IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 26 June 2022, accepted 1 August 2022, date of publication 5 August 2022, date of current version 10 August 2022.
Digital Object Identifier 10.1109/ACCESS.2022.3196775

RESEARCH ARTICLE

A Design of Parallel Content-Defined Chunking System Using Non-Hashing Algorithms on FPGA

HUNG VUONG, HUNG NGUYEN, AND LINH TRAN
Department of Electronics, Ho Chi Minh City University of Technology (HCMUT), Ho Chi Minh City 700000, Vietnam
Vietnam National University Ho Chi Minh City, Ho Chi Minh City 700000, Vietnam
Corresponding author: Linh Tran (linhtran@hcmut.edu.vn)

ABSTRACT Content-defined chunking is a common method in many applications such as data deduplication and data synchronization. In recent years, new CDC algorithms using non-hashing methods have been developed, and positive results have been obtained. However, most of the algorithms are developed for single-thread computation on microprocessors. After analyzing some popular CDC algorithms, we observed that the algorithms using the basic sliding window protocol are more feasible to process in parallel. In this work, we proposed a new parallel chunking method that aims for hardware implementation. Additionally, we used the PCI algorithm, which does not include hash functions, to implement a multi-thread chunking system on FPGA devices. By exploiting the strength of the FPGAs, our proposed design achieves not only high computational speed but also great scalability.

INDEX TERMS FPGA, content-defined chunking, non-hashing method, PCI algorithm, hardware design.

I. INTRODUCTION

Digital data is considered as a valuable resource in modern-day. Data storage and deduplication prove themselves as difficult challenges [1], [2]. Studies from Microsoft [3], [4] shows that about 50% of data stored in primary memory and 80% of data stored in backup memory are duplicated. Data deduplication techniques are currently deployed to save storage space and improve memory performance in data centers [3]–[6]. Moreover, the technique is also applied in low-bandwidth network systems to reduce duplicated packets transmitted over the network [7], [8].

A data deduplication system goes through two phases: data chunking and searching for duplicate values. Firstly, the data is broken into chunks of equal or non-equal lengths depending on the chosen algorithm. These chunks are represented by the fingerprints, which are the results of hash functions. The fingerprints are used to compare and check if the data has been stored before. As the first phase of computation, data chunking makes a huge impact on the performance of the whole deduplication process. If the data is divided into equal chunks by a fixed-size chunking (FSC) algorithm, the system can process it quickly. However, fixed-size chunking encounters the byte shifting problem. A minor change in any chunk leads to changes in the following chunks. It is for this reason that variable size chunking algorithms are more popular despite taking up a lot of resources and computation effort. Although many effective content-defined chunking (CDC) algorithms are researched, it is a huge challenge for existing algorithms to catch up with the throughput of current storage devices. Therefore, studies on data chunking play an important role in improving the performance of the data deduplication system, freeing CPU resources and optimizing the capacity of storage devices.

Basic Sliding Window (BSW) is one of the most basic and popular CDC algorithms [9]. Rabin roll hash [10] is commonly used in BSW to improve data chunking performance. However, calculating the fingerprints takes a considerable amount of time. Many studies have proposed new chunking algorithms that not using hash functions to decrease computation time. Local Maximum Chunking (LMC) [11] is a typical one when combining the sliding window and the contents of the data bytes to find the cutoff points in a file. This new approach uses comparison operations instead of modulo operations in the Rabin hash function. Two other algorithms with similar approaches are Asymmetric Extremum

The associate editor coordinating the review of this manuscript and approving it for publication was Harikrishnan Ramiah.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

(AE) [12] and Rapid Asymmetric Maximum (RAM) [13] that both use two windows, one of fixed-size and one of variable size, to determine the cutoff point. By eliminating the sliding window, AE and RAM reduce the great number of comparisons. In contrast, the lately Parity Check of Interval (PCI) algorithm [14], which examines the number of 1s in a sliding window and compares that value with a preset threshold to determine a cut point, gives very positive results.

