An Efficient High-Throughput LZ77-Based Decompressor in Reconfigurable Logic

Fang, Jian; Chen, Jianyu; Lee, Jinho; Al-Ars, Zaid; Hofstee, Peter

DOI
10.1007/s11265-020-01547-w

Publication date
2020

Document Version
Final published version

Published in
Journal of Signal Processing Systems

Journal of Signal Processing Systems (2020) 92:931–947
https://doi.org/10.1007/s11265-020-01547-w

An Efficient High-Throughput LZ77-Based Decompressor in Reconfigurable Logic

Jian Fang[1,2] · Jianyu Chen[2] · Jinho Lee[3] · Zaid Al-Ars[2] · H. Peter Hofstee[2,4]

Received: 2 December 2019 / Revised: 8 March 2020 / Accepted: 5 May 2020 / Published online: 28 May 2020

Abstract

To best leverage high-bandwidth storage and network technologies requires an improvement in the speed at which we can decompress data. We present a “refine and recycle” method applicable to LZ77-type decompressors that enables efficient high-bandwidth designs and present an implementation in reconfigurable logic. The method refines the write commands (for literal tokens) and read commands (for copy tokens) to a set of commands that target a single bank of block ram, and rather than performing all the dependency calculations saves logic by recycling (read) commands that return with an invalid result. A single “Snappy” decompressor implemented in reconfigurable logic leveraging this method is capable of processing multiple literal or copy tokens per cycle and achieves up to 7.2GB/s, which can keep pace with an NVMe device. The proposed method is about an order of magnitude faster and an order of magnitude more power efficient than a state-of-the-art single-core software implementation. The logic and block ram resources required by the decompressor are sufficiently low so that a set of these decompressors can be implemented on a single FPGA of reasonable size to keep up with the bandwidth provided by the most recent interface technologies.

Keywords Decompression · FPGA · Acceleration · Snappy · CAPI

1 Introduction

Compression and decompression algorithms are widely used to reduce storage space and data transmission bandwidth. Typically, compression and decompression are computation-intensive applications and can consume significant CPU resources. This is especially true for systems that aim to combine in-memory analytics with fast storage such as can be provided by multiple NVMe drives. A high-end system of this type might provide 70+GB/s of NVMe (read) bandwidth, achievable today with 24 NVMe devices of 800K (4KB) IOPs each, a typical number for a PCIe Gen3-based enterprise NVMe device. With the best CPU-based Snappy decompressors reaching 1.8GB/s per core 40 cores are required just to keep up with this decompression bandwidth. To release CPU resources for other tasks, accelerators such as graphic processing units (GPUs) and field programmable gate arrays (FPGAs) can be used to accelerate the compression and decompression.

While much prior work has studied how to improve the compression speed of lossless data compression [9, 13, 24], in many scenarios the data is compressed once for storage and decompressed multiple times whenever it is read or processed.

Existing studies [17, 20, 25, 29, 30] illustrate that an FPGA is a promising platform for lossless data decompression. The customizable capability, the feasibility of bit-level control, and high degrees of parallelism of the FPGA allow designs to have many light-weight customized cores,

✉ Jian Fang
j.fang-1@tudelft.nl

Jianyu Chen
j.chen-13@student.tudelft.nl

Jinho Lee
leejinho@yonsei.ac.kr

Zaid Al-Ars
z.al-ars@tudelft.nl

H. Peter Hofstee
hofstee@us.ibm.com

[1] National Innovation Institute of Defense Technology, Beijing, China

[2] Delft University of Technology, Delft, The Netherlands

[3] Yonsei University, Seoul, Korea

[4] IBM Austin, Austin, TX, USA

enhancing performance. Leveraging these advantages, the pipelined FPGA designs of LZSS [17, 20], LZW [30] and Zlib [18, 29] all achieve good decompression throughput. However, these prior designs only process one token per FPGA cycle, resulting in limited speedup compared to software implementations. The studies [25] and [5] propose solutions to handle multiple tokens per cycle. However, both solutions require multiple copies of the history buffer and require extra control logic to handle BRAM bank conflicts caused by parallel reads/writes from different tokens, leading to low area efficiency and/or a low clock frequency.

One of the popular compression and decompression algorithms in big data and data analytic applications is Snappy [15], which is supported by many data formats including Apache Parquet [8] and Apache ORC [7].

A compressed Snappy file consists of tokens, where a token contains the original data itself (literal token) or a back reference to previously written data (copy token). Even with a large and fast FPGA fabric, decompression throughput is degraded by stalls introduced by *read-after-write* (RAW) data dependencies. When processing tokens in a pipeline, copy tokens may need to stall and wait until the prior data is valid.

In this paper, we propose two techniques to achieve efficient high single-decompressor throughput by keeping only a single BRAM-banked copy of the history data and operating on each BRAM independently. A first stage efficiently refines the tokens into commands that operate on a single BRAM and steers the commands to the appropriate one. In the second stage, rather than spending a lot of logic on calculating the dependencies and scheduling operations, a recycle method is used where each BRAM command executes immediately and those that return with invalid data are recycled to avoid stalls caused by the RAW dependency. We apply these techniques to Snappy [15] decompression and implement a Snappy decompression accelerator on a CAPI2-attached FPGA platform equipped with a Xilinx VU3P FPGA. Experimental results show that our proposed method achieves up to 7.2 GB/s throughput per decompressor, with each decompressor using 14.2% of the logic and 7% of the BRAM resources of the device.

Compared to a state-of-the-art single-core software implementation [1] on the Power9 CPU, the proposed method is about an order of magnitude faster and an order of magnitude more power efficient. This paper extends our previous study [10, 11], implementing an instance with multiple decompressors and evaluating its performance. A VU3P FPGA can contain up to five such decompressors. An instance with two decompressors activated can saturate CAPI2 bandwidth, while a fully active instance with five decompressors can keep pace with the bandwidth of OpenCAPI.

Specifically, this paper makes the following contributions.

- We present a refine method to increase decompression parallelism by breaking tokens into BRAM commands that operate independently.
- We propose a recycle method to reduce the stalls caused by the intrinsic data dependencies in the compressed file.
- We apply these techniques to develop a Snappy decompressor that can process multiple tokens per cycle.
- We evaluate end-to-end performance. An engine containing one decompressor can achieve up to 7.2 GB/s throughput. A instance with two decompressors can saturate CAPI2 bandwidth, while an instance containing five compressors can keep pace with the OpenCAPI bandwidth.
- We implement multi-engine instances with different numbers of decompressors and evaluate and compare their performance. The performance of the multi-engine instance is proportional to the number of engines until it reaches the the bound of the interface bandwidth.
- We discuss and compare a strong decompressor that can process multiple tokens per cycle to multiple light decompressors that each can only process one token per cycle.

2 Snappy (De)compression Algorithm

Snappy is an LZ77-based [31] byte-level (de)compression algorithm widely used in big data systems, especially in the Hadoop ecosystem, and is supported by big data formats such as Parquet [8] and ORC [7]. Snappy works with a fixed uncompressed block size (64KB) without any delimiters to imply the block boundary. Thus, a compressor can easily partition the data into blocks and compress them in parallel, but achieving concurrency in the decompressor is difficult because block boundaries are not known due to the variable compressed block size. Because the 64kB blocks are individually compressed, there is a fixed (64kB) history buffer during decompression, unlike the sliding history buffers used in LZ77, for example. Similar to the LZ77 compression algorithm, the Snappy compression algorithm reads the incoming data and compares it with the previous input. If a sequence of repeated bytes is found, Snappy uses a (length, offset) tuple, *copy* token, to replace this repeated sequence. The length indicates the length of the repeated sequence, while the offset is the distance from the current position back to the start of the repeated sequence, limited to the 64kB block size. For those sequences not found in the

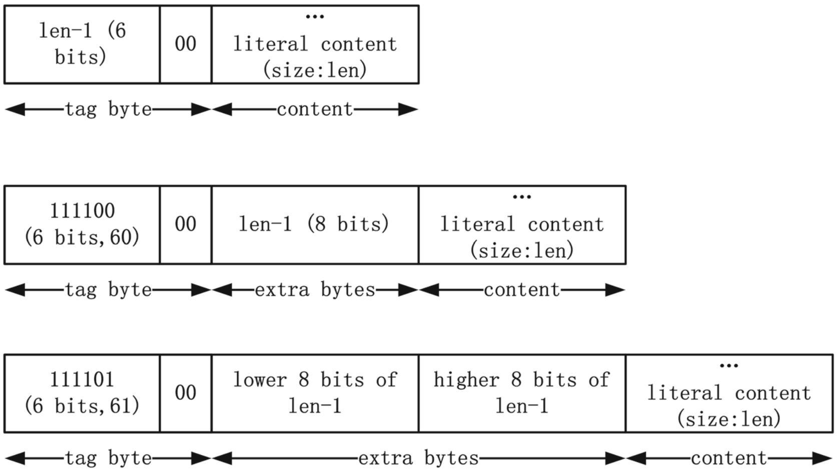


Figure 1 Format of Snappy literal tokens.

history, Snappy records the original data in another type of token, the $literal$ tokens.

