Refine and recycle

A method to increase decompression parallelism

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

DOI
10.1109/ASAP.2019.00017

Publication date
2019

Document Version
Final published version

Published in
2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)

2019 IEEE 30th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)

Refine and Recycle: A Method to Increase Decompression Parallelism

Jian Fang*, Jianyu Chen*, Jinho Lee†, Zaid Al-Ars* and H. Peter Hofstee*†
{j.fang-1, z.al-ars, h.p.hofstee}@tudelft.nl, j.chen-13@student.tudelft.nl, leejinho@us.ibm.com
*Delft University of Technology, Delft, the Netherlands
†IBM Austin, Texas, USA

***Abstract*—Rapid increases in storage bandwidth, combined with a desire for operating on large datasets interactively, drives the need for improvements in high-bandwidth decompression. Existing designs either process only one token per cycle or process multiple tokens per cycle with low area efficiency and/or low clock frequency.**

We propose two techniques to achieve high single-decoder throughput at improved efficiency by keeping only a single copy of the history data across multiple BRAMs 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, a relaxed execution model is used where each BRAM command executes immediately and those with invalid data are recycled to avoid stalls caused by the read-after-write dependency.

We apply these techniques to Snappy 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 output throughput per decompressor, with each decompressor using 14.2% of the logic and 7% of the BRAM resources of the device. Therefore, a single decompressor can easily keep pace with an NVMe device (PCIe Gen3 x4) on a small FPGA, while a larger device, integrated on a host bridge adapter and instantiating multiple decompressors, can keep pace with the full OpenCAPI 3.0 bandwidth of 25 GB/s.

***Index Terms*—Snappy, decompression, FPGA, Acceleration**

I. Introduction

While much prior work has studied how to improve the compression speed of lossless data compression [1]–[3], the common case is to compress the data once for storage and decompress it multiple times whenever it is processed.

Recent studies [4]–[8] illustrate that FPGAs are 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, enhancing performance. Leveraging these advantages, the pipelined FPGA designs of LZSS [4], [5], LZW [6] and Zlib [7], [9] 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 [8] and [10] 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.

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 [11] 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. One decompressor keeps pace with an NVMe device (PCIe Gen3 x4) on a small FPGA. Compared to the software implementation, significant performance improvement is achieved.

Specifically, this paper makes the following contributions.

- We increase decomprssion 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. Our decompressor achieves up to 7.2 GB/s throughput.

In the remainder of this paper, Section II introduces Snappy and summarizes related work. Section III discusses solutions to address BRAM bank conflicts and RAW dependencies. Section IV details the Snappy decompressor architecture. Section V presents experimental results and Section VI contains a summary and conclusions.

2160-052X/19/$31.00 ©2019 IEEE
DOI 10.1109/ASAP.2019.00017

II. Background

A. Snappy (De)compression

Snappy is an LZ77-based [12] 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 [13] and ORC [14]. 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 history, Snappy records the original data in another type of token, the $literal$ tokens.

TABLE I: Procedure of Snappy decompression

```
1  while(!eof) {
2    reset(&history);
3    while(!end_of_block()){
4      read_tag_byte(&ptr, &type, &extra_len, &lit_len, &copy_len);
5      read_extra_bytes(&ptr, extra_len, &lit_len, &copy_offset);
6      if(type==copy){
7        read_history(history, copy_offset, copy_len, &buffer);
8        update_history_c(&history, buffer, copy_len);
9      }
10     else // type==lit
11       update_history_l(&history, &ptr, lit_len);
12   }
13   output(history);
14 }
```

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 [15] using the "lzbench" [16] benchmark, where the throughput reaches 1.8GB/s on a Core i7-6700K CPU running at 4.0GHz. Table I 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 I, 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 is 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 need 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 IV-C (second dependency) and section III-B (third dependency).

B. Related Work

Many recent studies consider improving the speed of lossless decompression. To address the block boundary problems, [17] explores the block-level parallelism by performing pattern matching on the delimiters to predict the the block boundaries. Unfortunately, this technique cannot be applied to Snappy because Snappy uses a fixed uncompressed block size (64KB) without any delimiters. 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 [18] or add some meta data to indicate the boundary of the blocks [19]. 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 [4]–[7]. However, these methods only process one token each FPGA cycle, limiting throughput.

