

A Hardware Accelerator Generator for Zstandard Decompression

Junsun Choi

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-236

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-236.html>

December 20, 2024



Copyright © 2024, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

A Hardware Accelerator Generator for Zstandard Decompression

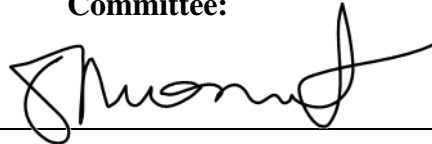
by Junsun Choi

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, in partial satisfaction of the requirements for the
degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



Professor Borivoje Nikolic
Research Advisor

Dec 19, 2024.

(Date)

* * * * *



Professor Sophia Shao
Research Advisor

Dec 20, 2024

(Date)

A Hardware Accelerator Generator for Zstandard Decompression

by

Junsun Choi

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Borivoje Nikolić, Co-chair

Professor Sophia Shao, Co-chair

Fall 2024

A Hardware Accelerator Generator for Zstandard Decompression

Copyright 2024
by
Junsun Choi

Abstract

A Hardware Accelerator Generator for Zstandard Decompression

by

Junsun Choi

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolić, Co-chair

Professor Sophia Shao, Co-chair

In hyperscale cloud computing systems, lossless data compression and decompression (referred to as “(de)compression”) is a widely used common low-level function that is heavily utilized across applications. However, (de)compression is unique as it trades off CPU cycles for reduced storage or network bandwidth and is considered a part of the *datacenter tax*. (De)Compression accounts for 3% of the fleet-wide CPU cycles in Google’s datacenters, ranking second only to protocol buffer serialization and deserialization. Accelerating (de)compression is anticipated to deliver significant total cost of ownership (TCO) savings in hyperscale systems. Among the various (de)compression algorithms, Zstandard decompression consumes the highest percentage of CPU cycles dedicated to (de)compression within Google’s fleet, accounting for 25.8%. This makes Zstandard decompression the top candidate for optimization through a custom hardware accelerator.

A large body of prior work has proposed enhanced microarchitectural features on the existing (de)compression hardware, but the effect of high-level design parameters on the feasibility of integrating the decompression hardware is not thoroughly explored. Therefore, this thesis presents a generator for Zstandard decompression accelerator that exposes design parameters for tuning including the history buffer size and the number of bits used in speculative Huffman decoding (referred to as “Huffman speculation bits”). The generator is open-sourced and is integrated into an open-source RISC-V SoC ecosystem for the performance and area evaluation of the accelerator designs with different placement configurations. With this approach, this thesis performs a design space exploration where the exploration range of the SoC speedup is 15.1x and that of the SoC silicon area is 1.5x. The design space exploration enables a better understanding of the impact of history buffer size, Huffman speculation bits, and accelerator placement on the SoC’s quality-of-result (QoR), leading to a in-depth assessment of different design strategies of the accelerated SoC for hyperscale systems. The

final optimized SoC with the Zstandard decompression accelerator is 5.6x faster than a single Xeon core while consuming a only a small portion (approximately 12 percent) of the Xeon core's area.

Contents

Contents	i
List of Figures	ii
1 Introduction	1
1.1 Optimizing Hyperscale Systems	1
1.2 Why Do We Need a (De)Compression Accelerator?	2
1.3 Previous Publication	3
2 Background	4
2.1 Lossless Compression Algorithm Fundamentals	4
2.2 Understanding the Fleet-wide (De)Compression Usage	7
2.3 Zstandard Decompression	8
3 Design of the Zstandard Decompression Accelerator Generator	11
3.1 System Interface Blocks	13
3.2 Huffman Decoder	13
3.3 FSE Decoder	15
3.4 LZ77 Decoder	17
4 Design Space Exploration	19
4.1 Methodology	19
4.2 Results	19
5 Conclusion	23
Bibliography	25

List of Figures

2.1	Example Huffman encoding [20]	5
2.2	Example LZ77 encoding. The arrow indicates the current position at each step.	6
2.3	Example LZ77 decoding, used in Zstandard sequence execution [7]	7
2.4	Fleet-wide CPU cycles spent on (de)compression over time. Source: [17]	8
2.5	Zstandard decompression [7]	9
2.6	FSE decoding in Zstandard decompression [7]	10
3.1	SoC Architecture with the decompression accelerator integrated	12
3.2	Architecture of the Zstandard decompression accelerator	13
3.3	4-bit Huffman Speculative Decoding Example	14
3.4	FSE Decode Table Reader	16
3.5	Simplified Diagram of the LZ77 Decoder	17
4.1	Effect of history buffer SRAM size and accelerator placement on speedup and area using 16-bit Huffman speculation	20
4.2	Effect of Huffman speculation bits, with 64K SRAM when integrated as RoCC (near-core) accelerator	21

Acknowledgments

I would like to first express my deepest gratitude to my advisors, Professor Borivoje Nikolić and Professor Sophia Shao, for their invaluable support and mentorship. This work would not have been completed without their thoughtful guidance.

This thesis expands upon work originally published in the Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA 2023) as “CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems.” I am particularly grateful to my primary collaborators of the CDPU paper, Sagar Karandikar and Joonho Whangbo. I would also like to thank Parthasarathy Ranganathan for his guidance of the research at Google, along with the collaborators at Google for their valuable input. My sincere appreciation extends to all co-authors of the CDPU paper, whose collective expertise helped shape this work.

Last but not least, I am profoundly thankful to my family for their unwavering and unconditional support throughout my graduate school journey.

Chapter 1

Introduction

1.1 Optimizing Hyperscale Systems

The importance of cloud computing in recent times cannot be overstated. Cloud has become the backbone of modern society, supporting everything from everyday tasks like web applications—such as search and email services—to the most computationally demanding tasks, including artificial intelligence (AI) and high-performance computing (HPC) applications. Since cloud relies on datacenters for its foundation, these datacenters are equally vital as the clouds. Today, a significant portion of the datacenters in the world is operated by *hyperscale* companies, or *hyperscalers* [11]. With the advantage provided by economies of scale, it is expected that hyperscale datacenters will continue to grow and capture an increasing share of the datacenter capacity, a benefit that traditional datacenters are unable to achieve to the same extent.

Given the scale of global users and the diversity of workloads in hyperscale cloud environments, which lead to a massive computation demand on clouds, the accompanied power consumption and the total cost of ownership (TCO) are non-trivial. In 2018, data centers already consumed 200 terawatt hours (TWh) of energy, which is 1% of global electricity demand [14]. This consumption is projected to grow in the following years [3] due to a number of trends. One trend is the explosion of traffic to and from datacenters, which grew from 50 exabytes in 2007 to 1.1 zettabytes¹ in 2017 [14]. More data to process and move means more power to burn. Another trend is the increasing size of machine learning models in scale leading to a larger energy demand [23].

Therefore, reducing the TCO is the primary goal for datacenter operators, especially for the hyperscalers, as cutting even a small portion of the CPU cycles can save terawatt hours of energy and millions of dollars in terms of the TCO in hyperscale environments. However, with the decline of Moore’s law, the performance improvements for workloads running on a hyperscaler’s server fleet are becoming less significant. Naturally, acceleration through a custom hardware accelerator (“accelerator” in this paper) is the next approach to achieve

¹Exabyte (10^{18} bytes); Zettabyte (10^{21} bytes)

performance gains. However, in hyperscale environments, as the workloads and the type of microservices are becoming considerably diverse, it is difficult to apply custom acceleration on all operations. Previous work has characterized different workloads or microservices in the cloud to explore custom acceleration opportunities.

Fortunately, there were a few common components that take a significant portion of the server fleet’s CPU cycles. Meta’s characterization of their top microservices found that a notable amount of cycles are spent on *orchestration*, such as compression, serialization, and I/O processing, even more than the core application logic [28]. Google’s profiling of its fleet workloads revealed that there is no “killer application” to optimize for, but there were six common low-level functions across binaries which is called the *datacenter tax* [15].

The six common functions are protocol buffer (protobuf) serialization and deserialization (“(de)serialization” in this paper), data compression and decompression (“(de)compression” in this paper), remote procedure calls (RPCs), memory allocation, data movement, and encryption/decryption. The characteristics of these functions are very similar to those of what Meta’s profiling categorized as orchestration. The CPU cycles spent on the six functions add up to 22 to 27 percent of the warehouse-scale (WSC) cycles. As these functions are widely used across applications and do not rapidly change over time, they are the top candidates for hardware acceleration. Inside Google’s infrastructure, protobuf (de)serialization consumes the most CPU cycles among the six datacenter tax functions with 9.6 percent of fleet-wide CPU cycles [16]. At UC Berkeley, we have already implemented an accelerator for protocol buffer serialization and deserialization, as published in [16].

1.2 Why Do We Need a (De)Compression Accelerator?

The datacenter tax function that consumes the second most fleet-wide CPU cycles, after protobuf (de)serialization, is data (de)compression. Data (de)compression consumes approximately 3 percent of the fleet-wide CPU cycles [17]. Although this number may look small, accelerating data (de)compression can potentially save gigawatts of power and millions of dollars in TCO.