The details of the Snappy token format are shown in Figs. 1 and 2. Snappy supports different sizes of tokens, including 1 to 5 bytes for literal tokens, excluding the data itself, and 2 to 5 bytes for copy tokens[1]. The size and the type of a token can be decoded from the first byte of the token, also known as the tag byte, which contains token information including the type of the token and the size of the token. Other information such as the length of the literal content, the length and the offset of the copy content is also stored in the tag byte, but they can be stored in the next two bytes if necessary.

Snappy decompression is the reverse process of the compression. It translates a stream with literal tokens and copy tokens into uncompressed data. Even though Snappy decompression is less computationally intensive than Snappy compression, the internal dependency limits the decompression parallelism. To the best of our knowledge, the highest Snappy decompression throughput is reported in [3] using the “lzbench” [1] benchmark, where the throughput reaches 1.8GB/s on a Core i7-6700K CPU running at 4.0GHz. Table 1 shows the pseudo code of the Snappy decompression, which can also be applied to other LZ-based decompression algorithms.

The first step is to parse the input stream (variable ptr) into tokens (Line 4 & 5). During this step, as shown in Line 4 of Table 1, the tag byte (the first byte of a token) is read and parsed to obtain the information of the token, e.g. the token type ($type$), the length of the literal string (lit_len), the length of copy string ($copy\_len$), and the length of extra bytes of this token ($extra\_len$). Since the token length varies and might be larger than one byte, if the token requires extra bytes (length indicated by $extra\_len$ in Line 5) to store the information, it needs to read and parse these bytes to extract and update the token information. For a literal token, as it contains the uncompressed data that can be read directly from the token, the uncompressed data is extracted and added to the history buffer (Line 11). For a copy token, the repeated sequence can be read according to the offset (variable $copy\_offset$) and the length (variable $copy\_len$), after which the data will be updated to the tail of the history (Line 7 & 8). When a block is decompressed (Line 3), the decompressor outputs the history buffer and resets it (Line 13 & 2) for the decompression of the next block.

There are three data dependencies during decompression. The first dependency occurs when locating the block boundary (Line 3). As the size of a compressed block is a variable, a block boundary cannot be located until the previous block has been decompressed, which brings challenges to leverage the block-level parallelism. The second dependency occurs during the generation of the token (Line 4 & 5). A Snappy compressed file typically contains different sizes of tokens, where the size of a token can be decoded from the first byte of this token (known as the tag byte), exclusive the literal content. Consequently, a token boundary cannot be recognized until the previous token is decoded, which prevents the parallel execution of multiple tokens. The third dependency is the RAW data dependency between the reads from the copy token and the writes from all tokens (between Line 7 and Line 8 & 11). During the execution of a copy token, it first reads the repeated sequence from the history buffer that might be not valid yet if multiple tokens are processed in parallel. In this case, the execution of this copy token needs to be stalled and wait until the request data is valid. In this paper, we focus on the latter two dependencies, and the solutions to reduce the impact of these dependencies are explained in Section 5.3 (second dependency) and Section 4.2 (third dependency).

[1] Snappy 1.1 supports 1 to 3 bytes literal tokens and 2 to 3 bytes copy tokens

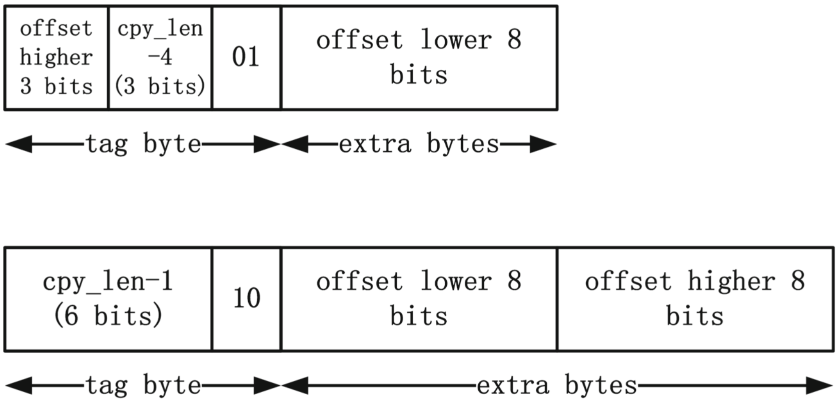


Figure 2 Format of Snappy copy tokens.

Snappy compression works on a 64KB block level. A file larger than 64KB will be divided into several 64KB blocks and compressed independently. Each 64KB block uses its own history. After that, the compressed blocks join together constructing a complete compressed file. Similarly, Snappy decompression follows the 64KB block granularity. However, the length of the compressed blocks differ block-by-block, and Snappy files do not record the boundary between different compressed blocks. Thus, to calculate the block boundaries, either additional computation is required or 64KB data blocks must be decompressed sequentially. This paper does not address the block boundary issue but instead focuses on providing fast and efficient decompression of a sequence of compressed blocks.

3 Related Work

Although compression tasks are more computationally intensive than the decompression, the decompressor is more difficult to parallelize than the compressor. Typically, the compressor can split the data into blocks and compress these blocks independently in different cores or processors to increase the parallelism and throughput, while the decompressor can not easily leverage this block-level parallelism. This is because the compressed block size is variable, and the block boundary is difficult to locate before the previous block is decompressed. Such challenges can be observed from some parallel version of compression algorithms such as pbzip2 [14], where the compression can have near-linear speedup to the single core throughput, but the decompression cannot.

Table 1 Procedure of Snappy decompression.

```
while(!eof) {
  reset(&history);
  while(!end_of_block()){
    read_tag_byte(&ptr, &type, &extra_len, &lit_len, &
         copy_len);
    read_extra_bytes(&ptr, extra_len, &lit_len, &
         copy_offset);
    if(type==copy){
      read_history(history, copy_offset, copy_len, &
         buffer);
      update_history_c(&history, buffer, copy_len);
    }
    else // type==literal
      update_history_l(&history, &ptr, lit_len);
  }
  output(history);
}
```

Many recent studies consider improving the speed of lossless decompression. The study in [12] discusses some of the prior work in the context of databases. To address block boundary problems, [19] explores the block-level parallelism by performing pattern matching on the delimiters to predict the block boundaries. However, this technique cannot be applied to Snappy decompression since Snappy does not use any delimiters but uses a fixed uncompressed block size (64KB) as the boundary. Another way to utilize the block-level parallelism is to add some constraints during the compression, e.g adding padding to make fixed size compressed blocks [4] or add some meta data to indicate the boundary of the blocks [26]. A drawback of these methods is that it is only applicable to the modified compression algorithms (add padding) or even not compatible to the original (de)compression algorithms (add meta data).

The idea of using FPGAs to accelerate decompression has been studied for years. On the one hand, FPGAs provide a high-degree of parallelism by adopting techniques such as task-level parallelization, data-level parallelization, and pipelining. On the other hand, the parallel array structure in an FPGA offers tremendous internal memory bandwidth. One approach is to pipeline the design and separate the token parsing and token execution stages [17, 20, 29, 30]. However, these methods only decompress one token each FPGA cycle, limiting throughput.

Other works study the possibility of processing multiple tokens in parallel. [23] proposes a parallel LZ4 decompression engine that has separate hardware paths for literal tokens and copy tokens. The idea builds on the observation that the literal token is independent since it contains the original data, while the copy token relies on the previous history. A similar two-path method for LZ77-based decompression is shown in [16], where a slow-path routine is proposed to handle large literal tokens and long offset copy tokens, while a fast-path routine is adopted for the remaining cases. [5] introduces a method to decode variable length encoded data streams that allows a decoder to decode a portion of the input streams by exploring all possibilities of bit spill. The correct decoded streams among all the possibilities are selected as long as the bit spill is calculated and the previous portion is correctly decoded. [25] proposes a token-level parallel Snappy decompressor that can process two tokens every cycle. It uses a similar method as [5] to parse an eight-byte input into tokens in an earlier stage, while in the later stages, a conflict detector is adopted to detect the type of conflict between two adjacent tokens and only allow those two tokens without conflict to be processed in parallel. However, these works cannot easily scale up to process more tokens in parallel because it requires very complex control logic and duplication of BRAM resources to handle the BRAM bank conflicts and data dependencies.

The GPU solution proposed in [26] provides a multi-round resolution method to handle the data dependencies. In each round, all the tokens with read data valid are executed, while those with data invalid will be pending and wait for the next round of execution. This method allows out-of-order execution and does not stall when a request needs to read the invalid data. However, this method requires specific arrangement of the tokens, and thus requires modification of the compression algorithm.

This paper presents a new FPGA decompressor architecture that can process multiple tokens in parallel and operate at a high clock frequency without duplicating the history buffers. It adopts a refine and recycle method to reduce the impact of the BRAM conflicts and data dependencies, and increases the decompression parallelism, while conforming to the Snappy standard. This paper extends our previous work [11] with multi-engine results and presents more extensive explanations of the architecture and implementation, and elaborates the evaluation and discussion.