In addition to researching new algorithms, improving and applying existing algorithms to exploit CPU capabilities is also a research direction that has huge attention [15], [16]. Besides, there are researches on powerful hardware such as GPU [17] and FPGA [18] in order to reduce CPU resource consumption and enhance the computing performance. However, these studies mainly overcome the Rabin chunking algorithm's weaknesses without taking advantage of the new algorithms. Existing methods like P-Dedupe [15] and MUCH [16] can replace Rabin Chunking with hashless CDC algorithms to simplify the chunking process. However, the parallel chunking method of P-Dedupe is not optimized because it requires plenty of recomputation to achieve correct results. Meanwhile, MUCH requires less recomputation but heavily depends on the chunk size constraints, which are not recommended in recent CDC algorithms. We proposed a parallel chunking method without chunk size restriction requirements to enhance the content dependence of hashless CDC algorithms. Moreover, because the number of CPU threads restricts software implementation, we proposed a FPGA implementation to achieve more scalability and higher throughput.

The contributions of this paper are:

- A new parallel chunking method which is independent of the chunk size constraints.
- A high speed, scalable FPGA implementation applying the proposed method.

The rest of the paper is organized as follows. Section II presents the theoretical basis of popular data chunking algorithms and motivations of our work with the PCI algorithm. Section III describes our parallel chunking method and the proposed FPGA implementation. Section IV provides the experimental results and hardware evaluation. Section V concludes the paper.

II. BACKGROUND AND MOTIVATION

This section provides the background of some popular CDC algorithms including Rabin Chunking, LMC, AE, RAM and PCI, and motivations of our research.

A. BACKGROUND

1) RABIN CHUNKING ALGORITHM

Data bytes are filled into a sliding window for Rabin fingerprint calculation. If the fingerprint matches the preset value, the cutoff is determined as the position of the last byte of the sliding window. Otherwise, change the sliding window position to one byte forward, do the hashing calculation and fingerprint comparison repeatedly. The algorithm terminates when all the cut points in the file have been found. The algorithm's operation is shown in Fig. 1.

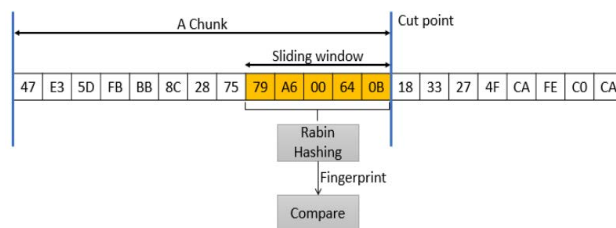

FIGURE 1. Operation of the Rabin Chunking algorithm.

The Rabin hash function is used to calculate the fingerprints because it can reuse the previously calculated fingerprint result, along with the newest and oldest byte in the window, to compute a new result. Even though Rabin rolling hash is suitable for the sliding window approach, it is a time-consuming function and abates the ability to resist low entropy strings. In addition, the chunk variance of Rabin Chunking is large because of the high probability of getting a long chunk [11], [12]. A maximum threshold could be imposed on the size of data chunks to overcome these defects, but this causes the degradation of boundaries-shifting resistance as some chunks do not depend on data content anymore. Algorithm 1 presents the Rabin Chunking algorithm.

Algorithm 1 Rabin Chunking Algorithm

Input:
- Data file: *file*
- Comparable value: *preset_value*
- Sliding window size: *width*

Output: Cut point: *cut_point*
function Rabin_chunking(*file*, *preset_value*, *width*)
  $cut_point = 1$
  $index = 0$
  while ($byte$ = Read_byte($file$)) **do**
    $array[index\%width + 1] = byte$
    if (Size($array$) $>=$ $width$) **then**
      if (Hash($array, index, width$) $==$ $preset_value$) **then**
        return cut_point
      end if
    else
      continue
    end if
    $cut_point = cut_point + 1$
  end while
end function

---

2) LOCAL MAXIMUM CHUNKING (LMC) ALGORITHM

The LMC algorithm uses a sliding window consisting of three components: a left window, a local maximum byte and a right

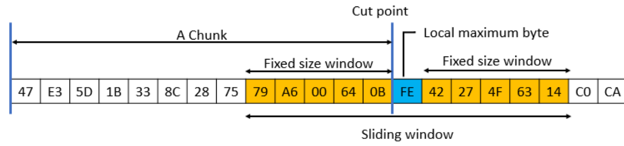

FIGURE 2. Operation of the Local Maximum Chunking algorithm.

window as illustrated in Fig. 2. The local maximum byte is determined as a cut point if its value is greater than all the other bytes in two fixed-size windows. Otherwise, the data window shifts one byte forward and do the comparisons again until a cutoff is found [11].