Other works study the possibility of processing multiple tokens in parallel. [20] 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 [21], 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. [10] 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. [8] proposes a token-level parallel Snappy decompressor that can process two tokens every cycle. It uses a similar method as [10] 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 [19] provides a multi-round resolution method to handle the data dependency. 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 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 improves on our previous proof-of-concept work [22] and integrates it in a CAPI 2.0-enabled POWER 9 system.

III. The Refine and Recycle Method

A. The Refine Method 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. 1, to construct a 64KB history buffer, a minimum number of BRAMs are required: 16 4KB blocks for the Xilinx Ultrascale Architecture [23]. 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. 1). We call this conflict a *BRAM bank conflict* (BBC).

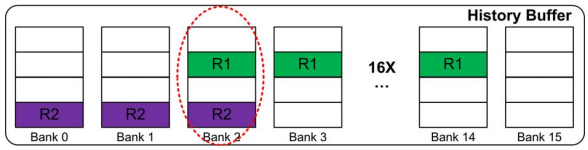

Fig. 1: An example of BRAM bank conflicts in Snappy

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. 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. 1, when a read request from a copy token ($R1$) has a BBC with another read request from another copy token ($R2$), the execution of $R2$ stalls and waits until $R1$ is finished. 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 [8] 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. 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. As illustrated in Fig. 1, $R1$ and $R2$ 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.

