Deflate hardware design shown is hypothetical. The latency numbers for this design make the conservative assumption that no pipelining can take place between LZ77 and Huffman compression or decompression, and each stage completes separately before the next one starts. Even making this assumption, my hardware designs could achieve significantly lower latencies than IBM if they were combined to form a Deflate compressor and decompressor.

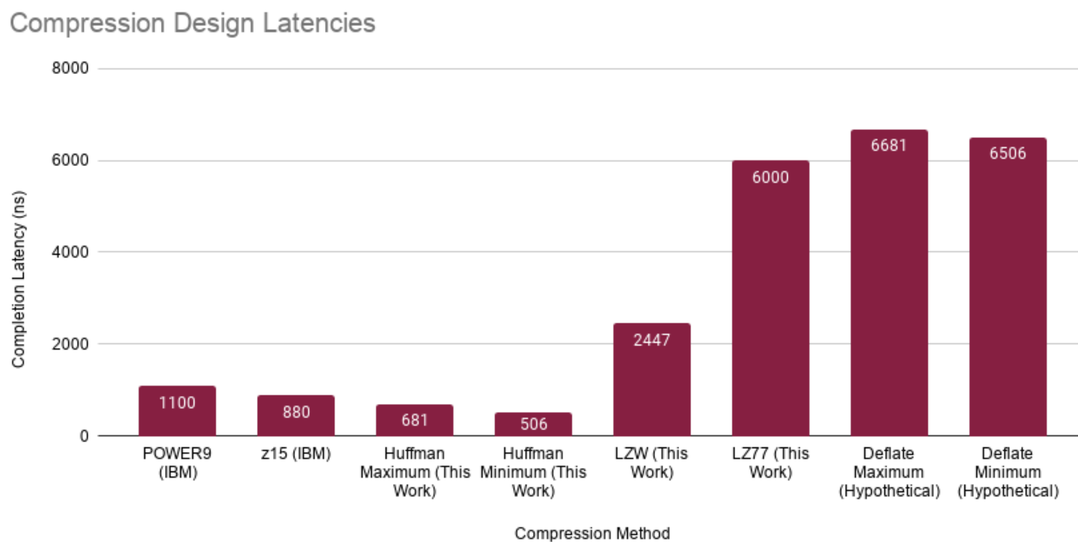

Figure 4.5: Compression Design Latencies

4.3 Compression Ratio Comparison

Although decompression latency is very important, it must not be at the cost of significant data compressibility. If the memory compressibility is significantly reduced, then there is no