The advantage of the LMC algorithm is that it is less affected by adding, removing or changing data bytes. If a byte is changed but its value is still less than the maximum local byte value, then only that chunk is affected. In case that data byte creates a new cut point, one more following chunk is changed as a result but the others are preserved. The LMC algorithm has many downsides such as not being able to remove low entropy strings and high variance in data chunk size. In addition, LMC algorithm is practically slower than Rabin Chunking algorithm because all the bytes in the window must be re-checked whenever the sliding window moves to a new position. Algorithm 2 presents the LMC algorithm.

Algorithm 2 Local Maximum Chunking Algorithm

Input:
- Data file: *file*
- Left, right window size: *width*

Output: Cut point: *cut_point*

function LMC_chunking(*file*, *width*)
  $i = 1$
  $max_value = 0$
  $max_position = 1$
  $start = 1$
  while ($byte$ = Read_byte($file$)) **do**
    if ($byte <= max_value$) **then**
      if ($i == max_position + width$) **then**
        if ($max_position >= start + width$) **then**
          $start = max_position + 1$
          $cut_point = max_position$
          return cut_point
        end if
      end if
    else
      $max_value = byte$
      $max_position = i$
    end if
    $i = i + 1$
  end while
end function

3) ASYMMETRIC EXTREMUM (AE) ALGORITHM

The AE algorithm uses a variable size window and a fixed-size window. Similar to the LMC algorithm, there is a byte between two windows called "extreme-valued byte" which could be either minimum or maximum value. In this paper, we used the maximum value to discuss for convenience since Y. Zhang *et al.* [12] prove that choosing the extreme-valued byte as the largest or smallest byte does not make a significant difference in performance. If the extreme-valued byte is greater than all bytes in the left window and not less than the bytes in the right window, the cut point is located at the end of the fixed-size window [12] as shown in Fig. 3.

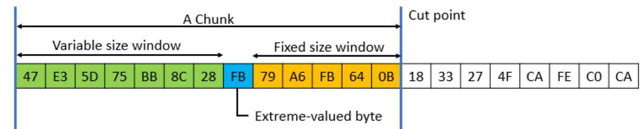

FIGURE 3. Operation of the Asymmetric Extremum algorithm.

The two windows and an extreme-valued byte cover the whole chunk in the AE algorithm. The number of comparisons of the AE algorithm is significantly lower than the LMC algorithm because each byte is used for comparison once. However, the extreme-valued byte located in the middle of the data chunk makes this algorithm less efficient in byte shifting resistance. When a byte larger than the extreme-value byte is added to the fixed window of a data chunk, that chunk and the next one are changed, and the subsequent data chunks might be affected. In return, the fact that the extreme-valued byte is located on the left side of the fixed window and accepts ties in this window helps the AE algorithm detect low entropy strings effectively. Algorithm 3 presents the AE algorithm.

4) RAPID ASYMMETRIC MAXIMUM (RAM) ALGORITHM

In contrast to the AE algorithm, RAM uses a fixed-size window followed by a variable size window. The algorithm starts by finding the maximum value in the left window. The first byte position in the right window whose value is greater than or equal to the maximum value of the left window will be selected as the cut point [13]. The algorithm's operation is shown in Fig. 4.