4 Refine and Recycle Technique

4.1 The Refine Technique for BRAM Bank Conflict

Typically, in FPGAs, the large history buffers (e.g. 32KB in GZIP and 64KB in Snappy) can be implemented using BRAMs. Taking Snappy as an example, as shown in Fig. 3, to construct a 64KB history buffer, a minimum number of BRAMs are required: 16 4KB blocks for the Xilinx Ultrascale Architecture [21]. These 16 BRAMs can be configured to read/write independently, so that more parallelism can be achieved. However, due to the structure of BRAMs, a BRAM block supports limited parallel reads or writes, e.g. one read and one write in the simple dual port configuration. Thus, if more than one read or more than one write need to access different lines in the same BRAM, a conflict occurs (e.g. conflict on bank 2 between read $R1$ and read $R2$ in Fig. 3). We call this conflict a *BRAM bank conflict* (BBC).

For Snappy specifically, the maximum literal length for a literal token and the maximum copy length for copy tokens in the current Snappy version is 64B. As the BRAM can only be configured to a maximum 8B width, there is a significant possibility that a BBC occurs when processing two tokens in the same cycle, and processing more tokens in parallel further increases the probability of a BBC. The analysis from [25] shows that if two tokens are processed in parallel, more than 30% of the reads from adjacent tokens have conflicts. This number might increase when more tokens are required to be processed in parallel. A naive way to deal with the BBCs is to only process one of the conflicting tokens and stall the others until the this token completes. For example, in Fig. 3, when a read request from a copy token ($R1$) has a BBC with another read request

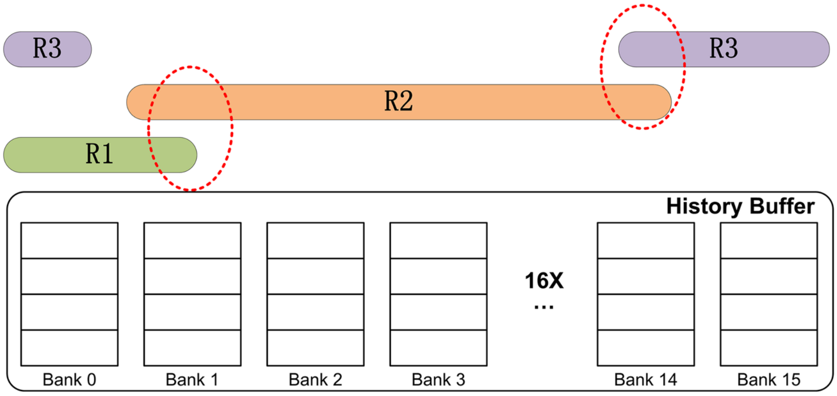


Figure 3 An example of BRAM bank conflicts in Snappy (request processing in the token level).

from another copy token ($R2$), the execution of $R2$ stalls and waits until $R1$ is finished, so does the case between ($R2$) and ($R3$). Obviously, this method sacrifices some parallelism and even leads to a degradation from parallel processing to sequential processing. Duplicating the history buffers can also relieve the impact of BBCs. The previous work [25] uses a double set of history buffers, where two parallel reads are assigned to different set of history. So, the two reads from the two tokens never have BBCs. However, this method only solves the read BBCs but not the write BBCs, since the writes need to update both sets of history to maintain the data consistency. Moreover, to scale this method to process more tokens in parallel, additional sets (linearly proportional to the number of tokens being processed in parallel) of BRAMs are required.

To reduce the impact of BBCs, we present a refine method to increase token execution parallelism without duplicating the history buffers. As demonstrated in Fig. 4, the idea is to break the execution of tokens into finer-grain operations, the BRAM copy/write commands, and for each BRAM to execute its own reads and writes independently. The read requests $R1$ and $R2$ in Fig. 3 only have a BBC in bank 2, while the other parts of these two reads do not conflict. We refine the token into BRAM commands operating on each bank independently. As a result, for the reads in the non-conflicting banks of $R2$ (bank 0 & 1), we allow the execution of the reads on these banks from $R2$. For the conflicting bank 2, $R1$ and $R2$ cannot be processed concurrently.

The implementation of this method requires extra logic to refine the tokens into independent BRAM commands (the BRAM command parser presented in Section 5.4) and distribute these commands to the proper BRAM, which costs extra FPGA resource, especially LUTs.

The proposed method takes advantage of the parallelism of the array structure in FPGAs by operating at a finer-grained level, the single BRAM read/write level, compared with the token-level. It supports partially executing multiple tokens in parallel even when these tokens have BBCs. In the extreme case, the proposed method can achieve up to 16 BRAM operations in parallel, meaning generating the decompressed blocks at a speed of 128B per cycle. This refine method can also reduce the read-after-write dependency impact mentioned in Section 4.2. If the read data of a read request from a copy token is partially valid, this method allows this copy token to only read the valid data and update the corresponding part of the history, instead of waiting until all the bytes are valid. This method can obtain high degree of parallelism without using resource-intensive dependency checking mechanisms such as “scoreboarding”.

4.2 The Recycle Technique for RAW Dependency

The *Read-After-Write* (RAW) dependency between data reads and writes on the history buffer is another challenge for parallelization. If a read needs to fetch data from some memory address that the data has not yet been written to, a hazard occurs, and thus this read needs to wait until the data is written. A straightforward solution [25] is to execute the tokens sequentially and perform detection to decide whether the tokens can be processed in parallel. If a RAW hazard is detected between two tokens that are being processed in the same cycle, it forces the latter token to stall until the previous token is processed. Even though we can apply the forwarding technique to reduce the stall penalty, detecting multiple tokens and forwarding the data to the correct position requires complicated control logic and significant hardware resource.

Another solution is to allow out-of-order execution. That is when a RAW hazard occurs between two tokens, subsequent tokens are allowed to be executed without waiting, similar to out-of-order execution in the CPU architecture. Fortunately, in the decompression case, this does not require a complex textbook solution such “Tomasulo” or “Scoreboarding” to store the state of the pending tokens.

Instead, rerunning pending tokens after the execution of all or some of the remaining tokens guarantees the correctness of this out-of-order execution. This is because there is no write-after-write or write-after-read dependency during the decompression, or two different writes never write the same place and the write data never changes after the data is read. So, there is no need to record the write data

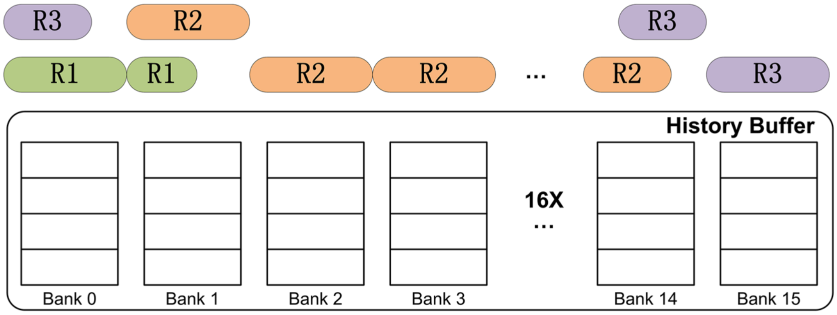


Figure 4 Request processing with the refine method.

states, and thus a simpler out-of-order execution model can satisfy the requirement, which saves logic resources.

In this paper, we present the recycle method to reduce the impact of RAW dependency at a BRAM command granularity. Specifically, when a command needs to read the history data that may not be not valid yet, the decompressor executes this command immediately without checking if all the data is valid. If the data that has been read is detected to be not entirely valid, this command (invalid data part) should be recycled and stored in a recycle buffer, where it will be executed again (likely after a few other commands are executed). If the data is still invalid in the next execution, this decompressor performs this recycle-and-execute procedure repeatedly until the read data is valid.