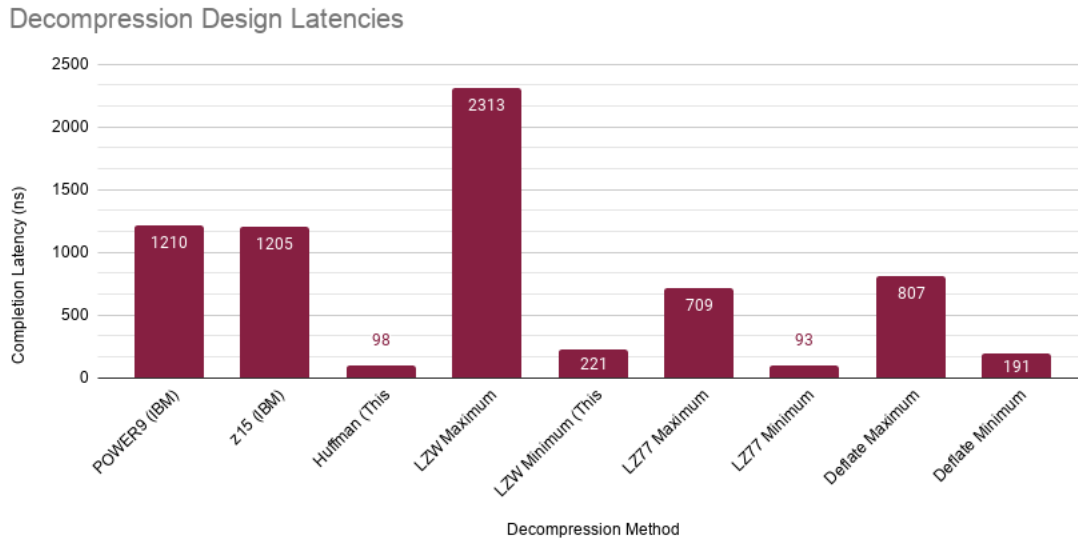

Figure 4.6: Decompression Design Latencies

point to compressing the memory in the first place. Figure 4.7 shows the compression ratios achieved by each of the hardware designs as configured above. Additionally, LZ77+Huffman on the graph shows the performance of a hypothetical design combining LZ77 compression and Huffman decompression to create a Deflate compressor. This figure shows that on average, the LZ77+Huffman hardware is capable of achieving compression ratios 87% as high as the software Deflate implementation. While this reduction is non-negligible, it is not enough to completely negate the value of memory compression, and the latency reduction of several orders of magnitude is likely worth the slightly reduced compressibility. This is especially true in the context of operating system memory compression. Because the operating system often does not compress memory data to a byte granularity, it is quite possible that many workloads would see similar or identical effective memory from the hardware implementation and the software implementation.