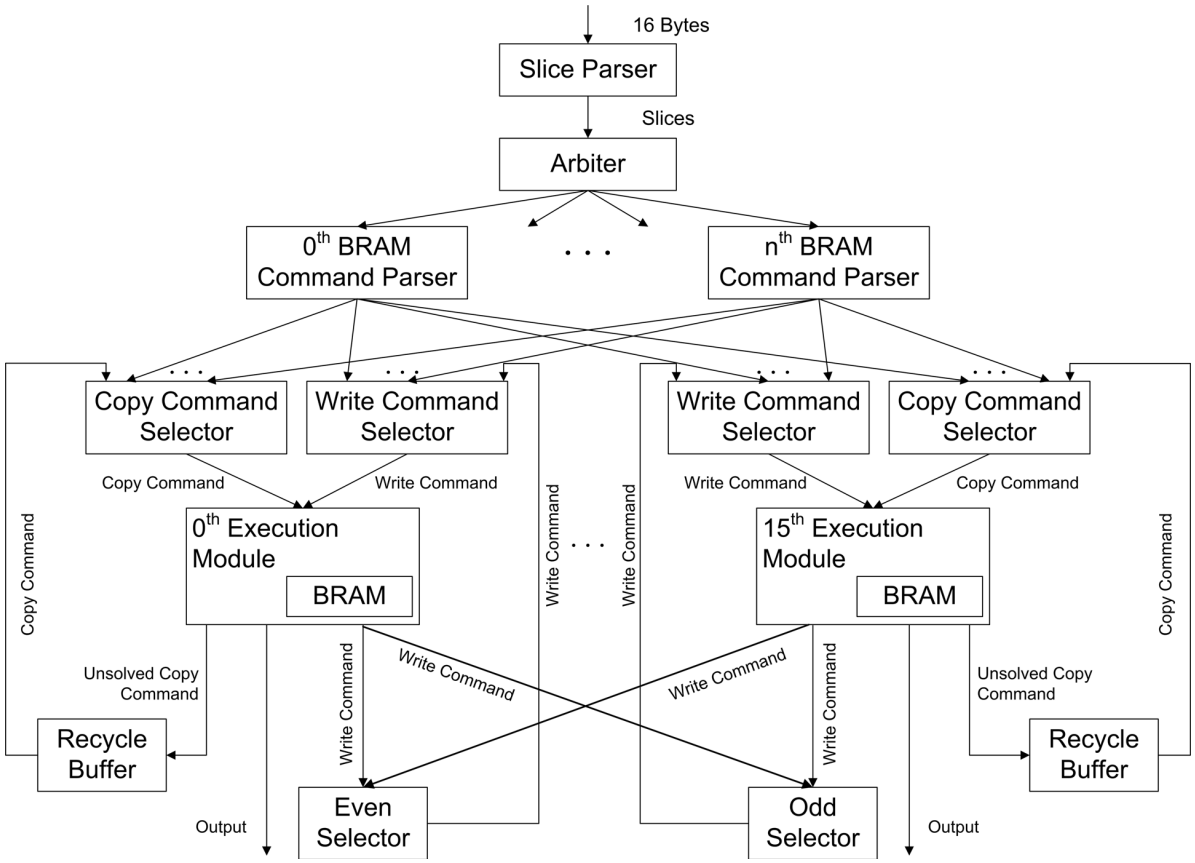

Fig. 2: Architecture overview

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 III-B. 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.

B. The Recycle Method 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 [8] 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, the follow-up tokens are allowed to be executed without waiting these two tokens are finished, which is very 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 follow-up tokens guarantees the correction 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 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 in a BRAM command granularity. Specifically, when a command needs to read the history data

that is 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 precedure repeatedly until the read data is valid.

This method executes the commands in a relaxed model and allows continuous execution on the commands without stalling the pipeline. The method provides more parallelism since it does not need to be restricted to the degree of parallelism calculated by dependency detection.

IV. Snappy Decompressor Architecture

A. Architecture Overview

Fig. 2 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 are utilized to perform the recycle mechanism.

The procedure starts with receiving a 16B input line in the slice parser. Together with the first 2B of the next input line, 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 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 the read data contains some valid bytes, new write commands are generated to write this data to its destination. If some bytes are still 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. 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.

B. 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.

C. Slice Parser

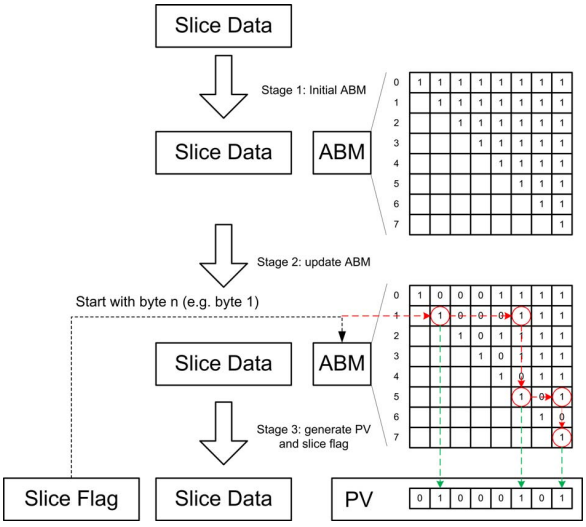

Fig. 3: Procedure of the slice parser and structure of the Assumption Bit Map

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 to 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 utilized 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 3, 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. 3). Then the corresponding row in the ABM is used to find the first byte of the next token (byte 5 in Fig. 3), 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.

D. BRAM Command Parser

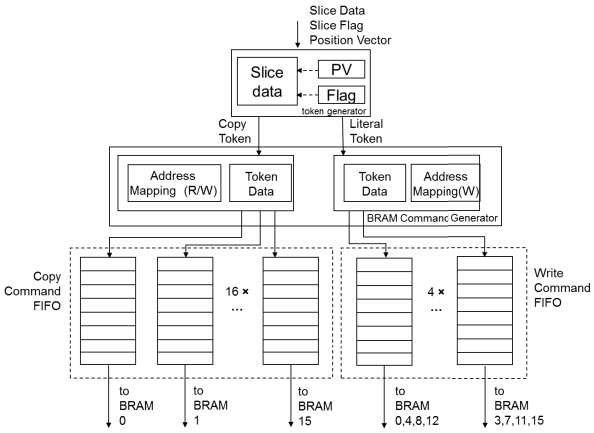

Fig. 4: Structure of BRAM command parser

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. 4. 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. 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.

E. Execution Module

The execution module performs BRAM command execution and the recycle mechanism. Its structure is illustrated in Fig. 5. 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 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.