Among the datacenter tax components, data (de)compression is different from others because the purpose of data (de)compression is to make a trade-off between CPU cycles and capacity or CPU cycles and bandwidth, while other datacenter tax functions are more for their functionalities. Thus, developing an accelerator that performs (de)compression much better than software creates an interesting chance to find a new point in the runtime-capacity trade-off space or the runtime-bandwidth trade-off space. For instance, we can use a compression algorithm that is slower but results in a smaller compressed file size if we have an accelerator, as we can run that algorithm faster with the accelerator. Moreover, equipping (de)compression accelerators in the system can make (de)compression more accessible by lowering the runtime cost, which can lead to a decision to use compression more extensively

in the fleet.

To sum, deploying a custom system-on-chip (SoC) for hyperscale systems provided with accelerators for data (de)compression will not only provide TCO savings in hyperscale context by outperforming the software radically but also open new opportunities to change the runtime versus data storage/bandwidth trade-off space. This paper focuses on the decompression accelerator, which is an important component of such SoC.

Although designing the accelerator is important, the adoption of the accelerator requires a careful investigation of the performance and area trade-off of different accelerator designs. In order to tackle this problem, in this paper, I present an RTL-based custom decompression accelerator generator that is configurable through parameters targeting a pervasively used compression algorithm at hyperscale systems called Zstandard. Next, I present the performance and silicon area evaluation results obtained by integrating the decompression processing unit into a RISC-V SoC with different decompression processing unit design parameters.

Extensive prior research including [1, 22, 13, 24, 5, 19, 25] has proposed enhanced microarchitectural features for better custom (de)compression acceleration opportunities, but their context fell short of evaluating the effect of design level parameters, such as accelerator placement in the SoC, history size, and more, which are the determining factors on the feasibility of accelerator integration into the system. Whether the accelerator placed as a near-core accelerator in the SoC or placed away from the core connected by PCIe or chiplet interconnect changes the I/O read and write latency cost for memory transactions issues from the accelerator. History size is the size of previously processed input to be stored for lookup and larger history size can lead to better (de)compression performance sacrificing the area cost. The assessment of these design parameters are crucial to judge the appropriateness of developing and integrating a (de)compression accelerator.

Using a generator-based approach enables a wide design space exploration. Running a Google fleet representative (de)compression benchmark, the exploration range is 15.1x in speedup and 1.5x in silicon area for different Zstandard decompression accelerator designs points. The best-performing decompression accelerator achieves 5.6x speedup compared to the Intel Xeon CPU, which is used to perform compression in real hyperscale environments, while only taking up only 12% of the area.

1.3 Previous Publication

The contents of this thesis is previously published as “CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems” [17] in collaboration with my co-authors. The graphs and numbers related to (de)compression in Google’s fleet, presented in Chapter 2.2 of the background section, draw extensively from the work of my co-authors at Google including Sagar Karandikar, Aniruddha Udipi, Svilen Kanev, Jyrki Alakuijala, and Parthasarathy Ranganathan. I would also like to state that the Huffman decoder presented in Chapter 3.2 was developed in collaboration with my colleague Joonho Whangbo.

Chapter 2

Background

2.1 Lossless Compression Algorithm Fundamentals

In this paper, the compression ratio is defined as the compressed file size divided by the uncompressed file size. The compression ratio is better when it is lower, as the compressed file size is small compared to the original file.

There are different types of lossless compression algorithms, such as Deflate [8], Zstandard [7], Snappy [10], and so on. Although these algorithms have different characteristics, most compression algorithms consist of compression primitives, and the algorithms can be interpreted as a mix of a few primitives. There are several popular primitives and they can be categorized into two classes: entropy coding and dictionary coding.

Entropy Encoding

The philosophy of entropy encoding is “A symbol that occurs more frequently should be encoded with fewer bits”. Widely used primitives in this category include Huffman encoding [12], arithmetic coding, and finite state entropy (FSE) encoding [6].

Let us take Huffman encoding as an example of entropy encoding. First, the frequency of each symbol in the input file is counted and the symbols are sorted by their frequency, where each symbol is now a node with a frequency value. Next, a Huffman tree is built by repeatedly merging the two least frequent nodes into a single parent node until one tree remains. Then, binary codes are assigned: 0 for left branches and 1 for right branches. Symbols with higher frequencies get shorter codes, and the encoded message is formed by replacing each character in the original data with its corresponding binary code. An example Huffman encoding corresponding to this procedure is described in Figure 2.1. It is shown that symbols with higher frequency is represented with less bits after encoding.

In Huffman encoding, the number of bits to represent an encoded symbol is a natural number so that the encoding scheme approximates the probability of appearance, which is the frequency of a symbol divided by the sum of the frequency of all symbols, to a power of 2. On the other hand, arithmetic coding, which uses a more exact probability distribution,

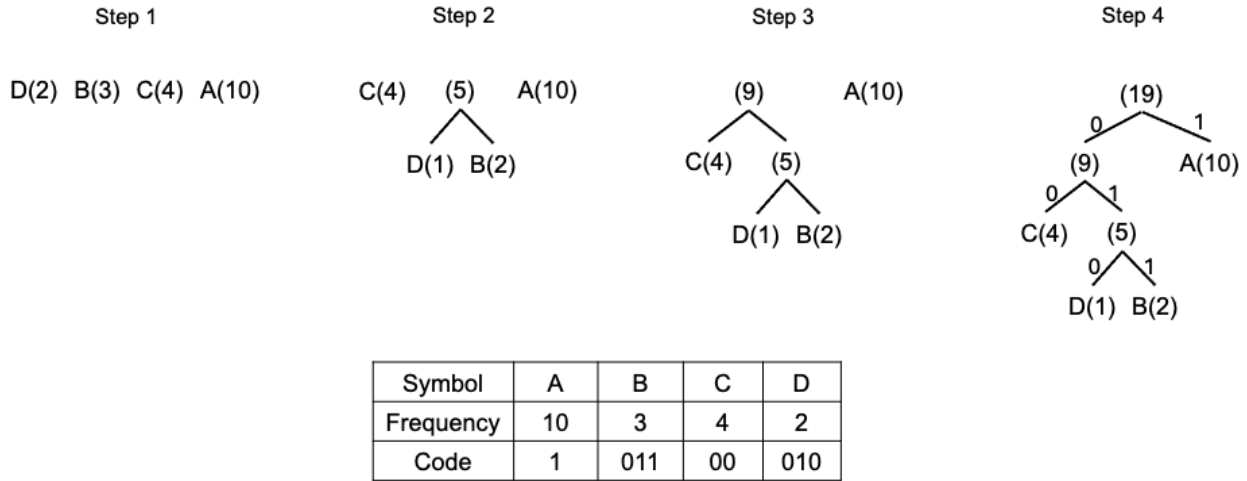


Figure 2.1: Example Huffman encoding [20]

achieves a better compression ratio with the sacrifice of computation speed. Asymmetric numerical system (ANS) encoding [9], including FSE, is in the middle ground with respect to the accuracy of the probability distribution, so that it achieves a decent compression ratio and computation speed.

Dictionary Encoding

A different approach from entropy encoding is dictionary encoding. In dictionary encoding, previously processed input symbols are stored as history in the history buffer. At each encoding iteration, a chunk of the input stream is compared to the history to find if there is the same pattern in both the current symbols and the history. If there is a such pattern, or a *match* in other words, the match is encoded to a shorter data structure. Finding a longer match and storing it in a short data structure format is the key to achieving a smaller encoded file size. As the likelihood of finding a longer match correlates with the maximum size of the history stored in the history buffer, the history buffer size is a determining factor of the compression ability of an algorithm that uses dictionary encoding.

One of the most popular compression primitives that can be classified as dictionary encoding is Lempel and Ziv’s LZ77 [32]. LZ77 algorithm encodes a match into a triplet of (offset, match length, literal length). This triplet is called as a *sequence*. The size of the sequence is usually shorter than the match if the match size is reasonably large. The meaning of each component of the sequence is as follows.

- Offset: Information to indicate where the match is from the current position.
- Match length: Number of symbols in a match, starting from (current position - offset).

- Literal length: Number of raw symbols, also known as *literals*, that do not belong to a match, following the match indicated by (offset, match length).

There are variants of LZ77 where the literal length is replaced with one literal symbol so that the sequence contains one literal after the match. However, popular compression algorithms such as Snappy and Zstandard use literal length as a component of the sequence instead of the literal. In this case, the literals have to be stored in a separate *literal buffer* in order to produce the original data during the decoding stage.