Thus this method executes the commands in a relaxed model and allows continuous execution on the commands without stalling the pipeline in almost all cases. The method provides more parallelism since it does not need to be restricted to the degree of parallelism calculated by dependency detection. The implementation of the recycle buffers needs extra memory resources and a few LUTs on an FPGA (see Table 3). Details of the design are presented in Section 5.5

5 Snappy Decompressor Architecture

5.1 Architecture Overview

Figure 5 presents an overview of the proposed architecture. It can be divided into two stages. The first stage parses the input stream lines into tokens and refines these tokens into BRAM commands that will be executed in the second stage. It contains a slice parser to locate the boundary of the tokens, multiple BRAM command parsers (BCPs) to refine the tokens into BRAM commands, and an arbiter to drive the output of the slice parser to one of the BCPs. In the second stage, the BRAM commands are executed to generate the decompressed data under the recycle method. The execution modules, in total 16 of them, are the main components in this stage, in which recycle buffers

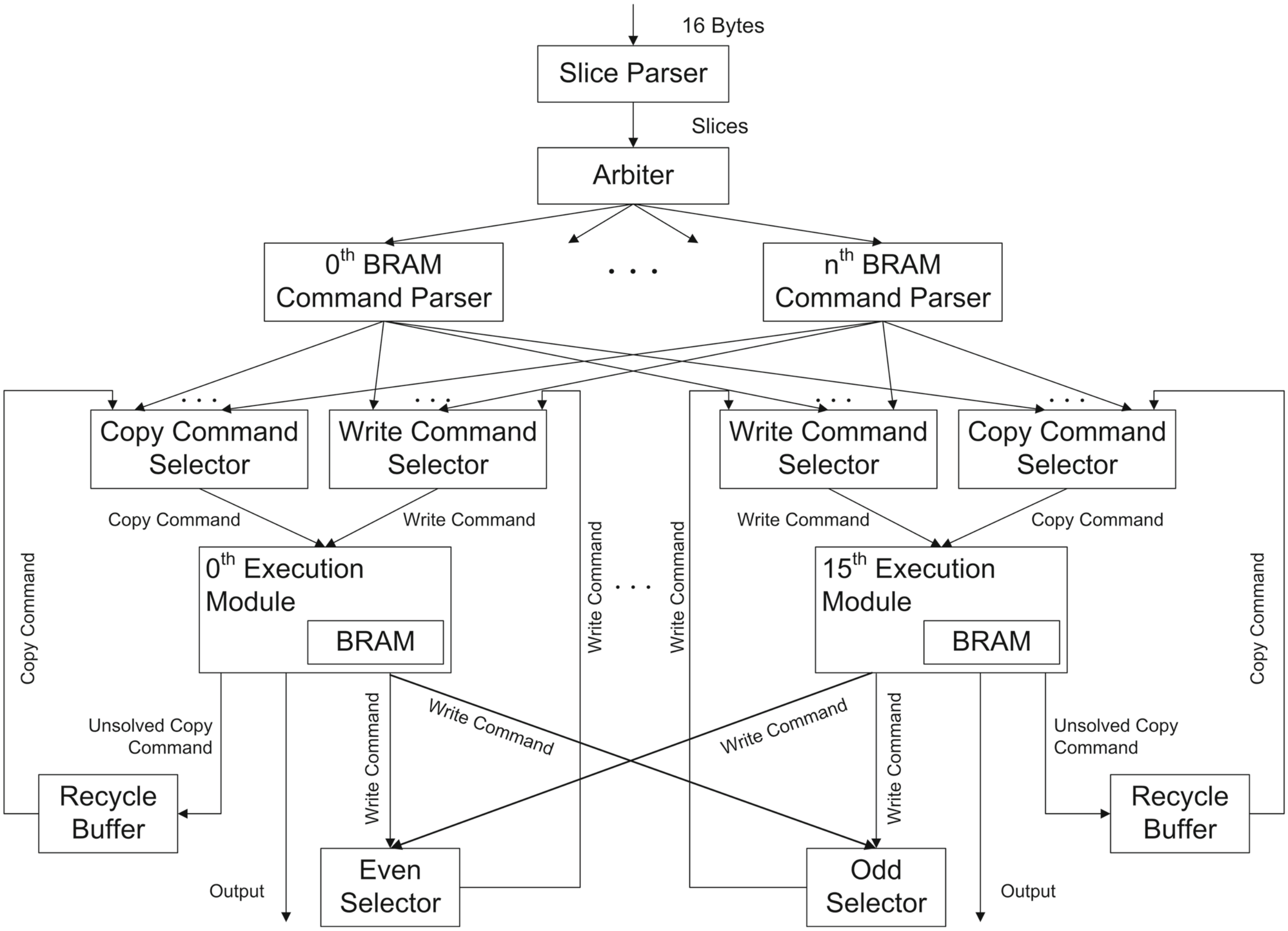


Figure 5 Architecture overview.

are utilized to perform the recycle mechanism. Since the history buffer requires at least 16 BRAMs, we use 16 execution modules to activate all 16 BRAMs to obtain high parallelism. Increasing (e.g. doubling) the number of BRAMs and the number of execution modules might further increase the parallelsim at the expense of more BRAMs. However, due to the data dependencies, using twice the number of execution modules does not bring obvious performance improvement.

As shown in Fig. 6, the procedure starts with receiving a 16B input line into the slice parser together with the first 2B of the next input line (required because a token is 1,2, or 3 bytes excluding the literal content). This 18B is parsed into a "slice" that contains token boundary information including which byte is a starting byte of a token, whether any of the first 2B have been parsed in the previous slice, and whether this slice starts with literal content, etc. After that, an arbiter is used to distribute each slice to one of the BCPs that work independently, and there the slice is split into one or multiple BRAM commands.

There are two types of BRAM commands, write commands and copy commands. The write command is

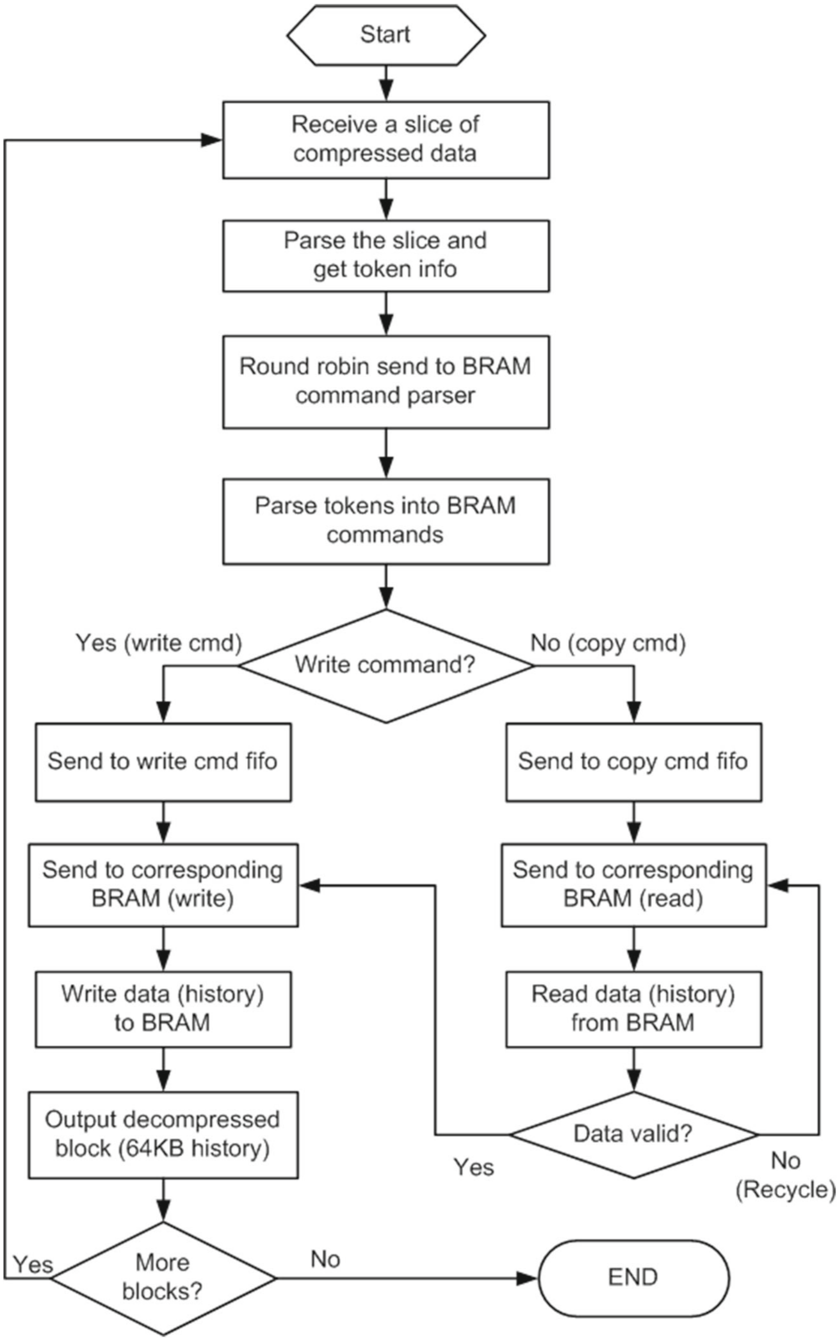

Start
Receive a slice of compressed data
Parse the slice and get token info
Round robin send to BRAM command parser
Parse tokens into BRAM commands
Yes (write cmd)
Write command?
No (copy cmd)
Send to write cmd fifo
Send to copy cmd fifo
Send to corresponding BRAM (write)
Send to corresponding BRAM (read)
Write data (history) to BRAM
Read data (history) from BRAM
Output decompressed block (64KB history)
Data valid?
Yes
No (Recycle)
Yes
More blocks?
No
END

Figure 6 Proposed Snappy Decompression Procedure.

generated from the literal token, indicating a data write operation on the BRAM, while the copy command is produced from the copy token which leads to a read operation and a follow-up step to generate one or two write commands to write the data in the appropriate BRAM blocks.