F. 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.

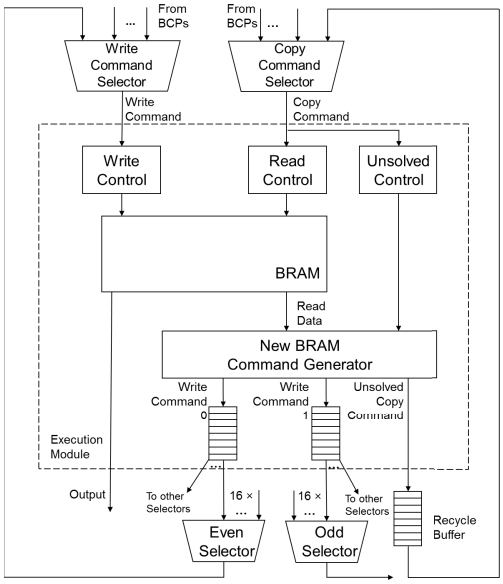

Fig. 5: Structure of execution module

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.

V. Evaluation

A. 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 [24] interface. The CAPI 2.0 interface on this card supports the CAPI protocol at an effective data rate of approximately 11 GB/s. The FPGA design is compared with an optimized software Snappy decompression implementation [16] 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 sets are listed in Table II. 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 [25] is an XML file dump from Wikipedia, while the Matrix is a sparse matrix from the Matrix Market [26]. We also use a very high compression ratio file (Geo) which stores geographic information.

TABLE II: Benchmarks used and throughput results

Files	Original size (MB)	Compression ratio	Throughput (GB/s) CPU	Throughput (GB/s) FPGA	Speedup
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

B. Resource Utilization

Table III 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.

TABLE III: Resource utilization of design components

Resource	LUTs	BRAMs[1]	Flip-Flops
Recycle buffer	1.1K(0.3%)	8(1.2%)	1K(0.1%)
Decompressor	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.

C. 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).

Fig. 6 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 [27], 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.

TABLE IV: 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) [9]	165	0.495	3.2	32	5.4K	10.5	93.9[1]	48.3
Snappy [8]	140	1.96	15	64	91K	32	22	62.7
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.

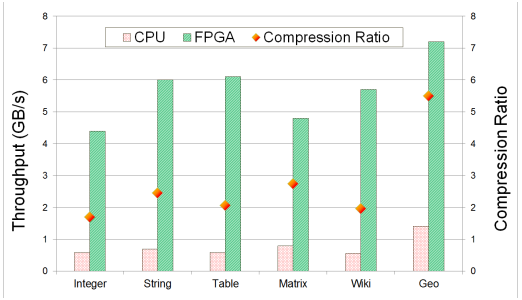

Fig. 6: Throughput of Snappy decompression

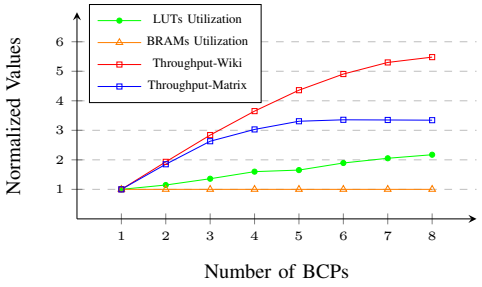

Fig. 7: Impact of number of BCPs

D. Design Trade-off with # of BCPs

As explained in section IV, the number of BCPs corresponds to the number of tokens that can be refined into BRAM commands per cycle. We compare the resource utilization 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 7. 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.

E. Comparison of Decompression Accelerators

We compare our design with state-of-the-art decompression accelerators in Table IV. 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 [9] and Snappy [8]. 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 [9] and 2.4x more BRAM efficient than the Snappy implementation in [8].

VI. Summary and 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 assume 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 make 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 x8 interface, and the design is efficient enough to easily support data rates for an OpenCAPI 3.0 x8 interface.

REFERENCES

[1] M. Bartík, S. Ubik, and P. Kubalik, "LZ4 compression algorithm on FPGA," in *Electronics, Circuits, and Systems (ICECS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 179–182.