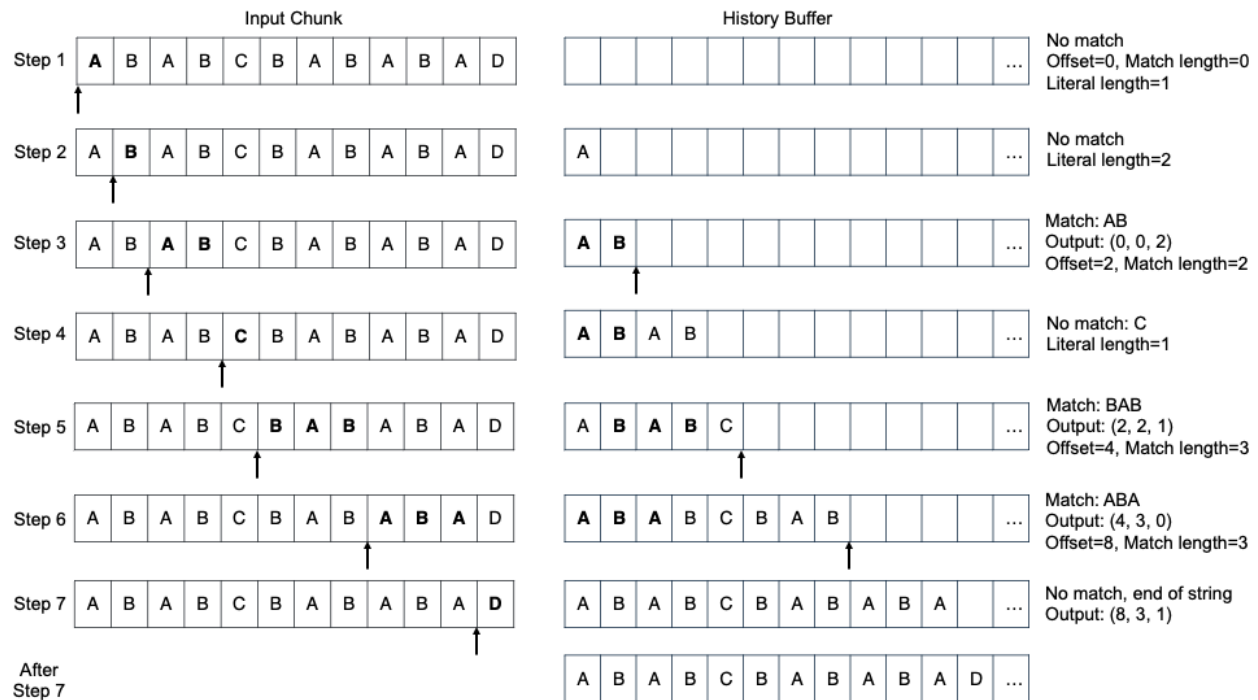


Figure 2.2: Example LZ77 encoding. The arrow indicates the current position at each step.

Figure 2.2 describes an example of how LZ77 encodes a string stream. Note that the history buffer is much longer than the input, even though the history buffer may graphically look similar in size to the input chunk. In step 1, the history buffer is empty, so there is no match found between the input and the history. Thus, the offset and the match length field of the sequence becomes zero. After step 1, the current input symbol 'A' should be put into the history buffer and the position advances to the next symbol. In every step, like step 1, the processed input is put into the history buffer and the position advances. In step 2, there is still no match between the current input symbol. In step 3, there is a match 'AB' between the current input stream and the history. Therefore, the literal is A and B from step 1 and step 2 and the produced sequence is (0, 0, 2), following the (offset, match length, literal length) format.

The match found in step 3 starts two symbols back from the current position, so the offset of this match is 2. The match length is 2 as the match is two symbols. The match is put into history. The literals following this match is 'C', as the 'BAB' is the following match found in step 5, so the literal length becomes 1. Then, the produced sequence is (2, 2, 1). The same procedure continues to reach the end of the input stream. When there are consecutive matches, such as in step 6, the literal length becomes zero.

LZ77 decoding can be done to restore the original stream, which is the input stream during decoding, with the encoded sequences and the literal buffer. An example decoding procedure corresponding to Figure 2.2 is depicted in Figure 2.3.

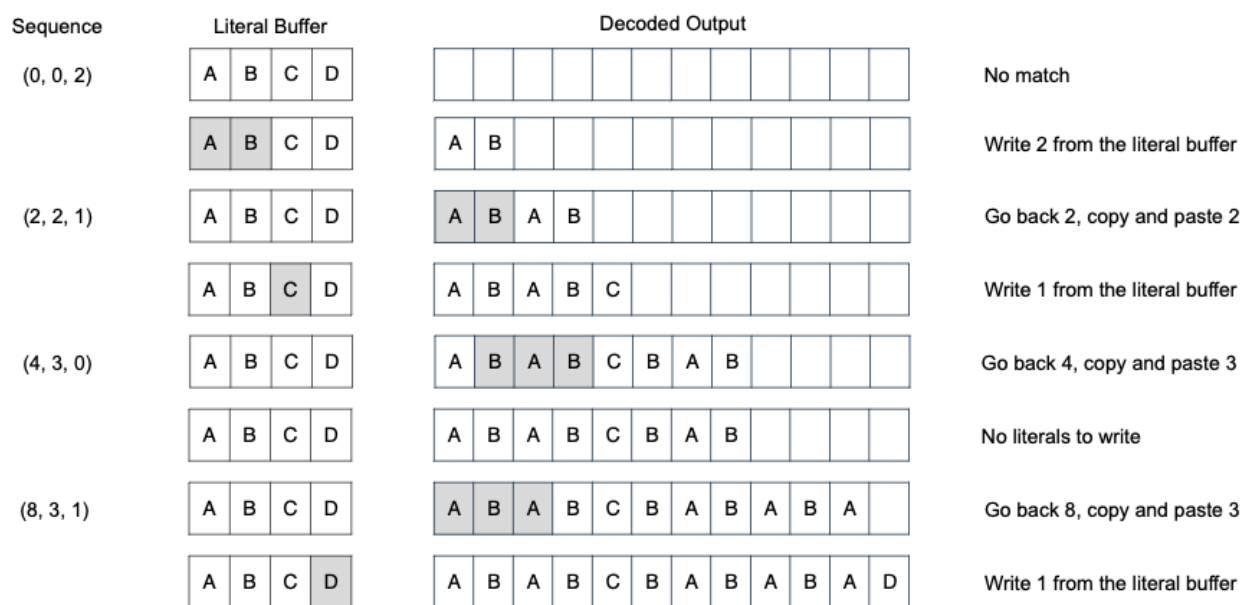


Figure 2.3: Example LZ77 decoding, used in Zstandard sequence execution [7]

2.2 Understanding the Fleet-wide (De)Compression Usage

Inside Google's server fleet, a number of algorithms are used for data (de)compression. The profiling of fleet-wide (de)compression usage in Google's datacenters shows that Zstandard and Snappy are the most-used compression algorithms regarding CPU cycles. The related data is displayed in Figure 2.4. At the latest time point when the profiling was performed, among the CPU cycles spent on compression and decompression, 41.2 percent is consumed in Zstandard—15.4 on compression and 25.8 percent on decompression. 39.8 percent is spent on

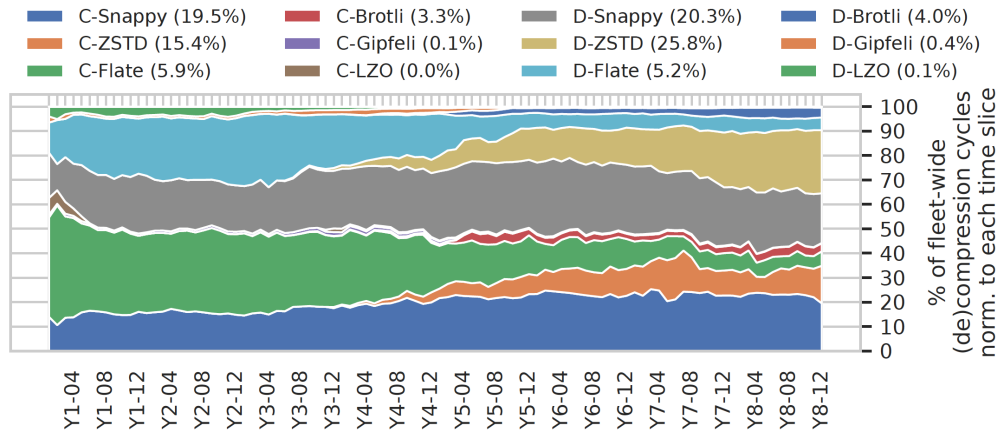


Figure 2.4: Fleet-wide CPU cycles spent on (de)compression over time. Source: [17]

Snappy—19.5 percent on compression and 20.3 percent on decompression. Other algorithms such as Flate (5.9%, 5.2% respectively on compression and decompression), Brotli (3.3%, 4.0%), Gipfeli (0.1%, 0.4%), and LZ0 (0.0%, 0.1%) take up the rest of the cycles.

One thing to note is the long lifetime of the compression algorithms. Profiling (de)compression over several years shows that compression algorithms being used in the fleet do not change in general. Also, the API for using these algorithms, which is taking an input stream and generating an output stream, does not change over time. This makes data (de)compression a good target to build a customized SoC for it.

Another prominent trend over time is that with the advent of Zstandard, Zstandard has been replacing the usage of other algorithms like Flate. Although the set of compression algorithms being used do not change quickly, when there is an algorithm that outperforms similar algorithms in terms of the compression ratio-runtime trade-off, the usage of that algorithm rises rapidly. To handle this case, a generator-based approach where the compression primitives commonly used in multiple popular algorithms exist as modules in the SoC can be beneficial to adjust to the emergence of a new algorithm.

As numbers present, Zstandard decompression expends the most fleet-wide CPU cycles among any algorithm-compression/decompression pair. This fact motivates the implementation of an accelerator for Zstandard decompression. Before diving into the design of the accelerator, the next section will explain how Zstandard decompression works.