In the next stage, write selectors and copy selectors are used to steer the BRAM commands to the appropriate execution module. Once the execution module receives a write command and/or a copy command, it executes the command and performs BRAM read/write operations. As the BRAM can perform both a read and a write in the same cycle, each execution module can simultaneously process a write command and one copy command (only the read operation) at the same time. The write command will always be completed successfully once the execution module receives it, which is not the case for the copy command. After performing the read operation of the copy command, the execution module runs two optional extra tasks according to the read data, including generating new write/copy commands and recycling the copy command.

If some bytes are invalid, the copy command will be renewed (removing the completed portion from the command) and collected by a recycle unit, and sent back for the next round of execution.

If the read data contains at least one valid byte, new write commands are generated to write this data to its destination. It is possible that it generates one or two new write commands. If the writing back address of read data crosses the boundary of the BRAMs, it should be written to two adjacent BRAMs, thus generating two write commands.

To save resources for multiplexers, we classify all BRAMs into two sets: even and odd. The even set consists of even numbered BRAMs, namely 0^{th}, 2^{nd}, 4^{th} and so on. The remaining BRAMs belong to the odd set. When two write commands are generated in one cycle, their writing targets are always one odd BRAM and one even BRAM. Therefore, we can first sort them once they are generated, and then do selection only within each set.

Once a 64KB history is built, this 64KB data is output as the decompressed data block. After that, a new data block is read, and this procedure will be repeated until all the data blocks are decompressed.

5.2 History Buffer Organization

The 64KB history buffer consists of 16 4KB BRAM blocks, using the FPGA 36Kb BRAM primitives in the Xilinx UltraScale fabric. Each BRAM block is configured to have one read port and one write port, with a line width of 72bits (8B data and 8bits flags). Each bit from the 8bits flags indicates whether the corresponding byte is valid. To access a BRAM line, 4 bits of BRAM bank address, and 9 bits of BRAM line address is required. The history data is stored in these BRAMs in a striped manner to balance the BRAM read/write command workload and to enhance parallelism.

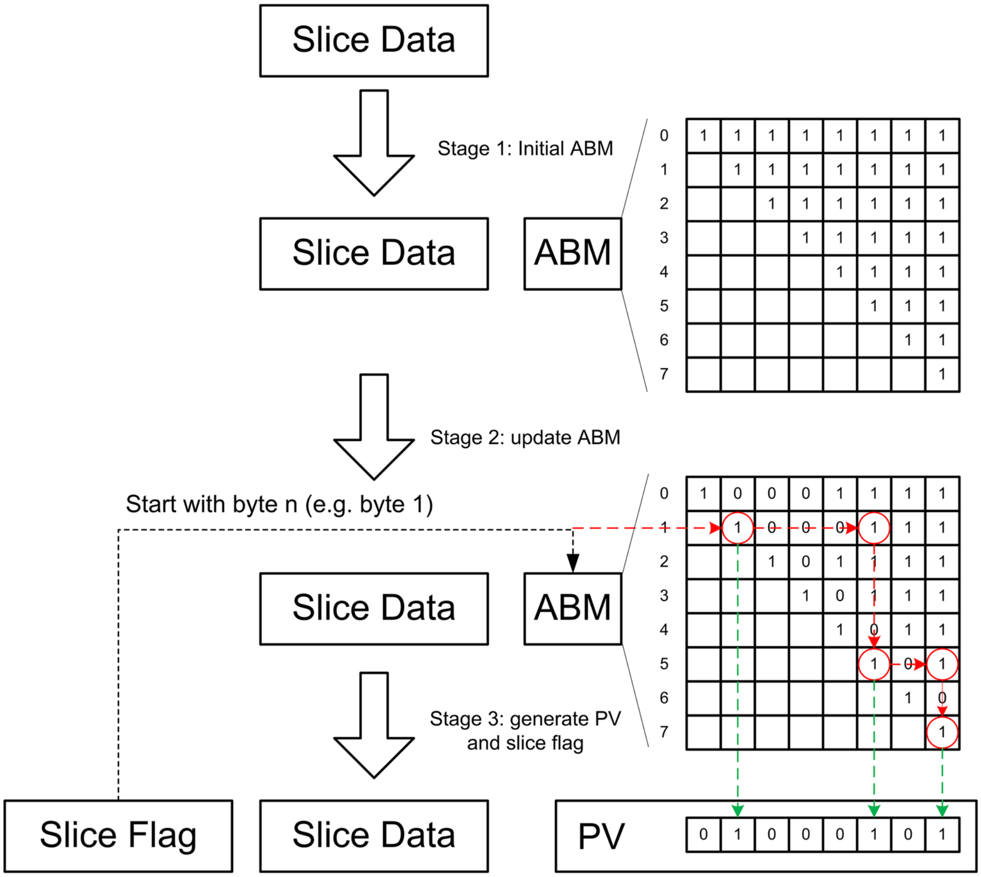
5.3 Slice Parser

The slice parser aims to decode the input data lines into tokens in parallel. Due to the variety of token sizes, the starting byte of a token needs to be calculated from the previous token. This data dependency presents an obstacle for the parallelization of the parsing process. To solve this problem, we assume all 16 input bytes are starting bytes, and parse this input data line based on this assumption. The correct branch will be chosen once the first token is recognized. To achieve a high frequency for the implementation, we propose a bit map based byte-split detection algorithm by taking advantage of bit-level control in FPGA designs.

A bit map is used to represent the assumption of starting bytes, which is called the Assumption Bit Map (ABM) in the remainder of this paper. For a N bytes input data line, we need a $N * N$ ABM. As shown in Fig 7, taking an 8B input data line as an example, cell(i, j) being equal to '1' in the ABM means that if corresponding byte i is a starting byte of one token, byte j is also a possible starting byte. If a cell has a value '0', it means if byte i is a starting byte, byte j cannot be a starting byte.

This algorithm has three stages. In the first stage, an ABM is initialized with all cells set to '1'. In the second stage, based on the assumption, each row in the ABM is updated in parallel. For row i, if the size of the token starts with the assumption byte is L, the following $L - 1$ bits are set to be 0. The final stage merges the whole ABM along with the slice flag from the previous slice, and calculate a Position Vector (PV). The PV is generated by following a cascading chain. First of all, the slice flag from the previous slice points out which is the starting byte of the first token in the current slice (e.g. byte 1 in Fig. 7). Then the corresponding row in the ABM is used to find the first byte of the next token (byte 5 in Fig. 7), and its row in the ABM is used for finding the next token. This procedure is repeated (all within a single FPGA cycle) until all the tokens in this slice are found. The PV is an N-bit vector that its ith bit equal to '1' means the ith byte in the current slice is a starting byte of a token. Meanwhile, the slice flag will be updated. In addition to the starting byte position of the first token in the next slice, the slice flag contains other informations such as whether the next slice starts with literal content, the unprocessed length of the literal content, etc.

Figure 7 Procedure of the slice parser and structure of the Assumption Bit Map.



0	1	1	1	1	1	1	1	1
1		1	1	1	1	1	1	1
2			1	1	1	1	1	1
3				1	1	1	1	1
4					1	1	1	1
5						1	1	1
6							1	1
7								1

0	1	0	0	0	1	1	1	1
1		1	0	0	0	1	1	1
2			1	0	1	1	1	1
3				1	0	1	1	1
4					1	0	1	1
5						1	0	1
6							1	0
7								1

0	1	0	0	0	1	0	1

5.4 BRAM Command Parser

The BRAM command parser refines the tokens and generates BRAM commands based on the parsed slice. The structure of the BCP is demonstrated in Fig. 8. The first step is to generate tokens based on the token boundary information that is stored in the PV. Literal tokens and copy tokens output from the token generator are assigned to different paths for further refining in the BRAM command generator. In the literal token path, the BRAM command generator calculates the token write address and length, and splits this write operation into multiple ones to map the write address to the BRAM address. Within a slice, the maximum length of the literal token is 16B (i.e. the largest write is 16B), which can generate up to 3 BRAM write commands where each write command has a maximum width of 8B. In the copy token path, the BRAM command generator performs a similar split operation but maps both the read address and the write address to the BRAM address. A copy token can copy up to 64B data. Hence, it generates up to 9 BRAM copy commands.

Since multiple commands are generated each cycle, to prevent stalling the pipeline, we use multiple sets of FIFOs to store them before sending them to the corresponding execution module. Specifically, 4 FIFOs are used to store the literal commands which is enough to store all 3 BRAM write commands generated in one cycle. Similarly, 16 copy command FIFOs are used to handle the maximum 9 BRAM copy commands. To keep up with the input stream rate (16B per cycle), multiple BCPs can work in parallel to enhance the parsing throughput.