[2] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 52–59.

[3] W. Qiao, J. Du, Z. Fang, M. Lo, M.-C. F. Chang, and J. Cong, "High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 291–291.

[4] M. Huebner, M. Ullmann, F. Weissel, and J. Becker, "Real-time configuration code decompression for dynamic fpga self-reconfiguration," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 138.

[5] D. Koch, C. Beckhoff, and J. Teich, "Hardware decompression techniques for FPGA-based embedded systems," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 2, no. 2, p. 9, 2009.

[6] X. Zhou, Y. Ito, and K. Nakano, "An efficient implementation of LZW decompression in the FPGA," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 599–607.

[7] J. Yan, J. Yuan, P. H. Leong, W. Luk, and L. Wang, "Lossless compression decoders for bitstreams and software binaries based on high-level synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2842–2855, 2017.

[8] Y. Qiao, "An FPGA-based Snappy Decompressor-Filter," Master's thesis, Delft University of Technology, 2018.

[9] C. Inc., "ZipAccel-D GUNZIP/ZLIB/Inflate Data Decompression Core," http://www.cast-inc.com/ip-cores/data/zipaccel-d/cast-zipaccel-d-x.pdf, 2016, accessed: 2019-03-01.

[10] K. B. Agarwal, H. P. Hofstee, D. A. Jamsek, and A. K. Martin, "High bandwidth decompression of variable length encoded data streams," Sep. 2 2014, uS Patent 8,824,569.

[11] Google, "Snappy," https://github.com/google/snappy/, accessed: 2018-12-01.

[12] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[13] Apache, "Apache Parquet," http://parquet.apache.org/, accessed: 2018-12-01.

[14] Apache, "Apache ORC," https://orc.apache.org/, accessed: 2018-12-01.

[15] "Zstandard," available: http://facebook.github.io/zstd/, accessed: 2019-05-15.

[16] "lzbench," available: https://github.com/inikep/lzbench, accessed: 2019-05-15.

[17] H. Jang, C. Kim, and J. W. Lee, "Practical speculative parallelization of variable-length decompression algorithms," in *ACM SIGPLAN Notices*, vol. 48, no. 5. ACM, 2013, pp. 55–64.

[18] M. Adler, "pigz: A parallel implementation of gzip for modern multiprocessor, multi-core machines," *Jet Propulsion Laboratory*, 2015.

[19] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *Proc. of the International Conference on Parallel Processing*. IEEE, 2016, pp. 242–247.

[20] A. O. Mahony, A. Tringale, J. J. Duquette, and P. O'carroll, "Reduction of execution stalls of LZ4 decompression via parallelization," May 15 2018, uS Patent 9,973,210.

[21] V. Gopal, S. M. Gulley, and J. D. Guilford, "Technologies for efficient LZ77-based data decompression," Aug. 31 2017, uS Patent App. 15/374,462.

[22] J. Fang, J. Chen, Z. Al-Ars, P. Hofstee, and J. Hidders, "A high-bandwidth Snappy decompressor in reconfigurable logic: work-in-progress," in *Proceedings of the International Conference on Hardware/-Software Codesign and System Synthesis*. IEEE Press, 2018, pp. 16:1–16:2.

[23] S. Leibson and N. Mehta, "Xilinx ultrascale: The next-generation architecture for your next-generation architecture," *Xilinx White Paper WP435*, 2013.

[24] J. Stuecheli, "A new standard for high performance memory, acceleration and networks," available: http://opencapi.org/2017/04/opencapi-new-standard-high-performance-memory-acceleration-networks/, accessed: 2018-06-03.

[25] M. Mahoney, "Large text compression benchmark," available: http://www.mattmahoney.net/text/text.html, 2011.

[26] "Uf sparse matrix collection," available: https://www.cise.ufl.edu/research/sparse/MM/LAW/hollywood-2009.tar.gz.

[27] Alpha Data, "ADM-PCIE-9V3 User Manual," available: https://www.alpha-data.com/pdfs/adm-pcie-9v3usermanual_v2_7.pdf, 2018, accessed: 2019-05-15.