2.3 Zstandard Decompression

Zstandard decompression first reads the header of the compressed file stream to retrieve metadata essential to further decompression. The next portion after the header of the input stream is decompressed by the Huffman decoding algorithm to produce literals. The rest of the compressed data is decompressed by the FSE decoding algorithm to produce

sequences. The literals and sequences are fed into the LZ77 decoding algorithm to produce the decompressed file stream output.

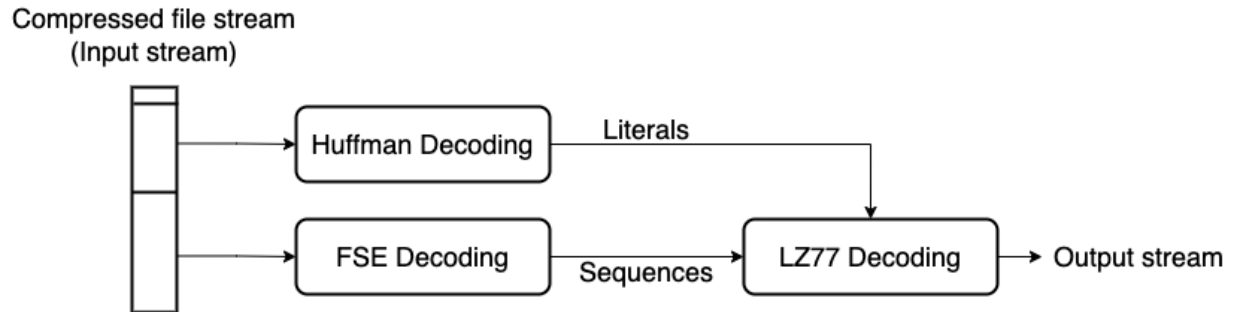


Figure 2.5: Zstandard decompression [7]

The Huffman decoding is an inverse operation of the Huffman encoding in Chapter 2.1. The LZ77 decoding process is described in Chapter 2.1. FSE decoding in Zstandard relies on a FSE *decode table*. The FSE decode table is retrieved by the normalized frequency statistics of each symbol which are stored in the compressed file stream during Zstandard compression. A decode table entry consists of four values: *nextState*, *nbAdd*, *nbBits*, and *baseValue*. Decompression is done by reading a decode table entry and the compressed file stream as follows:

- Symbol output = $\text{baseValue} + \text{nbAdd}$ bits from the compressed file stream
- Index of the next entry to read = $\text{nextState} + \text{nbBits}$ bits from the compressed file stream

As FSE decoding produces the (offset, match length, literal length) sequences to be used in LZ77 decoding, each component of the sequence has its own decode table. Therefore, FSE decoding in Zstandard requires building and reading three decode tables.

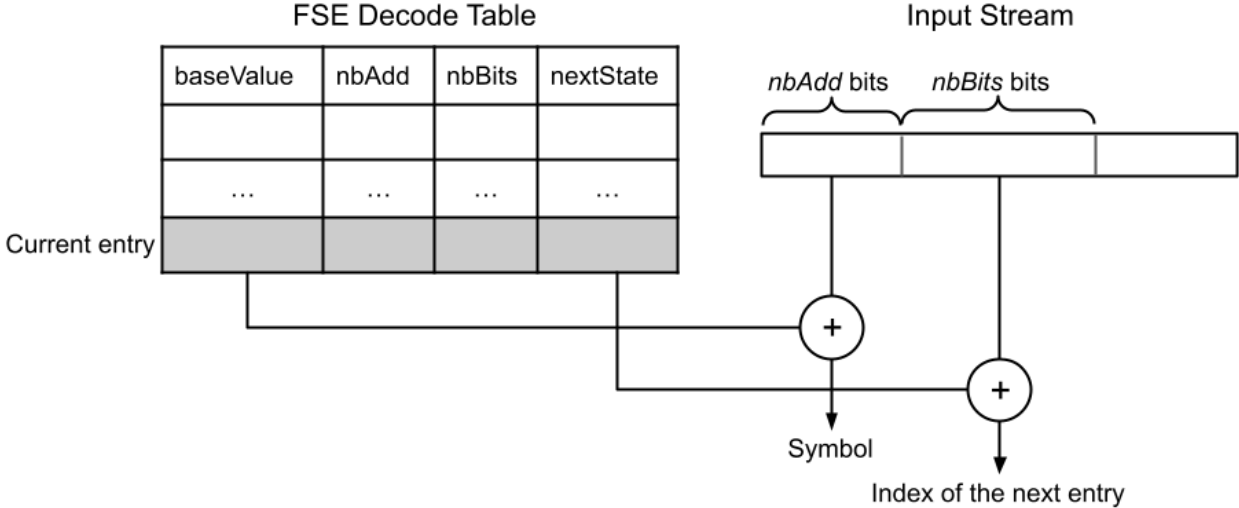


Figure 2.6: FSE decoding in Zstandard decompression [7]

Chapter 3

Design of the Zstandard Decompression Accelerator Generator

Zstandard decompression accounts for the largest proportion of all fleet-wide CPU cycles spent on compression and decompression. Consequently, developing a hardware accelerator for Zstandard decompression is a must to save the TCO of hyperscale systems. From this chapter, this dedicated hardware accelerator for decompression will be called as the *decompression processing unit*, or a decompression accelerator.

A generator for the Zstandard decompression processing unit is needed due to the following reasons.

1. Design space exploration: Most importantly, effectively evaluating the trade-offs between decompression performance and hardware complexity is the ultimate goal. A generator is essential to explore the design space with different parameters.
 - a) Determining the history buffer size: A larger history buffer can effectively handle Zstandard decompression calls that require a larger history size, but the history buffer SRAM cost will be too expensive above a certain size. The history buffer size of the decompression accelerator should be exposed as a design parameter to be explored to make a decision.
 - b) Determining the accelerator placement: Accelerator placement is directly related to the accelerator's I/O read and write latency, hence affecting the decompression accelerator's performance heavily, as decompression inherently requires numerous reads and writes. Where the decompression accelerator is integrated into the system should be exposed as a design parameter to be explored to make a decision.
2. Preparing for a sudden emergence of a new compression algorithm: As shown in Figure 2.4, Zstandard's share in the fleet-wide (de)compression CPU cycles grew 10 percent in one year from the algorithm's introduction. An agile hardware development method is needed in case of an occurrence of a similar phenomenon.

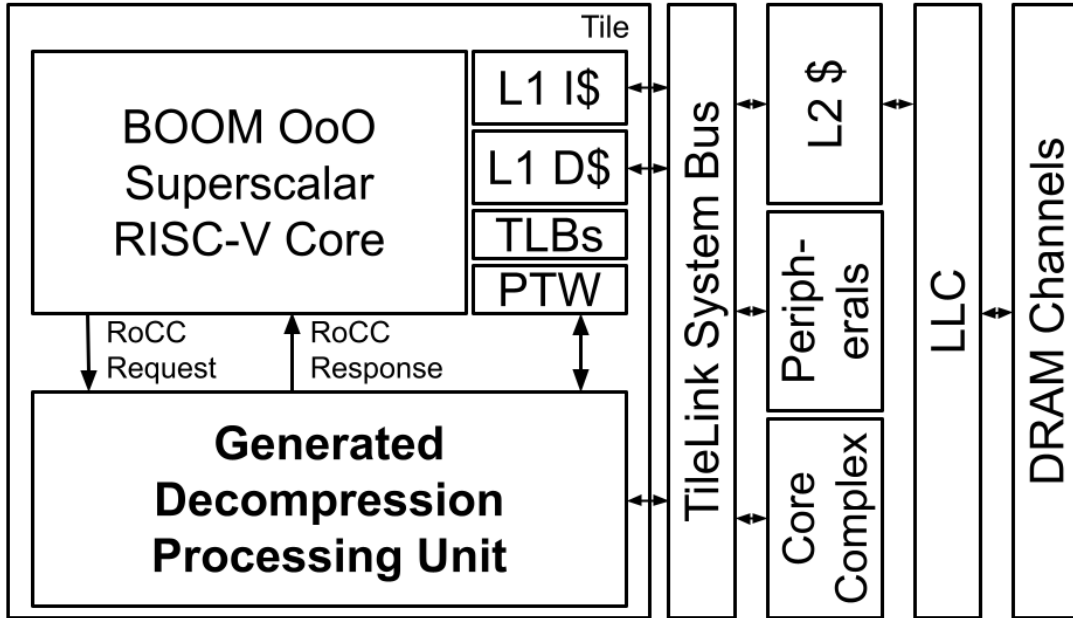


Figure 3.1: SoC Architecture with the decompression accelerator integrated

Therefore, this thesis proposes a Zstandard decompression processing unit generator that exposes important high-level design parameters for tuning.

The Zstandard decompression processing unit generator is implemented in Chisel RTL [4]. The generated decompression processing unit is integrated into the Chipyard RISC-V SoC generator ecosystem [2], as described in Figure 3.1. The SoC with the accelerator contains the BOOM out-of-order superscalar RISC-V core, whose performance is IPC-comparable to the ARM A72 core [31]. The accelerator is connected to the BOOM core via the RoCC interface where the CPU can directly issue custom RISC-V instructions called RoCC instructions. Two 64-bit register values can be sent through a RoCC instruction. The accelerator can access the CPU’s memory space with virtual addressing through 256-bit wide TileLink network-on-chip (NoC) [26]. Accordingly, the accelerator can access the L2 cache and the LLC through the TileLink system bus, as shown in Figure 3.1. The Zstandard decompression processing unit generator is open-sourced as a part of the CDPU framework [17].