5.5 Execution Module

The execution module performs BRAM command execution and the recycle mechanism. Its structure is illustrated in Fig. 9. It receives up to 1 write command from the write command selector and 1 copy command from the copy command selector. Since each BRAM has one independent read port and one independent write port, each BRAM can process one read command and one copy command each clock cycle. For the write command, the write control logic

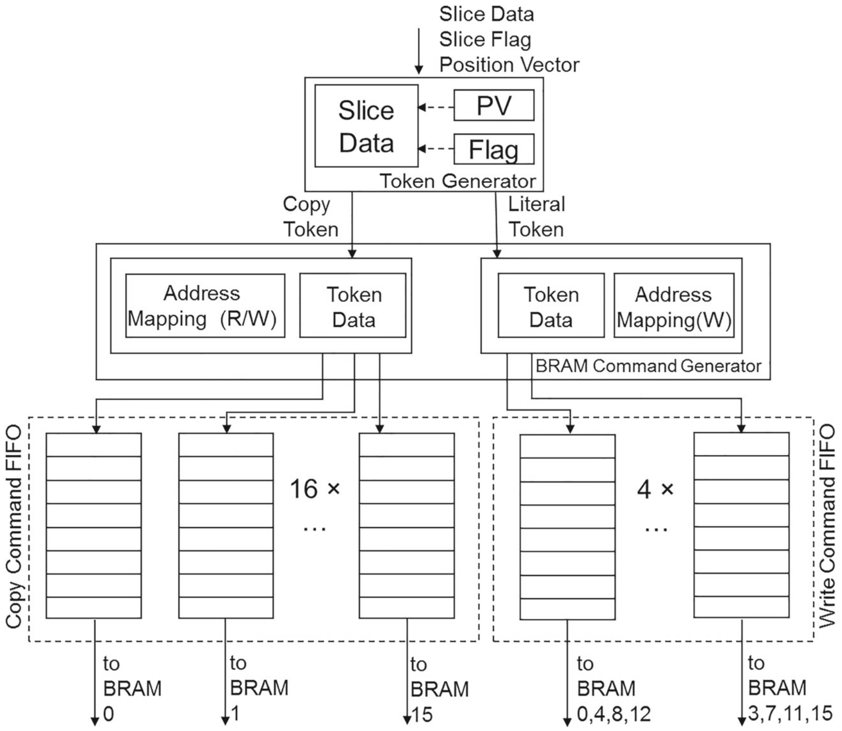


Figure 8 Structure of BRAM command parser.

extracts the write address from the write command and performs a BRAM write operation. Similarly, the read control logic extracts the read address from the read command and performs a BRAM read operation.

While the write command can always be processed successfully, the copy command can fail when the target data is not ready in the BRAM. So, there should be a recycle mechanism for failed copy commands. After reading the data, the unsolved control logic checks whether the read data is valid. There are three different kinds of results: 1) all the target data is ready (hit); 2) only part of the target data are ready (partial hit); 3) none of the target data is ready (miss). In the hit case and the partial hit case, the new command generator produces one or two write commands to write the copy results to one or two BRAMs, depending on the alignment of the write data. In the partial hit case and the miss case, a new copy command is generated and recycled, waiting for the next round of execution.

5.6 Selector Selection Strategy

The BRAM write commands and copy commands are placed in separate paths, and can work in parallel. The Write Command Selector gives priority to recycled write commands. Priority is next given to write commands from one of the BCPs using a round robin method. The Copy Command Selector gives priority to the copy commands from one of the BCPs when there is a small number of copy commands residing in the recycle FIFO. However, when this number reaches a threshold, priority will be given back to the recycle commands. This way, it not only provides enough commands to be issued and executed, but also guarantees the recycle FIFO does not overflow, and no deadlock occurs.

6 Experimental Results

6.1 Experimental Setup

To evaluate the proposed design, an implementation is created targeting the Xilinx Virtex Ultrascale VU3P-2 device on an AlphaData ADM-PCIE-9V3 board and integrated with the POWER9 CAPI 2.0 [27] interface. The CAPI 2.0 interface on this card supports the CAPI protocol at an effective data rate of approximately 13 GB/s. The FPGA design is compared with an optimized software Snappy decompression implementation [1] compiled by gcc 7.3.0 with "O3" option and running on a POWER9 CPU in little endian mode with Ubuntu 18.04.1 LTS.

We test our Snappy decompressor for functionality and performance on 6 different data sets. The features of the data

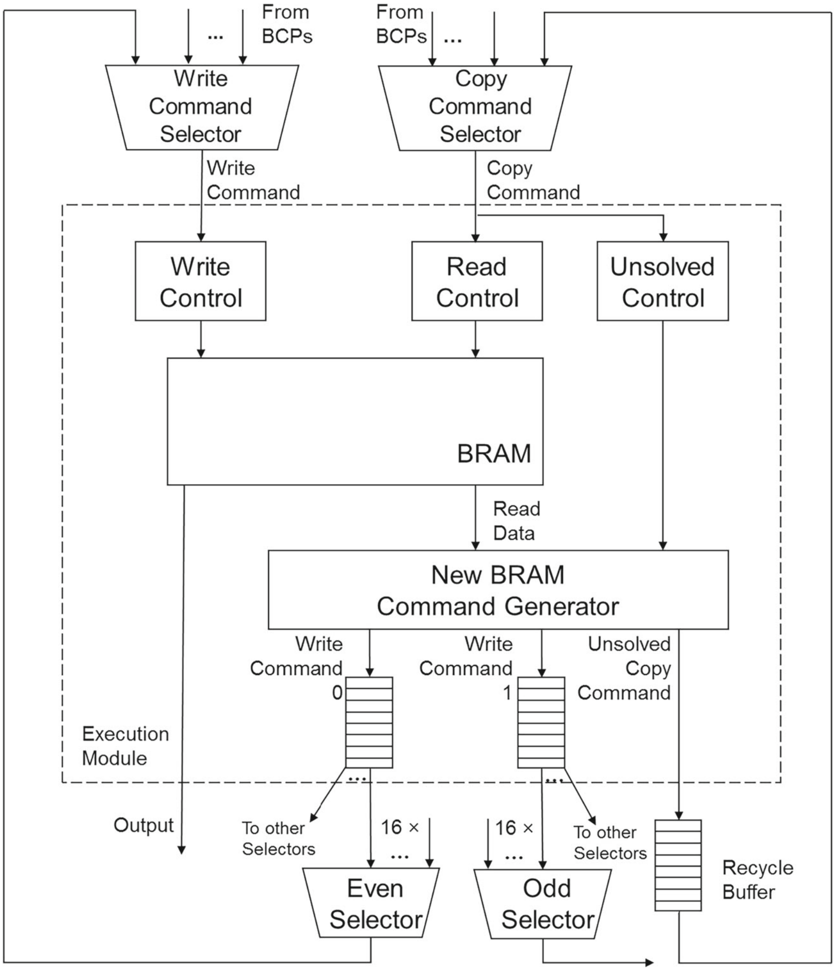


Figure 9 Structure of execution module.

sets are listed in Table 2. The first three data sets are from the “lineitem” table of the TPC-H benchmarks in the database domain. We use the whole table (Table) and two different columns including a long integer column (Integer) and a string column (String). The data set Wiki [22] is an XML file dump from Wikipedia, while the Matrix is a sparse matrix from the Matrix Market [2]. We also use a very high compression ratio file (Geo) that stores geographic information .

6.2 System Integration

The proposed Snappy decompressor communicates with host memory through the CAPI 2.0 interface. The Power Service Layer (PSL) in the CAPI 2.0 support conversion between the CAPI protocol and the AXI protocol, and thus the Snappy decompressor can talk to the host memory using the AXI bus. To activate full CAPI 2.0 bandwidth, the AXI bus should be configured in 512 bits or 64 bytes width. If

Table 2 Benchmarks used and throughput results.

Files	Original	Compression	Throughput (GB/s)		Speedup
	size (MB)	ratio	CPU	FPGA	
Integer	45.8	1.70	0.59	4.40	7.46
String	157.4	2.45	0.69	6.02	8.70
Table	724.7	2.07	0.59	6.11	10.35
Matrix	771.3	2.75	0.80	4.80	6.00
Wiki	953.7	1.97	0.56	5.72	10.21
Geo	128.0	5.50	1.41	7.21	5.11

Table 3 Resource utilization of design components.

Resource	LUTs	BRAMs[1]	Flip-Flops
Recycle buffer	1.1K(0.3%)	8(1.2%)	1K(0.1%)
Decompressor(incl. recycle buffer)	56K(14.2%)	50(7.0%)	37K(4.7%)
CAPI2 interface	82K(20.8%)	238(33.0%)	79K(10.0%)
Total	138K(35.0%)	288(40.0%)	116K(14.7%)

[1] One 18kb BRAM is counted as a half of one 36kb BRAM

an instance with multiple decompressors is deployed, an IO controller is needed and placed between the decompressors and the CAPI 2.0 interface to handle the input data distribution to the correct decompressor, as well as output data collection from the corresponding decompressor.

6.3 Resource Utilization

Table 3 lists the resource utilization of our design timing at 250MHz. The decompressor configured with 6 BCPs and 16 execution module takes around 14.2% of the LUTs, 7% of the BRAMs, 4.7% of the Flip-Flops in the VU3P FPGA. The recycle buffers, the components that are used to support out-of-order execution, only take 0.3% of the LUTs and 1.2% of the BRAMs. The CAPI 2.0 interface logic implementation takes up around 20.8% of the LUTs and 33% of the BRAMs. Multi-unit designs can share the CAPI 2.0 interface logic between all the decompressors, and thus the (VU3P) device can support up to 5 engines.