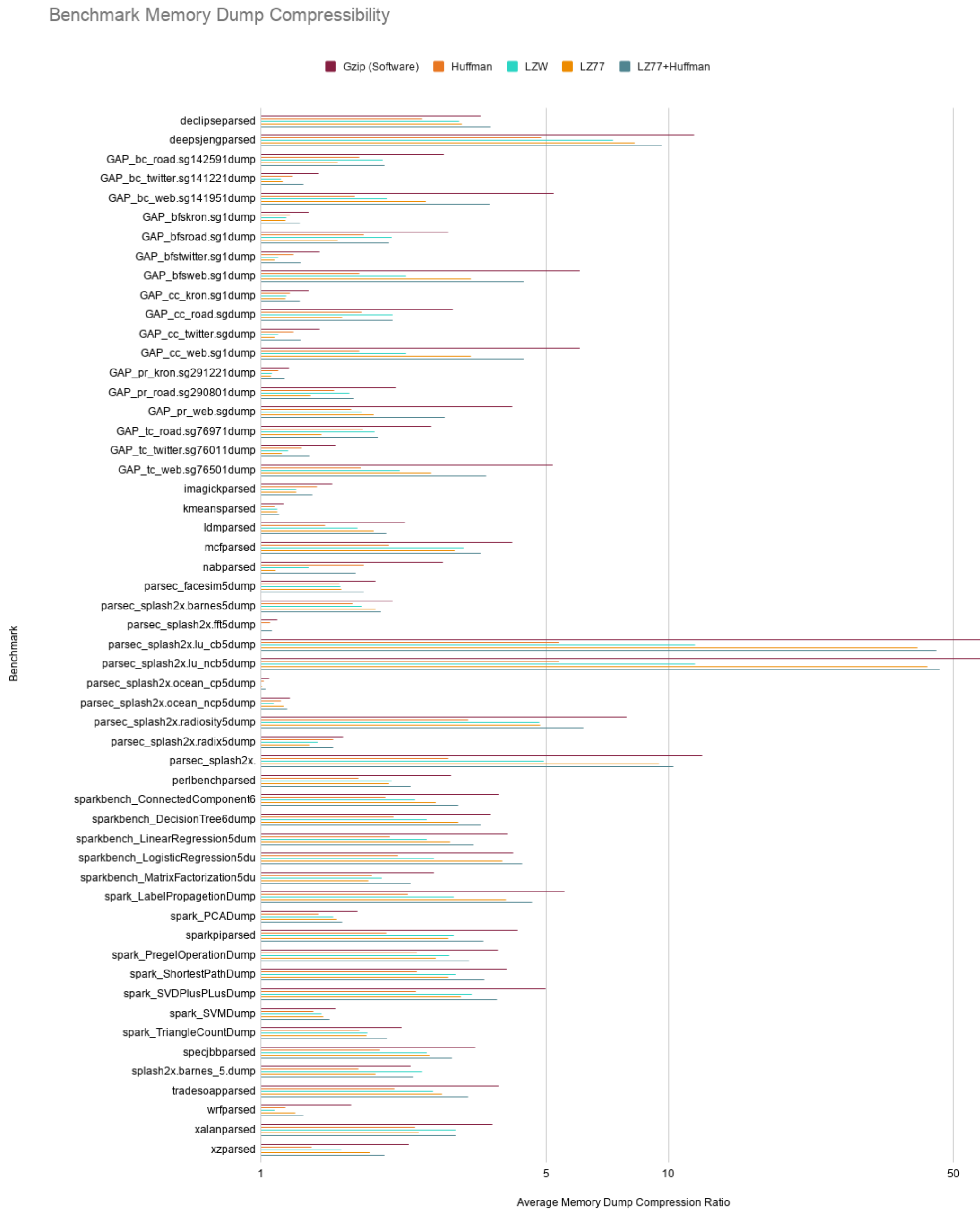

Figure 4.7: Benchmark Memory Dump Compressibility

Chapter 5

Limitations and Future Work

Although this work compares well to the current state of the art, there are a number of potential improvements that could strengthen it significantly. In this chapter, these limitations will be discussed in order of potential benefit.

Currently, the LZ77 decompressor design is capable of decompressing multiple characters per cycle when given an encoded pattern. However, when the LZ77 decompressor receives uncompressed literal characters, it currently processes them one-at-a-time, sequentially. This makes the LZ77 decompression latency very closely tied to the compressibility of the data. To fix this, the hardware could support multiple bytes of literal input per cycle. A naive implementation would only do this search if there were no escape characters at all in the characters read in parallel. However, a more complex implementation could support variable parallel read speeds, only reading characters that come before an escape character in the input data. This would significantly decouple the decompression latency from the compressibility of the data, improving the speed of nearly all decompression operations by a significant margin.

The designs would also benefit from being formally verified. Although the designs have each been tested with over 400,000 4 KB memory dump files without any bugs found, it is possible that the testing did not trigger edge cases that could cause the hardware designs to compress or decompress incorrectly. Formal verification is a method of mathematically proving that a design has certain characteristics, given a set of initial assumptions. If the designs were

formally verified to never incorrectly compress or decompress data, this would make them significantly better candidates for incorporating into an ASIC. Additionally, an easy-to-use formal verification flow would allow for faster design changes, because bugs could be caught as soon as they were written and tested.

While the designs are all able to achieve low latencies, some of the designs are not able to run above 2 GHz. Because one of the ideal use cases of these designs are in a memory controller, it would be good for the designs to be able to run at the clock speeds common in server CPUs, somewhere in the 2-3 GHz range. This would allow integration into memory controllers without the need for multiple clock domains. Additionally, if this could be achieved without increasing the number of cycles needed to perform the compression or decompression operations, increasing the clock speeds of the designs would also further reduce latency.

Another valuable future work would be to add the design to a larger project. Currently, the designs have only been tested in an isolated context. Adding the designs to something like a memory controller or FPGA to accelerate compression and decompression would provide valuable feedback about how easily the design can be interfaced with. This is an important future work, as it is a primary motivation for open-sourcing this work.

While this work is primarily targeted towards memory compression with its low latencies, all of the designs in this work have very low area overhead. Additionally, they implement algorithms that are frequently used for data outside of memory compression. While it has not yet been investigated, it would be valuable to investigate the compression ratios and performance that could be expected from these designs in a file or network compression context. Because the algorithms used in this work are common for file compression, it might be possible for this work to be used for file and network compression as well as memory compression.