The rest of this chapter explains the blocks of a generated Zstandard decompression processing unit. The blocks can be found in Figure 3.2.

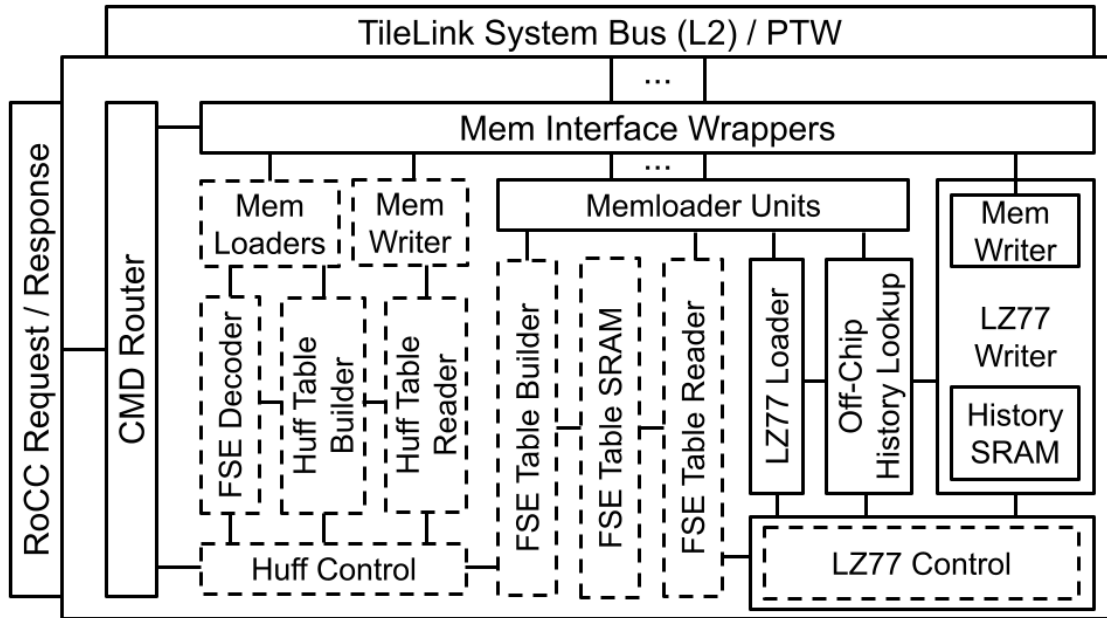


Figure 3.2: Architecture of the Zstandard decompression accelerator

3.1 System Interface Blocks

Memloaders and Memwriters are modules that facilitate streaming from and to the L2 cache. A Memloader queues read requests and sends them to the L2 cache, queues the responses from the L2 cache in buffers, and shifts the buffers to provide data as much as the consumer requires. Similarly, a Memwriter receives the data to write from the producer, gathers the data in buffers, manipulates the order of data coming from the buffers, then issues a write request to the L2. The Command router illustrated as CMD Router in Figure 3.2 receives the RoCC instructions from the core and issues the right commands to the appropriate module, then sends a response back to the core when necessary.

3.2 Huffman Decoder

The Huffman decoder consists of Huff Control, Huff Table builder, and Huff Table Reader drawn in Figure 3.2. To perform Huffman decoding of the input stream, the decoder builds and reads the Huffman table. The input stream contains Huffman codes, which are the Huffman-encoded version of the original symbols. The Huffman table is indexed by the Huffman code whose length is fixed to the maximum Huffman code length (11 bits, defined by the Zstandard algorithm). A Huffman table has two columns, the *number of bits* column and the *symbol* column. To obtain the original symbols, the Huff table reader reads 11 bits from the input stream and indexes the Huffman table using that 11-bit code. The value of

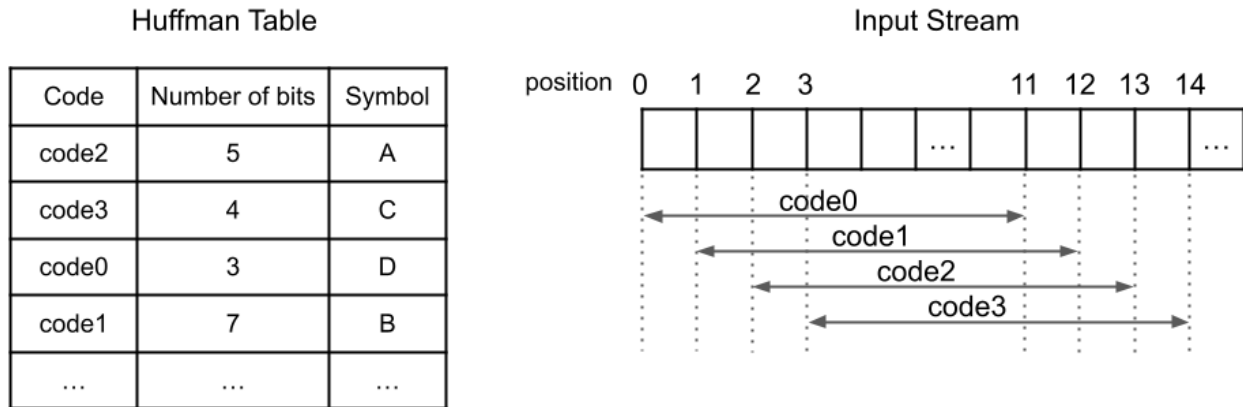


Figure 3.3: 4-bit Huffman Speculative Decoding Example

the *symbol* column of that entry is the decoded symbol. The value of the *number of bits* indicates how many bits should be consumed from the input stream. For example, if the *number of bits* value is 5, only 5 bits out of the 11 bits read from the input stream are consumed and the Huff table reader resumes reading the input stream from 5 bits after the current position.

Speculative Decoding

Since the number of bits to read is determined by reading the decode table entry, the starting position of the 11-bit code is unknown before decoding the last code. Therefore, Huffman decoding is innately serial. The serial nature of Huffman decoding is not optimal for the accelerator’s performance. To cope with this problem, the Huffman decoder speculatively decodes by performing code lookups at multiple starting positions, similar to the IBM z15 accelerator [1]. The number of positions to perform speculative decoding is parameterized as a design parameter of the Zstd decompression accelerator generator.

Figure 3.3 demonstrates an example of Huffman speculative decoding using 4 speculation bits. The Huffman decoder reads 11 bits from positions 0, 1, 2, and 3. The Huffman decoder then performs parallel indexing of the Huffman table using these four codes. This does not mean symbols from all four Huffman table entries will be considered as the decoded output. The Huffman decoder selectively chooses the symbols rooted from codes read from valid starting positions. Let us take a look at what should be done. The decoder indexes the Huffman table with code0. The symbol in the corresponding entry is D and the number of bits is 3. Thus, 3 bits starting from the input stream position 0 should be consumed and the next position to read 11 bits is position 3, which means code1 and code2 are not valid values. The decoder only emits symbols indexed by code 0 and code 3, so the output at this decode iteration is DC. The next decoding iteration will start from position 7.

It is expected that utilizing more speculation bits during Huffman decoding, which is to use more input stream positions to read 11-bit codes and perform Huffman table lookup in parallel, will boost the Huffman decoder throughput, as the number of output symbols emitted per iteration is likely to increase. However, there is a limit to the speculation bits as using more speculation bits accompanies increased hardware complexity by requiring more parallel look-ups of the Huffman table. The added hardware cost will differ by implementation details. The current design implements the Huffman table as registers. In this case, the number of register file ports will be proportional to the speculation bits. Thus, utilizing speculation bits over a certain amount may not be realistic depending on the silicon area budget. Also, the speculation bit should not exceed the L2 bandwidth of the system.

3.3 FSE Decoder

A FSE decoder is composed of a FSE decode table builder, FSE decode table reader, and FSE decode table SRAM. Instead of the L2 cache, FSE decode table built by the decode table builder is written to the FSE decode table SRAM. The decode table reader reads the table entries from that SRAM to produce sequences.

FSE Decode Table Builder

By parsing the first few bytes of the input stream, the FSE decode table builder obtains the number of sequences to produce and the compression modes of offset, match length, and literal length. After getting the number of sequences and compression modes, the decode table builder generates decode tables for offset, match length, and literal length depending on the mode.

Compression mode 0 means using a predefined decode table, which is a fixed table already determined by the Zstandard algorithm. In this Zstandard decompression processing unit, these predefined tables exist as a hard-wired read-only memory (ROM), so the decode table builder skips the building and tells the decode table reader to use the predefined table. Compression mode 1 is the run-length encoding (RLE) mode, where the decode table only consists of a single symbol value used for all sequences. The decode table builder reads the next byte of the input stream to find out what the symbol is. Then it writes the only decode table entry to the decode table SRAM. Compression mode 2 is when the sequences are compressed in the standard FSE algorithm. This case will be explained in the next paragraph. Compression mode 3 is repeat mode where the previously built decode table is used. In this case, the decode table builder does not build a new decode table.