Like the AE algorithm, the RAM method has limitations for the byte shifting problem. If a byte is added or changed to a value greater than the maximum byte in the chunk, the cut point of that chunk will be changed, which might affect subsequent chunks. Additionally, the RAM algorithm has difficulty dealing with low entropy strings. For instance, if a sequence of bytes with small values occurs in the right window and the maximum value in the fixed-size window is large enough, the cut point can only be found when this sequence ends. This drawback causes an increase in chunk size variance as a result. In the study [13], Widodo *et al.* solve this problem by applying the maximum threshold to control the chunk size variance. Algorithm 4 presents the RAM algorithm.

Algorithm 3 Asymmetric Extremum Algorithm

Input:
- Data file: *file*
- Right window size: *width*

Output: Cut point: *cut_point*
function AE_chunking(*file*, *width*)
$i = 1$
$max_value = 0$
$max_position = 1$
while ($byte$ = Read_byte(*file*)) **do**
  if ($byte <= max_value$) **then**
    if ($i == max_position + width$) **then**
      $cut_point = i$
      return cut_point
    end if
  else
    $max_value = byte$
    $max_position = i$
  end if
  $i = i + 1$
end while
end function

Algorithm 4 Rapid Asymmetric Maximum Algorithm

Input:
- Data file: *file*
- Left window size: *width*

Output: Cut point: *cut_point*
function RAM_chunking(*file*, *width*)
$i = 1$
$max_value = 0$
$max_position = 1$
while ($byte$ = Read_byte(*file*)) **do**
  if ($byte >= max_value$) **then**
    if ($i > width$) **then**
      $cut_point = i$
      return cut_point
    end if
    $max_value = byte$
    $max_position = i$
  else
    continue
  end if
  $i = i + 1$
end while
end function

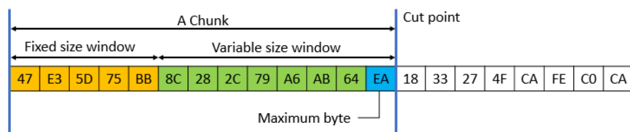

FIGURE 4. Operation of the Rapid Asymmetric Maximum algorithm.

5) PARITY CHECK OF INTERVAL (PCI) ALGORITHM

Initially, the PCI algorithm establishes a sliding window and a preset value as the comparison threshold. After filling the window with data bytes, if the number of 1s in the window is not less than the preset value, the cut point is determined as the last position of that window. If the cutoff condition is not satisfied, slide the window to the next byte and re-perform the comparison of the sum of 1s with the threshold. This process is repeated until all the cut points and chunks of a data stream are found [14]. The algorithm's operation is shown in Fig. 5.

Similar to Rabin Chunking, the PCI algorithm could reuse the previous result for the following calculation. Nevertheless, it costs less computation time due to simple additions and comparisons. The PCI algorithm is also great at byte shifting resistance. If a byte is changed, there is a chance that a new cutoff will occur and split that chunk in half, but the other chunks will not be affected. In the worst case that a byte is added, deleted, or changed near the chunk boundary, that chunk and the subsequent one are affected. One of the PCI drawbacks is the varying size of chunks due to the geometric distribution [14]. Besides, the ability to eliminate low entropy strings is another aspect to consider. Suppose a byte sequence has approximately the same value that the sum of 1s of the sliding window does not exceed the preset value. In that case, the cutoff is nearly impossible to find unless changing the algorithm configuration. Algorithm 5 presents the PCI algorithm.

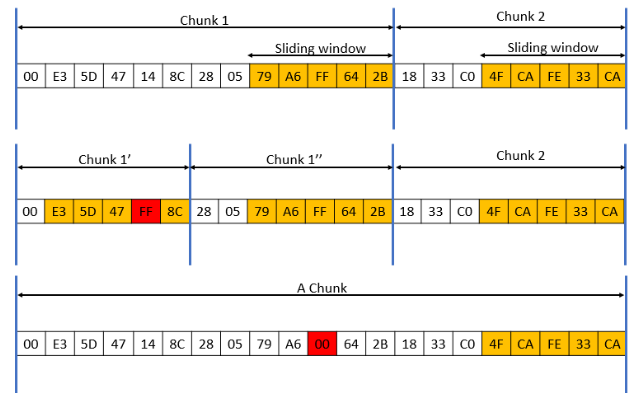

FIGURE 5. Operation of the Parity Check of Interval algorithm.