6.4 End-to-end Throughput Performance

We measure the end-to-end decompression throughput reading and writing from host memory. We compare our design with the software implementation running on one POWER9 CPU core (remember that parallelizing Snappy decompression is difficult due to unknown block boundaries).

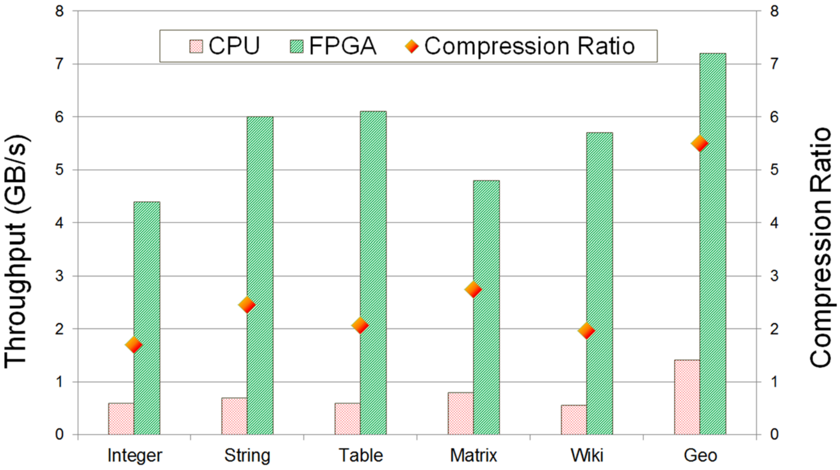
Figure 10 shows the end-to-end throughput performance of the proposed architecture configured with 6 BCPs. The proposed Snappy decompressor reaches up to 7.2 GB/s output throughput or 31 bytes per cycle for the file (Geo) with high compression ratio, while for the database application (Table) and web application (Wiki) it achieves 6.1 GB/s and 5.7 GB/s, which is 10 times faster than the software implementation. One decompressor can easily keep pace with a (Gen3 PCIe x4) NVMe device, and the throughput of an implementation containing two of such engines can reach the CAPI 2.0 bandwidth upper bound.

Regarding the power efficiency, the 22-core POWER9 CPU is running under 190 watts, and thus it can provide up to 0.16GB/s per watt. However, the whole ADM 9V3 card can support 5 engines under 25 watts [6], which corresponds to up to 1.44GB/s per watt. Consequently, our Snappy decompressor is almost an order of magnitude more power efficient than the software implementation.

6.5 Impact of # of BCPs

As explained in Section 5, the number of BCPs corresponds to the number of tokens that can be refined into BRAM commands per cycle. We compare the resource utilization

Figure 10 Throughput of Snappy decompression.



and throughput of different numbers of BCPs, and present the results that are normalized by setting the resource usage and throughput of one BCP as 1 in Fig 11. Increasing from one BCP to two leads to 10% more LUT usage, but results in around 90% more throughput and no changes in BRAM usage. However, the increase of the throughput on Matrix slows down after 3 BCPs and the throughput remains stable after 5 BCPs. A similar trend can be seen in Wiki where the throughput improvement drops after 7 BCPs. This is because after increasing the number of BCPs, the bottleneck moves to the stage of parsing the input line into tokens. Generally, a 16B-input line contains 3-7 tokens depending on the compressed file, while the maximum number of tokens is 8, thus explaining the limited benefits of adding more BCPs. One way to achieve higher performance is to increase both the input-line size and the number of BCPs. However, this might bring new challenges to the resource utilization and clock frequency, and even reach the upper bound of the independent BRAM operations parallelism.

6.6 Comparison of Decompression Accelerators

We compare our design with state-of-the-art decompression accelerators in Table 4. By using 6 BCPs, a single decompressor of our design can output up to 31B per cycle at a clock frequency of 250MHz. It is around 14.5x and 3.7x faster then the prior work on ZLIB[18] and Snappy [25]. Even scaling up the other designs to the same frequency, our design is still around 10x and 2x faster, respectively. In addition, our design is much more area-efficient, measured in MB/s per 1K LUTs and MB/s per BRAM (36kb), which is 1.4x more LUT efficient than the ZLIB implementation in [18] and 2.4x more BRAM efficient than the Snappy implementation in [25].

Compared with FPGA-based Snappy decompression implementation from the commercial library, the Vitis Data Compression Library (VDCL) [28], the proposed methods

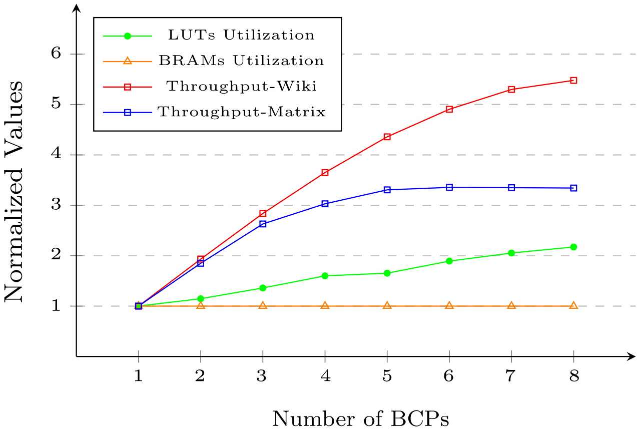


Figure 11 Impact of number of BCPs.

Table 4 FPGA decompression accelerator comparison.

Design	Frequency	Throughput		History	Area		Efficiency	
	(MHz)	GB/s	bytes/cycle	Size(KB)	LUTs	BRAMs	MB/s per 1K LUT	MB/s per BRAM
ZLIB (CAST) [18]	165	0.495	3.2	32	5.4K	10.5	93.9[1]	48.3
Snappy[25]	140	1.96	15	64	91K	32	22	62.7
Vitis-Snappy [28]	300	0.283	1	64	0.81K	16	358	18.1
Vitis-8-engine[2][28]	300	1.8	6.4	64	30.6K	146	60.2	12.6
This Work	250	7.20	30.9	64	56K	50	131.6	147.5

[1] Please note that ZLIB is more complex than Snappy and takes more LUTs to obtain the same throughput performance in principle

[2] Vitis-8-engine is an instance of 8 Snappy engines with Data Movers

is 25x and 4x faster than a single engine implementation and an 8-engine implementation, respectively. The single engine implementation from the VDCL only cost 0.81K LUTs, meaning 2.7x more LUT-efficient than the proposed design, while the proposed design is still around 2.2x more LUT-efficient than the 8-engine implementation from VDCL. However, the proposed design requires the minimum number of BRAMs to achieve the same throughput, which means 8.1x and 11.7x more BRAM efficient than the single engine implementation and the 8-engine implementation from the VDCL.

7 Multi-Engine Evaluation

7.1 Evaluation

Multiple decompressors working in parallel decompressing multiple files can achieve throughput that can saturate the interface bandwidth. According to the throughput performance of a single decompressor, two to four of such decompressors should be sufficient to keep up with the CAPI 2.0 bandwidth depending on the compressed file. We implement multi-engine instances with the number of decompressors varying from two to four and evaluate their performance. To avoid workload unbalancing between different decompressors, in each test, we use a same input file for all the decompressors.

Table 5 illustrates the resource usage of the multi-engine instances that are configured with different numbers of the decompressor engines at 250MHz. We can see that the different types of resources, including LUTs, Flip-Flops, and BRAMs, increase linearly with the number of decompressor engines. The critical path of the design is within the decompressor slice parser. With multiple engines no non-local resources are required to build the engine. In addition, cycles can be added to get data to the engine if needed to maintain frequency. Thus, to maintain 250MHz frequency in a multi-engine instance, only a few resources are needed The design is limited by LUTs, and a VU3P FPGA can contain up to 5 engines. More engines are possible to be placed in such a device if the design can be optimized to have more balance between LUTs and other resources such as BRAMs.

Fig 12 presents the throughput of the multi-engine instances with different numbers of decompressors configured. We can see that the throughput performance of multi-engine instances increases proportionally when the number of engines change from 1 to 2, and almost linearly from 2 to 3. However, performance saturates when the engine number increases from 3 to 4, and finally remains at around 13 GB/s. This is because the CAPI 2.0 interface has an effective rate of around 13 GB/s, and the throughput of the 4-engine instance has reached this bound.

7.2 Discussion

In this section we examine some of the tradeoffs between a designs with fewer strong decompressors and one with more smaller decompressors. Strong engines generally consume more resource per unit of performance than less performant ones, but may require more logic to coordinate between the host interface and the engines. Also, as is the case for the 64kB history file in the Snappy decompressor, a minimum amount of resource may be required independent of the performance of the decompressor.

Obviously, when there is insufficient parallelism in the problem, e.g. not enough independent files to be processed, fewer, more performant engines will perform better. On the other hand, when there is sufficient concurrency, e.g. a number of files to process that is substantially larger than the number of engines, then a design that optimizes performance per unit of resource with smaller engines may well be preferred. There may also be overhead associated with task switching an engine from one file to the next, further favoring the smaller engine case.