For compression mode 2, the FSE decode table builder first recovers the normalized count statistics of symbols by reading the input stream. Using the statistics, the decode table entries can be retrieved. The challenge of building this module is that the decode table entries are dependent on each other. The key to achieving high throughput is to minimize memory accesses and write as many bytes in one write request. Small tables

are implemented as ROMs, intermediate data structures to calculate the table entry are implemented as registers instead of a region in the L2 cache, and all fields of an entry are processed and written to the decode table SRAM in one cycle.

FSE Decode Table Reader

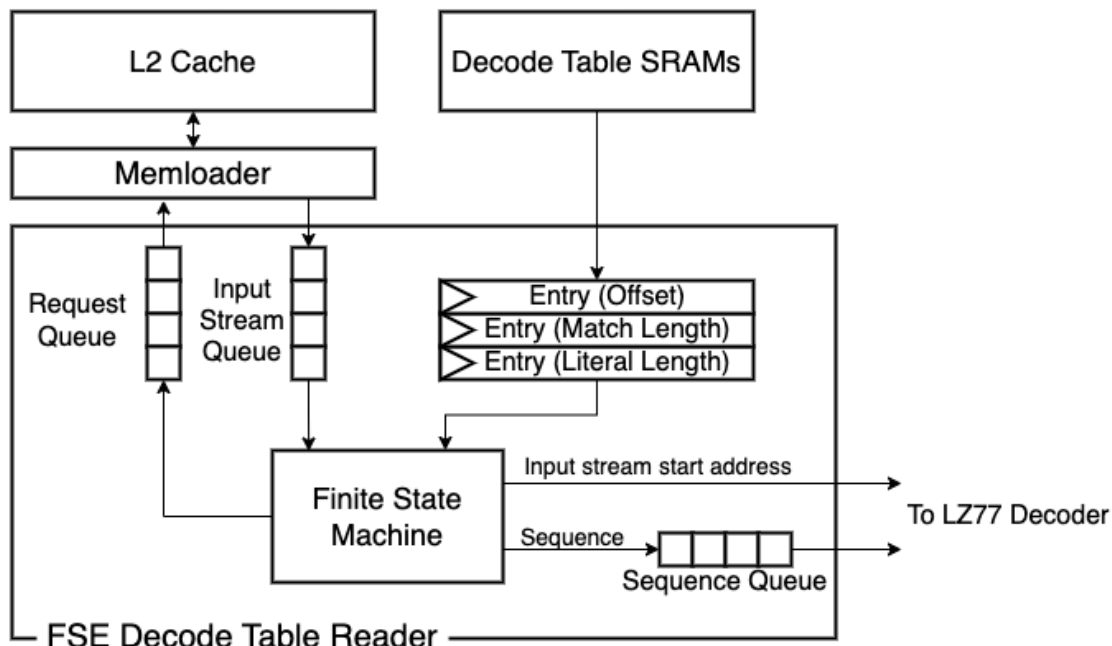


Figure 3.4: FSE Decode Table Reader

Figure 3.4 illustrates the internals of the FSE decode table reader. The decode table reader loads the appropriate decode table entry from the SRAM where it is stored and the input stream from the L2 cache. The finite state machine (FSM) deals with the processing of the table entry and input stream to recover the sequences.

To maximize performance, memory requests and computation are overlapped. The L2 read request for the input stream is put into the request queue whenever the queue is not full, so the L2 read request can be overlapped with the computation to produce the sequences. Also, requesting the decode table entries is also overlapped with computation as in state 4 the request to SRAM and computation for sequence production is done in the same cycle. The generated sequences are directly fed to the LZ77 decoder through a ready-valid interface.

The decode table reader is able to emit one sequence at a cycle to the LZ77 decoder after a small setup time. This rate is fast enough considering the LZ77 decoder's speed of decoding a sequence.

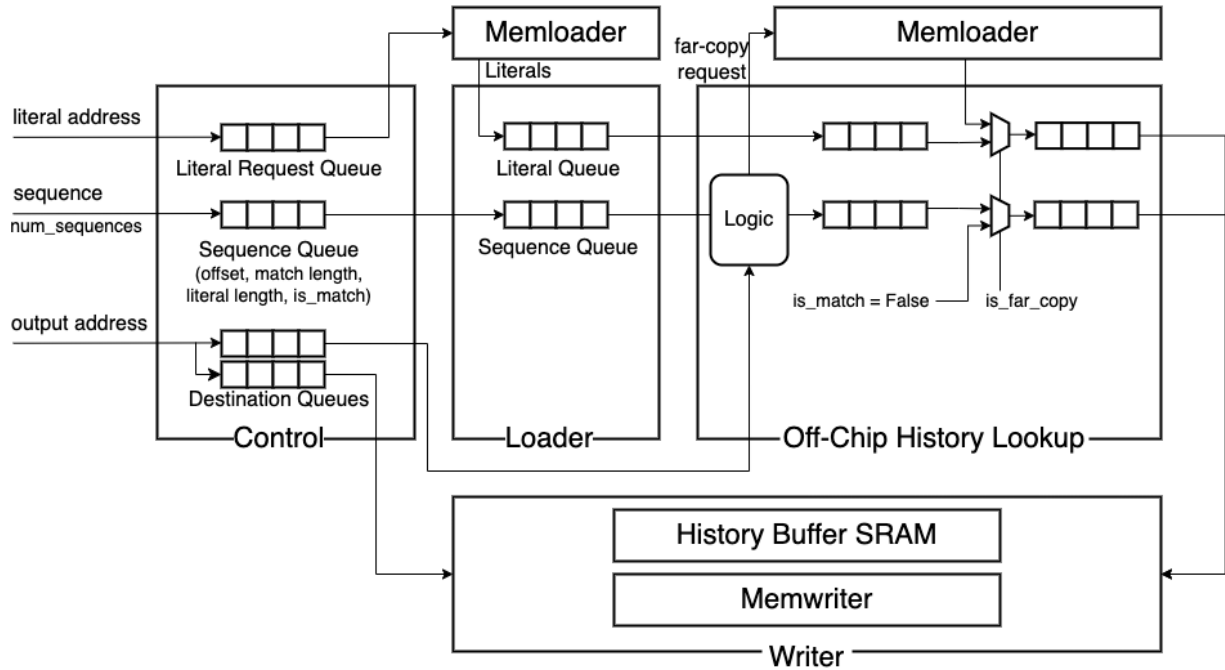


Figure 3.5: Simplified Diagram of the LZ77 Decoder

3.4 LZ77 Decoder

The LZ77 decoder is composed of LZ77 control, LZ77 loader, off-chip history lookup, and LZ77 writer, as noted in Figure 3.2.

LZ77 Control and Loader

The LZ77 control receives the necessary information from the Huffman decoder and the FSE decoder and dispatches appropriate commands to the LZ77 loader, off-chip history lookup, and writer. From the FSE decode table reader, the LZ77 control takes the sequences to decode and the total number of sequences to decode. The Huffman decoder consumes a portion of the compressed file stream (input stream) to produce literals and stores the literals in an L2 cache address. The Huffman decoder sends the address of the start of these literals to the LZ77 control. The LZ77 control also obtains the address where the output is going to be written, from the command router.

The LZ77 control processes the inputs and puts them into queues. Using the literal address, the LZ77 control sends requests to the memloader to read the literals from the L2 cache. The LZ77 loader is responsible for receiving the literals coming from the memloader, slicing the literals, and putting the literal chunks into the literal queue.

Off-Chip History Lookup

The LZ77 off-chip history lookup deals with matches that do not fit into the range of the history buffer. The offset of a sequence can exceed the history buffer SRAM size in the LZ77 decoder. The off-chip history lookup logic detects such far-copy by looking at the sequence information and sends an L2 read request to the associated memloader to retrieve the match. In this case, the match given by the memloader is put into the literal queue and the sequence is no longer considered to have a match, as described in Figure 3.5.

LZ77 Writer

The LZ77 control sends the output address to the LZ77 writer and the off-chip history lookup sends the literal chunks and sequence to the LZ77 writer. The LZ77 writer contains the history buffer SRAM so that matches whose offset is smaller than the history buffer size can be copied from the history buffer SRAM to be written as the next output. As matches that do not fall into the history buffer's range are already handled in the off-chip history lookup, all matches are handled. The output is written to the L2 cache through the memwriter inside the LZ77 writer and at the same time to the history buffer SRAM.

Chapter 4

Design Space Exploration

4.1 Methodology

Three design knobs are controlled in the design space exploration experiment: history buffer SRAM size, accelerator placement, and the number of bits used in speculative Huffman decoding (“Huffman speculation bits” in this thesis).

The design space exploration (DSE) of the accelerated SoC implemented in RTL is performed by using FireSim [18], which produces deterministic cycle-exact performance numbers by modeling designs and I/Os cycle-accurately. The experiments are run on the HyperCompressBench benchmark [17], which is an open-source benchmark suite representative of the Google fleet’s (de)compression requirements.