The only component of this work that is not fully open-source is the Synopsys Design Compiler. Synopsys Design Compiler is used to synthesize the area, power, and frequency numbers of this work. Unfortunately, a Synopsys license is expensive and the software is proprietary. To match the open-source nature of this project and the other tools used by it, it would be valuable to use a fully open-source toolchain for the synthesis. The OpenROAD project seems to have made significant headway into an open-source, fully-automated RTL-to-GDS toolflow [1]. If support for OpenROAD were added to this work, it would provide the added benefit of seeing the area, frequency, and power of the design on the variety of OpenROAD-supported manufacturing nodes. If this support were added, every possible aspect of the design and tooling would be open-source and free for anyone to try.

Finally, software libraries could be made compatible with the compressors and decompressors shown in this work. Many previous works utilize only a compression accelerator or only a decompression accelerator [4, 29, 31, 36, 43, 44, 48]. However, modifications have been made to all of the algorithms used in this work. This makes combining the proposed techniques with software implementations impossible at present. To further improve the flexibility of this work, it would be valuable to implement performant, configurable software implementations of the compressors and decompressors.

Chapter 6

Conclusions

As workloads continue to eat up more RAM, memory compression will continue to become an important cost-saving tool. Many of the benchmarks shown demonstrate compression ratios significantly higher than 2x. Existing compression and decompression accelerators are able to significantly reduce the latency of compression and decompression. However, existing accelerators add several times the latency of an uncompressed memory access.

This work improves upon the existing state-of-the-art designs by reducing decompression latency by over an order of magnitude at best and by 33% even in the worst case for the benchmarks studied. With all of the designs in this work combined, this work still utilizes nearly 8x less area than IBM's state-of-the-art implementation. This smaller area will lead to lower costs and higher yields for designs based on this work.

This work is parameterized, making it suitable for a variety of use cases. This work has the potential to be used anywhere from small microcontrollers to large warehouse-scale data centers. This work is much more configurable than any existing designs. This configurability enables faster iteration, because design space exploration can be done in parallel.

Most importantly, the design is available to download for free under the MIT open-source license. This allows for any individual, corporation, or other entity to use, modify, or distribute this work or derivative works free of charge. This will hopefully save future researchers hundreds of hours of development time, speed further research into memory compression,

and result in stability, speed, and area improvements as bugs are found and fixed and improvements are implemented.

This work is the first hardware compression and decompression framework to be open-sourced for the community to use, and the performance and configurability make the designs in this work very easy to use in existing designs and modify. Through open-sourcing the proposed artifacts, the intent is to encourage others to pursue new designs and optimizations related to memory compression.

Bibliography

[1] D. Blaauw 175. T. Ajayi. Openroad: Toward a self-driving, open-source digital layout implementation tool chain. *Proceedings of Government Microcircuit Applications and Critical Technology Conference*, 2019. URL https://par.nsf.gov/biblio/10171024.

[2] B. Abali, H. Franke, Xiaowei Shen, D.E. Poff, and T.B. Smith. Performance of hardware compressed main memory. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 73–81, 2001. doi: 10.1109/HPCA.2001.903253.

[3] B. Abali, B. Blaner, J. Reilly, M. Klein, A. Mishra, C. B. Agricola, B. Sendir, A. Buyuktosunoglu, C. Jacobi, W. J. Starke, H. Myneni, and C. Wang. Data compression accelerator on ibm power9 and z15 processors : Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2020. doi: 10.1109/ISCA45697.2020.00012.

[4] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330077. doi: 10.1145/2664666.2664670. URL https://doi.org/10.1145/2664666.2664670.

[5] AHA. Data compression, 2015. URL http://www.aha.com/data-compression/AHA371.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a

scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012. doi: 10.1145/2228360.2228584.