Thus, it is difficult to tell which is the better one between the "Strong-but-fewer" solution and the "Light-but-more" solution. An reasonable answer is that it depends on the application. For example, if the application contains decompression on a large number of small compressed files, it might be better to choose the "Light-but-more" solution.

Table 5 Resource utilization of multi-engine (250MHz).

Resource	LUTs	BRAMs[1]	Flip-Flops
CAPI2	82K(20.8%)	238(33.0%)	79K(10.0%)
1-engine	138K(35.0%)	288(40.0%)	116K(14.7%)
2-engine	195K(49.4%)	338(47.0%)	153K(19.4%)
3-engine	251K(63.9%)	388(54.0%)	190K(24.1%)
4-engine	308K(78.3%)	438(61.0%)	227K(28.7%)

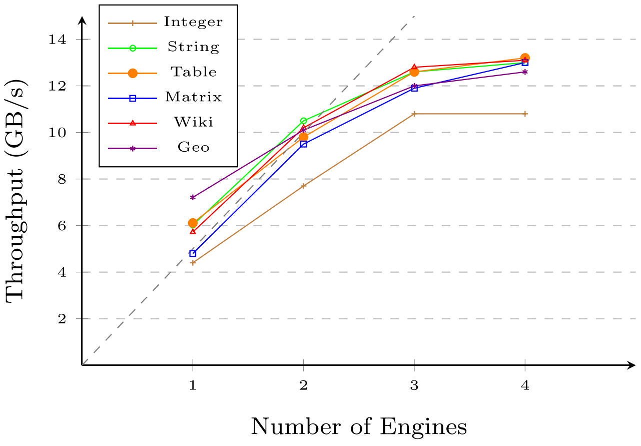


Figure 12 Throughput of multi-engine instance.

In contrast, if the application requires to decompress files with large size, the “Strong-but-fewer” solution should be selected.

When a general-purpose engine is implemented, strong engines are generally preferred, as the worst case for a strong engine (e.g. processing 100 files sequentially) is a lot better than the worst case for small engines (decompressing a single large file with just one small engine).

8 Conclusion

The control and data dependencies intrinsic in the design of a decompressor present an architectural challenge. Even in situations where it is acceptable to achieve high throughput performance by processing multiple streams, a design that processes a single token or a single input byte each cycle becomes severely BRAM limited for (de)compression protocols that require a sizable history buffer. Designs that decode multiple tokens per cycle could use the BRAMs efficiently in principle, but resolving the data dependencies leads to either very complex control logic, or to duplication of BRAM resources. Prior designs have therefore exhibited only limited concurrency or required duplication of the history buffers.

This paper presented a refine and recycle method to address this challenge and applies it to Snappy decompression to design an FPGA-based Snappy decompressor. In an earlier stage, the proposed design refines the tokens into commands that operate on a single BRAM independently to reduce the impact of the BRAM bank conflicts. In the second stage, a recycle method is used where each BRAM command executes immediately without dependency checking and those that return with invalid data are recycled to avoid stalls caused by the RAW dependency. For a single Snappy input stream our design processes up to 16 input bytes per cycle. The end-to-end evaluation shows that the design achieves up to 7.2 GB/s output throughput or about an order of magnitude faster than the software implementation in the POWER9 CPU. This bandwidth for a single-stream decompressor is sufficient for an NVMe (PCIe x4) device. Two of these decompressor engines, operating on independent streams, can saturate a PCIe Gen4 or CAPI 2.0×8 interface, and the design is efficient enough to easily support data rates for an OpenCAPI 3.0×8 interface.

References

1. lzbench. available: https://github.com/inikep/lzbench. Accessed: 2019-05-15.
2. Uf sparse matrix collection. available: https://www.cise.ufl.edu/research/sparse/MM/LAW/hollywood-2009.tar.gz.
3. Zstandard. available: http://facebook.github.io/zstd/. Accessed: 2019-05-15.
4. Adler, M. (2015). pigz: A parallel implementation of gzip for modern multi-processor, multi-core machines. *Jet Propulsion Laboratory*.
5. Agarwal, K.B., Hofstee, H.P., Jamsek, D.A., Martin, A.K. (2014). High bandwidth decompression of variable length encoded data streams. US Patent 8,824,569.
6. Alpha Data. (2018) ADM-PCIE-9V3 User Manual. available: https://www.alpha-data.com/pdfs/adm-pcie-9v3usermanual_v2_7.pdf. Accessed: 2019-05-15.
7. Apache: Apache ORC. https://orc.apache.org/. Accessed: 2018-12-01.
8. Apache: Apache Parquet. http://parquet.apache.org/. Accessed: 2018-12-01.
9. Bartík, M., Ubik, S., Kubalik, P. (2015). LZ4 compression algorithm on FPGA. In *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, (pp 179–182): IEEE.
10. Fang, J., Chen, J., Al-Ars, Z., Hofstee, P., Hidders, J. (2018). A high-bandwidth Snappy decompressor in reconfigurable logic: work-in-progress. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis* (pp. 16:1–16:2): IEEE Press.
11. Fang, J., Chen, J., Lee, J., Al-Ars, Z., Hofstee, H. (2019). Refine and recycle: a method to increase decompression parallelism. In *2019 IEEE 30Th international conference on application-specific systems, architectures and processors (ASAP)* (pp. 272–280): IEEE.
12. Fang, J., Mulder, Y.T.B., Hidders, J., Lee, J., Hofstee, H.P. (2020). In-memory database acceleration on FPGAs: a survey. *The VLDB Journal*, *29*(1), 33–59. https://doi.org/10.1007/s00778-019-00581-w.

13. Fowers, J., Kim, J.Y., Burger, D., Hauck, S. (2015). A scalable high-bandwidth architecture for lossless compression on fpgas. In *2015 IEEE 23rd annual international symposium on Field-programmable custom computing machines (FCCM)* (pp. 52–59): IEEE.
14. Gilchrist, J. (2004). Parallel data compression with bzip2. In *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems* (vol. 16, pp. 559–564).
15. Google: Snappy. https://github.com/google/snappy/. Accessed: 2018-12-01.
16. Gopal, V., Gulley, S.M., Guilford, J.D. (2017). *Technologies for efficient LZ77-based data decompression*. US Patent App. 15/374,462.
17. Huebner, M., Ullmann, M., Weissel, F., Becker, J. (2004). Real-time configuration code decompression for dynamic fpga self-reconfiguration. In *2004. Proceedings. 18th international Parallel and distributed processing symposium* (pp. 138): IEEE.
18. Inc., C. (2016). ZipAccel-D GUNZIP/ZLIB/Inflate Data Decompression Core. http://www.cast-inc.com/ip-cores/data/zipaccel-d/cast-zipaccel-d-x.pdf. Accessed: 2019-03-01.
19. Jang, H., Kim, C., Lee, J.W. (2013). Practical speculative parallelization of variable-length decompression algorithms. In *ACM SIGPLAN Notices* (vol. 48, pp. 55–64): ACM.
20. Koch, D., Beckhoff, C., Teich, J. (2009). Hardware decompression techniques for FPGA-based embedded systems. *ACM Transactions on Reconfigurable Technology and Systems*, *2*(2), 9.
21. Leibson, S., & Mehta, N. (2013). *Xilinx ultrascale: The next-generation architecture for your next-generation architecture*. Xilinx White Paper WP435.
22. Mahoney, M. (2011). Large text compression benchmark available: http://www.mattmahoney.net/text/text.html.
23. Mahony, A.O., Tringale, A., Duquette, J.J., O'carroll, P. (2018). Reduction of execution stalls of LZ4 decompression via parallelization. US Patent 9,973,210.
24. Qiao, W., Du, J., Fang, Z., Lo, M., Chang, M.C.F., Cong, J. (2018). High-Throughput lossless compression on tightly coupled CPU-FPGA platforms. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (pp. 291–291): ACM.
25. Qiao, Y. (2018). *An FPGA-based Snappy Decompressor-Filter*. Master's thesis, Delft University of Technology.
26. Sitaridi, E., Mueller, R., Kaldewey, T., Lohman, G., Ross, K.A. (2016). Massively-parallel lossless data decompression. In *Proceedings of the international conference on parallel processing* (pp. 242–247): IEEE.
27. Stuecheli, J. A new standard for high performance memory, acceleration and networks. http://opencapi.org/2017/04/opencapi-new-standard-high-performance-memory-acceleration-networks/. Accessed: 2018-06-03.
28. Xilinx: Vitis Data Compression Library. https://xilinx.github.io/Vitis_Libraries/data_compression/source/results.html. Accessed: 2020-02-15.
29. Yan, J., Yuan, J., Leong, P.H., Luk, W., Wang, L. (2017). Lossless compression decoders for bitstreams and software binaries based on high-level synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *25*(10), 2842–2855.
30. Zhou, X., Ito, Y., Nakano, K. (2016). An efficient implementation of LZW decompression in the FPGA. In *2016 IEEE International parallel and distributed processing symposium workshops (IPDPSW)* (pp. 599–607): IEEE.
31. Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, *23*(3), 337–343.