The HyperCompressBench files are run on two systems—(1) a single-core RISC-V system where the Zstandard decompression processing unit integrated is into (2) a baseline Xeon server. The BOOM core and the Zstandard decompression accelerator of the RISC-V system are modeled at 2 GHz frequency. The baseline system is a one-core two-HT Xeon E5-2686 v4-based server at 2.3GHz/2.7GHz turbo frequency. Performance results of the RISC-V system measured the end-to-end runtime of an entire decompression call without overlapping jobs. The performance results of the Xeon are collected by running *lzbench* [27], which is an established tool used for benchmarking in-memory (de)compression algorithms. The throughput is calculated as each benchmark file size divided by the time took to finish decompressing. The overall throughput is the sum of all benchmark file sizes divided by the sum of all decompression runtime.

The area estimation numbers of the RISC-V system are obtained by advancing the designs through synthesis [29] for Intel’s 16nm-class process.

4.2 Results

Figure 4.1 shows the speedup and area results from a decompression processing unit generated all with 16 Huffman speculation bits but with different history buffer SRAM sizes

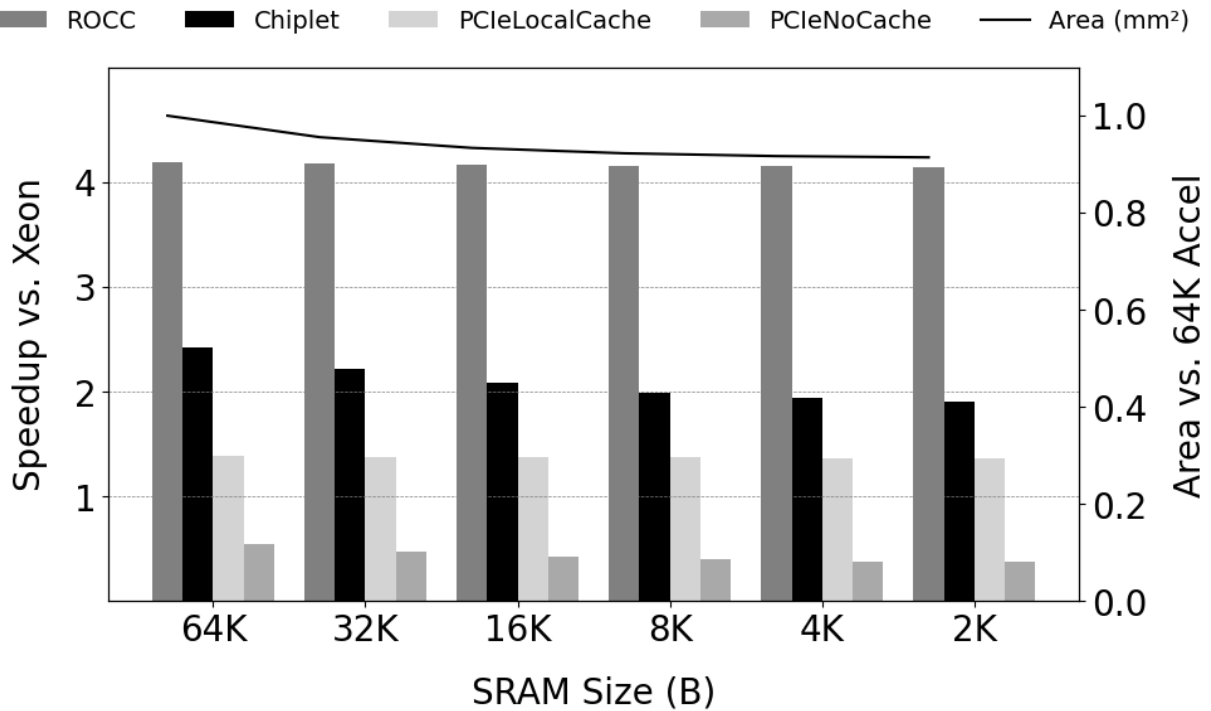


Figure 4.1: Effect of history buffer SRAM size and accelerator placement on speedup and area using 16-bit Huffman speculation

and accelerator placements. The history buffer SRAM size changes from 64K to 2K in this figure.

In the design space exploration experiment, four accelerator placement parameters are used: *RoCC*, *Chiplet*, *PCIeNoCache*, *PCIeLocalCache*. *RoCC* is when the accelerator is placed near-core, so no latency is injected for memory requests. *Chiplet* is modeled by injecting 25ns latency for all requests from the accelerator. *PCIeNoCache* is when the accelerator is integrated over PCIe+DDIO, without a cache attached to the PCIe card. *PCIeNoCache* is modeled by injecting 200ns latency [21] for all requests. *PCIeLocalCache* models a shared on-die cache and local DRAM attached to the PCIe card. 200ns latency is injected only for the decompression input and output, not for any intermediate read and writes.

The design with the highest speedup compared to the Xeon CPU achieves 4.2x speedup (3.95 GB/s) using the most area among the designs explored. The silicon area of this design is 1.899 mm², which is only 10.56 percent of the area of the Xeon Core Tile of 17.98 mm² in 14nm [30].

Including a large history buffer SRAM leads to higher speedup for all placements, but the performance gain of using a large history buffer SRAM compared to its area overhead is negligible. For example, the SoC that has the decompression accelerator integrated as a

RoCC accelerator with a 64K SRAM is 1.2 percent faster than that with a 2K SRAM while using 9.4 percent more area.

Even with 64K SRAM, the speedup is considerably worse if the accelerator is integrated over PCIe. The speedup of the PCIeLocalCache case is 3.0x lower and that of the PCIeNoCache case is 7.72x lower. This is owing to (1) higher read and write cost over PCIe than the near-core case and (2) the fact that there are many small decompressions in the fleet. The PCIeLocalCache case mitigates this effect by modeling the shared on-die cache and local DRAM. The performance degradation due to using smaller SRAM is not as much as the PCIeNoCache case.

The performance of the *Chiplet* setting in Figure 4.1 is worse than the *RoCC* but better than PCIe options. It is up to the system operator to choose this option considering the performance, area, and chiplet integration cost. The 64K SRAM configuration achieves a 2.42x speedup compared to the Xeon. At smaller history buffer sizes, the performance gets worse as more read and write requests have to go through the chiplet interconnect, which would be the bottleneck to the performance. At the smallest history buffer size, the performance is almost similar to the PCIe case.

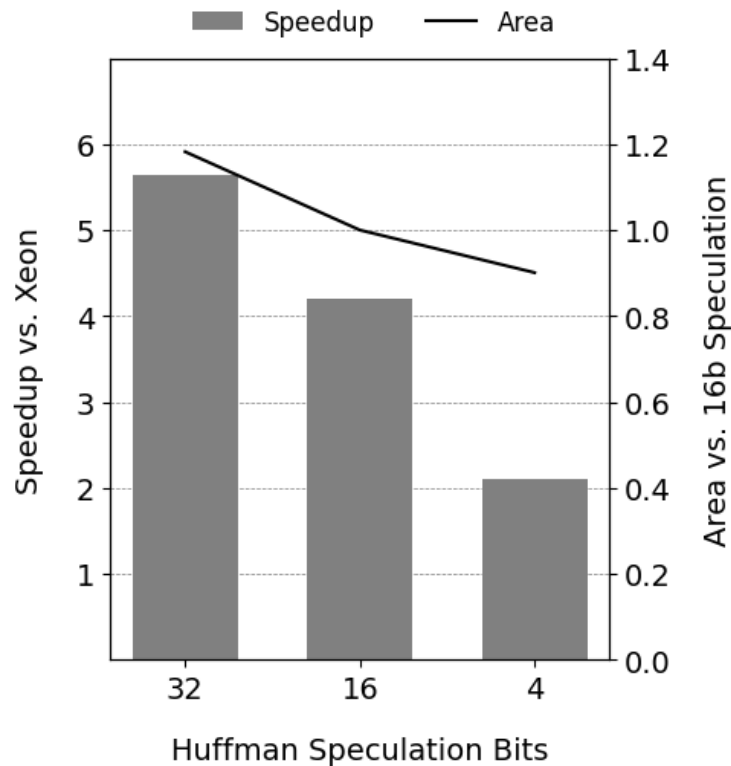


Figure 4.2: Effect of Huffman speculation bits, with 64K SRAM when integrated as RoCC (near-core) accelerator

Figure 4.2 shows the impact of the Huffman speculation bits on the performance and area when the decompression processing unit is integrated as a near-core accelerator with history buffer SRAM size fixed to 64K. The 16-bit Huffman speculation case corresponds to the configuration used in Figure 4.1. 32-bit speculation is what is used in the IBM z15 (de)compression accelerator [1] and 4-bit speculation represents the minimal reasonable design point.

The change in the Huffman speculation bits creates a large variation in the quality-of-result (QoR) of the decompression processing unit’s design. The 32 Huffman speculation bits design achieves 5.64x speedup (5.304 GB/s) over the Xeon while consuming only about 12 percent of Xeon’s area (2.245 mm²). This is an 18 percent bigger area and 34 percent higher speedup than the 16-bit speculation design. On the other hand, the 4-bit speculation design only achieves 2.11x speedup compared to the Xeon. The silicon area of the 4-bit speculation design (1.711 mm²) is 10 percent less than the 16-bit speculation design and the speedup of the 4 bits design is 50 percent lower than the 16 bits design. As the numbers present, doubling the Huffman speculation bits is more influential on the speedup and area than doubling the history SRAM size is.