[7] Jonathon Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel, Apr 2021. URL https://www.chisel-lang.org/api/3.4.3/chisel3/Vec.html.

[8] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. Openpiton: An open source manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 217–232, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872414. URL http://doi.acm.org/10.1145/2872362.2872414.

[9] L. Benini, D. Bruni, A. Macii, and E. Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 449–453, 2002. doi: 10.1109/DATE.2002.998312.

[10] G. Bruns, P. Merkelbach, C. Schlockermann, M. Salinga, M. Wuttig, T. D. Happ, J. B. Philipp, and M. Kund. Nanosecond switching in GeTe phase change memory cells. *Applied Physics Letters*, 95(4):043108, July 2009. doi: 10.1063/1.3191670. URL https://doi.org/10.1063/1.3191670.

[11] Chris Celio. *A Highly Productive Implementation of an Out-of-Order Processor Generator*. PhD thesis, University of California Berkeley, 2018.

[12] Raul Cervera, Toni Cortes, and Yolanda Becerra. Improving application performance through swap compression. In *1999 USENIX Annual Technical Conference*

(USENIX ATC 99), Monterey, CA, June 1999. USENIX Association. URL https://www.usenix.org/conference/1999-usenix-annual-technical-conference/improving-application-performance-through-swap.

[13] E. Choukse, M. Erez, and A. R. Alameldeen. Compresso: Pragmatic main memory compression. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 546–558, 2018. doi: 10.1109/MICRO.2018.00051.

[14] Yann Collet, Apr 2011. URL https://github.com/lz4/lz4.

[15] Yann Collet, Jan 2015. URL https://facebook.github.io/zstd/.

[16] Toni Cortes, Yolanda Becerra, and Raúl Cervera. Swap compression: resurrecting old ideas. *Softw., Pract. Exper.*, 30:567–587, 04 2000. doi: 10.1002/(SICI)1097-024X(20000425)30:53.3.CO;2-Q.

[17] L. Peter Deutsch. Deflate compressed data format specification version 1.3, May 1996. URL https://tools.ietf.org/html/rfc1951.

[18] Bao Ergude, Li Weisheng, Fan Dongrui, and Ma Xiaoyu. A study and implementation of the huffman algorithm based on condensed huffman table. In *2008 International Conference on Computer Science and Software Engineering*, volume 6, pages 42–45, 2008. doi: 10.1109/CSSE.2008.1432.

[19] Brian Fitzgerald. The transformation of open source software. *MIS Quarterly*, 30(3): 587–598, 2006. ISSN 02767783. URL http://www.jstor.org/stable/25148740.

[20] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '15, page 52–59, USA, 2015. IEEE Computer Society. ISBN

9781479999699. doi: 10.1109/FCCM.2015.46. URL https://doi.org/10.1109/FCCM.2015.46.

[21] Suresh Gopalakrishnan. 3rd gen amd epyc™ cpus to expand inside ibm cloud, Mar 2021. URL https://www.ibm.com/cloud/blog/announcements/amd-7763-on-ibm-cloud-bare-metal-servers#:~:text=3rd%20Gen%20AMD%20EPYC%E2%84%A2%207763%20on%20IBM%20Cloud%20platform,4096%20GB%20RAM%20per%20CPU.

[22] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[23] Yiming Huai. Spin-transfer torque mram (stt-mram): Challenges and prospects. *AAPPS Bulletin*, 18, 01 2008.

[24] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. doi: 10.1109/JRPROC.1952.273898.

[25] Dawoon Jung, Jin soo Kim, Seon yeong Park, Jeong uk Kang, and Joonwon Lee. Fass: A flash-aware swap system. In *Proc. of International Workshop on Software Support for Portable Storage (IWSSPS*, 2005.

[26] Florent Kermarrec, Sébastien Bourdeauducq, Jean-Christophe Le Lann, and Hannah Badier. Litex: an open-source soc builder and library based on migen python dsl, 2020.