B. MOTIVATION

In the publication [12], Y. Zhang and colleagues gave some important criteria used to evaluate the CDC algorithm and were widely used in later studies [13], [14]. The evaluation criteria include:

- *Content-defined*: In order to limit the effect of byte shifting, chunking algorithms need to decide the cutoff point based on the content of the data.
- *Low chunk size variance*: Data deduplication efficiency is significantly influenced by the chunk size variance.

TABLE 1. Content-defined chunking algorithm comparisons.

Criteria	Content-Defined chunking algorithm				
	Rabin Chunking	LMC	AE	RAM	PCI
Content dependent	Yes	Yes	Yes	Yes	Yes
Chunk variance	Large	Large	Small	Large, depends on limitation	Large
Low entropy string elimination	Yes, depends on limitation	No	Yes	Yes, depends on limitation	Yes
Computation speed	Slow	Very Slow	Fast	Very Fast	Slow

Algorithm 5 Parity Check of Interval Algorithm

```
Input:
  • Data file: file
  • window size: width
  • Comparison threshold: thres_value
Output: Cut point: cut_point
function PCI_chunking(file, width, thres_value)
i = 1
index = 0
array[] = {−1}
while (byte = Read_byte(file)) do
  array[index%width + 1] = byte
  if (Content(array) >= width) then
    if (Parity(array, index, width) >= thres_value) then
      cut_point = i
      return cut_point
    else
      continue
    end if
  end if
  i = i + 1
end while
function Parity(array, index, w)
i = index
num = 0
parity_of_byte[ ] = 0, 1, 1, . . . , 8; //size: 256
while ((i! = index − 1)) do
  num = num + parity_of_byte[array[i]]
  i = (i + 1)%w
end while
return num
end function
```

The deduplication efficiency is high if the variance of chunk size is low and vice versa [9].

- *Capable of detecting and eliminating low-entropy strings*: In practice, files may contain repetitive bytes with low variance. The CDC algorithm is expected to have the ability to identify and eliminate these strings.
- *High computation speed*: The CDC algorithm is desired to be simple and avoid time-consuming calculations in order to achieve high throughput.
- *Less restriction on chunk size*: Minimum threshold and maximum threshold are often used to reduce data chunks that are too small or too large. Imposing such limitations reduces the variance of chunk size [9] but it makes chunks become less content dependent.

Based on the theoretical basis presented in the previous section and experimental results presented in studies [12]–[14], Table 1 illustrates the comparisons of CDC algorithms. Practical experiments show that RAM and AE algorithms are always in the leading positions in computation speed and are the top candidates for parallel processing research. Usually, the algorithms process data sequentially. If the data stream is divided into segments for parallel processing, it is necessary to ensure chunk consistency, which means the results are the same as chunks sequentially processed. Another vital aspect to consider is the difficulty in chunk post-processing when executing an algorithm in parallel.

Current parallel studies are mainly done with the Rabin Chunking algorithm [15], [16], [19]. These models all start with splitting the data file into several equal segments. Then, these segments of data will be processed by parallel threads simultaneously. The difference between parallel studies is how they deal with chunks across multiple segments. A multi-thread design separates the chunking process into two stages. The first stage is chunking the input segments in parallel. After parallel threads chunk input segments, the adjacent chunks between segments need to be post-processed in the second stage because their boundaries are not content dependent. The post-process stage recomputes the boundaries of adjacent chunks between segments to achieve the same results as the sequential algorithm. However, after the actual cut points of the adjacent chunks are found, byte shifting problems might occur and affect the following chunks in segments.

Rabin Chunking and LMC algorithms only affect a few adjacent chunks when the byte shifting problem occurs, so it is suitable for parallel applications. Due to the LMC's low entropy strings elimination ability and the low throughput, Rabin Chunking is more prevalent in multi-threaded implementation studies. Research [18] proposed a hardware implementation for Rabin fingerprint calculations and achieved very positive results as the design could operate at 300 MHz.