The results of the design space exploration prove that consideration of high-level parameters of the decompression processing units is crucial to understanding the design QoR and the microarchitectural design of the accelerator is not the sole factor that determines the QoR. This justifies the generator approach used in this paper which parameterizes the high-level knobs. The key findings include

- The Huffman speculation bits significantly influence the accelerator’s QoR, far more than the history buffer size.
- Integrating the Zstandard decompression unit as a near-core accelerator is recommended. Integrating the accelerator over PCIe is not recommended because its performance matches or falls short of the Xeon’s.
- The fastest design point among the explored designs is 5.6x faster than the Xeon while requiring about 12% of the Xeon’s area.

Chapter 5

Conclusion

This thesis presented an open-source RTL-based Zstandard decompression processing unit generator that enables tuning many design knobs. The generated accelerator is integrated into an open-source RISC-V SoC ecosystem that is capable of fast evaluation of the performance and silicon area of an SoC with different design parameters. Many existing works related to (de)compression accelerators demonstrate enhanced microarchitecture in a constrained design point. Instead of presenting a single microarchitectural design point, the generator-based approach of this paper enables the evaluation of the effect of high-level design parameters that are critical to judging the appropriateness of integrating the accelerator, such as history buffer size and Huffman speculation bits, which are not covered thoroughly in previous works.

The key achievements of this thesis are as follows:

- Comprehensive sweeping of the decompression processing unit’s design space, spanning a 15.1x range in accelerator speedup and a 1.5x range in silicon area.
- Design space exploration whose results led to a better understanding of the effect of each design parameter and how to optimize the design considering the integration into a hyperscale system.
- Finding a hyperscale-optimized design which is 5.6x faster than a single Xeon core while consuming approximately 12 percent of a Xeon core’s area.

Future work

The end goal of developing the (de)compression accelerator generator is to have a universal (de)compression accelerator that works on a wide range of compression algorithms. Currently, the Zstandard decompressor presented in this paper is already modularized to a set of decoders of three compression primitives: Huffman, FSE, and LZ77. Ideally, any algorithm that uses these primitives as its building blocks can reuse the accelerator. However,

the accelerator control is now configured to follow Zstandard's file format. In order to support other algorithms, the logic related to producing the output according to the format should be reconfigurable to algorithms other than Zstandard. Small changes are necessary to reuse the accelerator because of this. For example, to reuse the LZ77 decoder for Snappy decompression, the LZ77 control module should change to follow the Snappy file format and the LZ77 loader should include a variable integer (varint) decoder. If this challenge is solved and a completely modularized reconfigurable (de)compression accelerator is available, agile development will be possible in case of a sudden emergence of a new compression algorithm.

Bibliography

- [1] Bulent Abali et al. “Data compression accelerator on IBM POWER9 and z15 processors”. In: *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ISCA '20. Virtual Event: IEEE Press, 2020, pp. 1–14. ISBN: 9781728146614. DOI: 10.1109/ISCA45697.2020.00012. URL: <https://doi.org/10.1109/ISCA45697.2020.00012>.
- [2] Alon Amid et al. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs”. In: *IEEE Micro* 40.4 (2020), pp. 10–21. DOI: 10.1109/MM.2020.2996616.
- [3] Anders S. G. Andrae and Tomas Edler. “On Global Electricity Usage of Communication Technology: Trends to 2030”. In: *Challenges* 6.1 (2015), pp. 117–157. ISSN: 2078-1547. DOI: 10.3390/challe6010117. URL: <https://www.mdpi.com/2078-1547/6/1/117>.
- [4] Jonathan Bachrach et al. “Chisel: constructing hardware in a Scala embedded language”. In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225. ISBN: 9781450311991. DOI: 10.1145/2228360.2228584. URL: <https://doi.org/10.1145/2228360.2228584>.
- [5] Jianyu Chen, Maurice Daverveldt, and Zaid Al-Ars. “FPGA Acceleration of Zstd Compression Algorithm”. In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, pp. 188–191. DOI: 10.1109/IPDPSW52791.2021.00035.
- [6] Yann Collet. *FiniteStateEntropy: New Generation Entropy coders*. <https://github.com/Cyan4973/FiniteStateEntropy>.
- [7] Yann Collet and Murray Kucherawy. *Zstandard Compression and the 'application/zstd' Media Type*. RFC 8878. Feb. 2021. DOI: 10.17487/RFC8878. URL: <https://www.rfc-editor.org/info/rfc8878>.
- [8] P. Deutsch. *RFC1951: DEFLATE Compressed Data Format Specification version 1.3*. USA, 1996.

- [9] Jarek Duda et al. “The use of asymmetric numeral systems as an accurate replacement for Huffman coding”. In: *2015 Picture Coding Symposium (PCS)*. 2015, pp. 65–69. DOI: 10.1109/PCS.2015.7170048.
- [10] Google. *Snappy: A fast compressor/decompressor*. <https://github.com/google/snappy>.
- [11] Synergy Research Group. *Hyperscale Operators and Colocation Continue to Drive Huge Changes in Data Center Capacity Trends*. <https://www.srgresearch.com/articles/hyperscale-operators-and-colocation-continue-to-drive-huge-changes-in-data-center-capacity-trends>. Reno, NV, Aug. 2024.
- [12] David A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.
- [13] Intel. *Intel® Infrastructure Processing Unit (Intel® IPU)*. <https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html>. 2023.
- [14] Nikola Jones. “How to stop data centres from gobbling up the world’s electricity”. In: *Nature* 561.7722 (2018), pp. 163–166. DOI: 10.1038/d41586-018-06610-y. URL: <https://doi.org/10.1038/d41586-018-06610-y>.
- [15] Svilen Kanev et al. “Profiling a warehouse-scale computer”. In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 2015, pp. 158–169. DOI: 10.1145/2749469.2750392.
- [16] Sagar Karandikar et al. “A Hardware Accelerator for Protocol Buffers”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 462–478. ISBN: 9781450385572. DOI: 10.1145/3466752.3480051. URL: <https://doi.org/10.1145/3466752.3480051>.
- [17] Sagar Karandikar et al. “CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems”. In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA ’23. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: 10.1145/3579371.3589074. URL: <https://doi.org/10.1145/3579371.3589074>.
- [18] Sagar Karandikar et al. “FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 29–42. DOI: 10.1109/ISCA.2018.00014.
- [19] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. “High-Throughput FPGA-Based Hardware Accelerators for Deflate Compression and Decompression Using High-Level Synthesis”. In: *IEEE Access* 8 (2020), pp. 62207–62217. DOI: 10.1109/ACCESS.2020.2984191.

- [20] Jan van Leeuwen. “On the Construction of Huffman Trees”. In: *Third International Colloquium on Automata, Languages and Programming, University of Edinburgh, UK, July 20-23, 1976*. Ed. by S. Michaelson and Robin Milner. Edinburgh University Press, 1976, pp. 382–410.
- [21] Rolf Neugebauer et al. “Understanding PCIe performance for end host networking”. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. SIGCOMM ’18*. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 327–341. ISBN: 9781450355674. DOI: 10.1145/3230543.3230560. URL: <https://doi.org/10.1145/3230543.3230560>.
- [22] NVIDIA. *NVIDIA BlueField-3 Datasheet*. <https://resources.nvidia.com/en-us-accelerated-networking-resource-library/datasheet-nvidia-bluefield.2023>.
- [23] David A. Patterson et al. “Carbon Emissions and Large Neural Network Training”. In: *CoRR* abs/2104.10350 (2021). arXiv: 2104.10350. URL: <https://arxiv.org/abs/2104.10350>.
- [24] Open Compute Project. *Project-Zipline*. <https://github.com/opencomputeproject/Project-Zipline>. 2021.
- [25] Sudhir Satpathy et al. “A 1.4GHz 20.5Gbps GZIP decompression accelerator in 14nm CMOS featuring dual-path out-of-order speculative Huffman decoder and multi-write enabled register file array”. In: *2019 Symposium on VLSI Circuits*. 2019, pp. C238–C239. DOI: 10.23919/VLSIC.2019.8777934.
- [26] SiFive Inc. *SiFive TileLink Specification*. 2019. URL: https://sifive.cdn.prismic.io/sifive%2Fcab05224-2df1-4af8-adee-8d9cba3378cd_tilelink-spec-1.8.0.pdf.
- [27] Przemyslaw Skibinski. *lzbench*. 2022. URL: <https://github.com/inikep/lzbench>.
- [28] Akshitha Sriraman and Abhishek Dhanotia. “Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS ’20*. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 733–750. ISBN: 9781450371025. DOI: 10.1145/3373376.3378450. URL: <https://doi.org/10.1145/3373376.3378450>.
- [29] Edward Wang et al. “A Methodology for Reusable Physical Design”. In: *2020 21st International Symposium on Quality Electronic Design (ISQED)*. 2020, pp. 243–249. DOI: 10.1109/ISQED48828.2020.9136999.
- [30] Wikichip. *Skylake (server) - Microarchitectures - Intel*. 2023. URL: [https://en.wikichip.org/wiki/intel/microarchitectures/skylake_\(server\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)).
- [31] Jerry Zhao et al. “SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine”. In: (May 2020).

- [32] J. Ziv and A. Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714.