[27] J. Y. Kim, S. Hauck, and D. Burger. A scalable multi-engine xpress9 compressor with asynchronous data transfer. In *2014 IEEE 22nd Annual International Symposium on*

Field-Programmable Custom Computing Machines, pages 161–164, 2014. doi: 10.1109/FCCM.2014.49.

[28] M Kjelsø, M Gooch, and S Jones. Performance evaluation of computer architectures with main memory data compression. *Journal of Systems Architecture*, 45(8):571–590, 1999. ISSN 1383-7621. doi: https://doi.org/10.1016/S1383-7621(98)00006-X. URL https://www.sciencedirect.com/science/article/pii/S138376219800006X.

[29] M. Ledwon, B. F. Cockburn, and J. Han. Design and evaluation of an fpga-based hardware accelerator for deflate data decompression. In *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*, pages 1–6, 2019. doi: 10.1109/CCECE.2019.8861851.

[30] John Lee. Amd epyc 7763 review top for this generation, Mar 2021. URL https://www.servethehome.com/amd-epyc-7763-review-top-for-this-generation/.

[31] Sang Muk Lee, Ji Hoon Jang, Jung Hwan Oh, Ji Kwang Kim, and Seung Eun Lee. Design of hardware accelerator for lempel-ziv 4 (LZ4) compression. *IEICE Electronics Express*, 14(11):20170399–20170399, 2017. doi: 10.1587/elex.14.20170399. URL https://doi.org/10.1587/elex.14.20170399.

[32] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *2005 IEEE International Conference on Cluster Computing*, pages 1–10, 2005. doi: 10.1109/CLUSTR.2005.347050.

[33] Mingwei Lin and Shuyu Chen. Flash-aware linux swap system for portable consumer electronics. *IEEE Transactions on Consumer Electronics*, 58(2):419–427, 2012. doi: 10.1109/TCE.2012.6227442.

[34] Linux Memory Compression. Linux 3.14, Dec 2017. URL https://kernelnewbies.org/Linux_3.14.

[35] Ke Liu, Xuechen Zhang, Kei Davis, and Song Jiang. Synergistic coupling of ssd and hard disk for qos-aware virtual memory. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 24–33, 2013. doi: 10.1109/ISPASS.2013.6557143.

[36] H. Luo, Y. Cai, Q. Luo, and R. Mao. Fpga-based parallel multi-core gzip compressor in hdfs. In *2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 31–35, 2019. doi: 10.1109/PDCAT46702.2019.00017.

[37] Jose Luis Nunez-Yanez. Xmatchpro lossless data compressor, Mar 2006. URL https://opencores.org/projects/xmatchpro.

[38] Markus Oberhumer. Lzo, May 1996. URL http://www.oberhumer.com/opensource/lzo/.

[39] Gennady Pekhimenko, Todd C. Mowry, and Onur Mutlu. Linearly compressed pages: A main memory compression framework with low complexity and low latency. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 489–489, 2012.

[40] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. BlackParrot: An agile open-source RISC-v multicore for accelerator SoCs. *IEEE Micro*, 40(4):93–102, July 2020. doi: 10.1109/mm.2020.2996145. URL https://doi.org/10.1109/mm.2020.2996145.

[41] W. Qiao, J. Du, Z. Fang, M. Lo, M. F. Chang, and J. Cong. High-throughput lossless compression on tightly coupled cpu-fpga platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44, 2018. doi: 10.1109/FCCM.2018.00015.

[42] Simone Raoux, Feng Xiong, Matthias Wuttig, and Eric Pop. Phase change materials and phase change memory. *MRS Bulletin*, 39(8):703–710, August 2014. doi: 10.1557/mrs.2014.139. URL https://doi.org/10.1557/mrs.2014.139.

[43] Gonçalo César Mendes Ribeiro. *Data Compression Algorithms in FPGAs*. PhD thesis, Instituto Superior Técnico, May 2017. URL https://fenix.tecnico.ulisboa.pt/downloadFile/1970719973966488/tese.pdf.

[44] S. Rigler, W. Bishop, and A. Kennings. Fpga-based lossless data compression using huffman and lz77 algorithms. In *2007 Canadian Conference on Electrical and Computer Engineering*, pages 1235–1238, 2007. doi: 10.1109/CCECE.2007.315.

[45] S. Roy, R. Kumar, and M. Prvulovic. Improving system performance with compressed memory. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 7 pp.–, 2001. doi: 10.1109/IPDPS.2001.925011.

[46] Samsung Memory. M386a8k40bmb-cpb - samsung 1x 64gb ddr4-2133 lrdimm pc4-17000p-l quad rank x4 module, Dec 2019. URL https://memory.net/product/m386a8k40bmb-cpb-samsung-1x-64gb-ddr4-2133-lrdimm-pc4-17000p-l-quad-rank-x4-module/

[47] David Schor. Tsmc talks 7nm, 5nm, yield, and next-gen 5g and hpc packaging, Jul 2019. URL https://fuse.wikichip.org/news/2567/tsmc-talks-7nm-5nm-yield-and-next-gen-5g-and-hpc-packaging/.

[48] Lukas Schrittwieser. Lzrw1 compressor core, Apr 2013. URL https://opencores.org/projects/lzrw1-compressor-core.

[49] Greg Shultz. How to monitor memory compression in windows 10, Feb 2018. URL https://www.techrepublic.com/article/how-to-monitor-memory-compression-in-windows-10/.

[50] John Siracusa. Os x 10.9 mavericks: The ars technica review, Oct 2013. URL https://arstechnica.com/gadgets/2013/10/os-x-10-9/17/.

[51] Willy Tarreau and Dave Rodgman. Linux, Jul 2014. URL https://www.kernel.org/doc/html/v5.10/staging/lzo.html.

[52] S. Tehrani, J.M. Slaughter, E. Chen, M. Durlam, J. Shi, and M. DeHerren. Progress and outlook for mram technology. *IEEE Transactions on Magnetics*, 35(5):2814–2819, 1999. doi: 10.1109/20.800991.

[53] E.Y. Tsymbal and D.G. Pettifor. Perspectives of giant magnetoresistance. In Henry Ehrenreich and Frans Spaepen, editors, *Solid State Physics*, volume 56 of *Solid State Physics*, pages 113–237. Academic Press, 2001. doi: https://doi.org/10.1016/S0081-1947(01)80019-9. URL https://www.sciencedirect.com/science/article/pii/S0081194701800199.

[54] Tom Vijlbrief. Hdl-deflate, Dec 2018. URL https://www.librecores.org/tomtor/hdl-deflate.

[55] Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984. doi: 10.1109/MC.1984.1659158.

[56] Wikichip IBM Area. Power9 - microarchitectures - ibm, 2021. URL https://en.wikichip.org/wiki/ibm/microarchitectures/power9.

[57] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010. doi: 10.1109/JPROC.2010.2070050.

[58] H.-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T. Chen, and Ming-Jinn Tsai. Metal–oxide rram. *Proceedings of the IEEE*, 100(6):1951–1970, 2012. doi: 10.1109/JPROC.2012.2190369.

[59] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015. doi: 10.1109/TKDE.2015.2427795.

[60] Furqan Zhoor, Tun Zainal Azni Zulkifli, and Farooq Ahmad Khanday. Resistive random access memory (rram): an overview of materials, switching mechanism, performance, multilevel cell (mlc) storage, modeling, and applications. *Nanoscale Research Letters*, 15(90), 2020. doi: 10.1186/s11671-020-03299-9.

[61] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi: 10.1109/TIT.1977.1055714.

[62] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978. doi: 10.1109/TIT.1978.1055934.