The AE algorithm has the advantages of low chunk size variance and high throughput. When two adjacent chunks between segments are post-processed, the following chunks might be influenced. The unpredictable number of affected chunks raises concerns about the capacity of buffers storing chunks for recalculation and the impact of this stage on chunking throughput. It is desirable for a simple

post-processing method as the one proposed by Ni *et al.* [19] that all the potential cut points are calculated and saved for chunk boundary determination. However, this method is not applicable to the AE algorithm because there is no sliding window used in the algorithm, and chunks are partitioned by different cutoff conditions due to unpredictable extreme-value bytes.

The RAM algorithm starts by finding the maximum value in a fixed-size window and then determines the cutoff to be the position of the larger byte outside that window. With this minor change compared to the AE algorithm, RAM improves its throughput but lessens the byte shifting resistance. When the chunk boundary moves, it might change the maximum value in the next chunk's fixed-size window, and the cut points of the subsequent chunks might be redefined as a result. Similar to AE, RAM algorithm does not have a fixed comparing value, so it is hard to marshal chunks affected by segment boundaries in the post-processing stage.

PCI is our next candidate for parallel chunking because of the difficulty in marshalling adjacent chunks of AE and RAM algorithms. PCI algorithm determines the cut points by comparing the total of 1s in the sliding window with a preset threshold. As a result, the PCI algorithm has better throughput than Rabin Chunking and LMC. Because the sliding window size limits the number of recalculations, it is more simple to post-process chunks generated in parallel. For instance, the first and last chunks of adjacent segments could be stored, then combined and re-computed to achieve expected results. Additionally, other proposed methods [16], [19] could be referred to and applied to the PCI algorithm thanks to the same sliding window protocol.

Ni *et al.* proposed an efficient approach called SS-CDC [19] that executes the chunking process in two stages. The first stage computes all potential cut points in parallel and marks their positions in bit arrays. In the second stage, the chunk boundaries are quickly selected from the candidates based on the chunk size requirements. This stage, however, can only perform sequentially because it needs to know the previous valid cutoff position to determine the next chunk boundary. Consequently, the post-processing stage requires a sizeable cache for temporarily storing data from multiple threads until bit arrays are fully filled.

Won *et al.* developed another method, MUCH, that partitions a data stream in parallel but still preserves the chunk consistency [16]. MUCH employs Dual Mode Chunking method which uses the chunk size constraints of CDC algorithms to separate a usual chunking process into *slow mode* and *accelerated mode*. The chunking thread starts in slow mode and ignores the lower and upper bounds of the chunk size. If certain conditions are satisfied, the chunking thread will switch to accelerated mode and apply the chunk size restrictions. After segments are processed in parallel, chunks generated in slow mode are used for marshalling by coalescing and splitting to achieve final results. In contrast, chunks generated in accelerated mode do not need to be stored for post-processing.

After analyzing some existing algorithms, we observed that it is challenging to make multi-thread implementations for AE and RAM algorithms due to the complicated post-process. Because both AE and RAM do not have a fixed comparing value, the post-process of adjacent chunks between segments might affect the subsequent chunks, which means more recomputation. Assuming that multiple post-processing threads operate in parallel. Each thread combines with the last chunk of the preceding thread to recalculate. If the last chunk is changed because of the post-process, the re-computational results of the next post-processing threads are wrong. This problem makes it difficult to implement parallel post-processing threads.

Meanwhile, algorithms with a rolling window like PCI and Rabin Chunking have many approaches to process in parallel but still preserve the result consistency, such as SS-CDC or MUCH. SS-CDC is storage resource consuming as chunk boundary determination has to perform sequentially. MUCH requires less storage capacity since only chunks generated in slow mode are needed for post-processing. However, because MUCH relies on the chunk size restrictions for chunking and marshalling, the PCI algorithm cannot be applied without adding these limitations. In this paper, we propose a parallel CDC chunking method independent of the chunk size constraints. Moreover, we take advantage of the low computational overhead of the PCI algorithm to implement a high speed, scalable chunking system on FPGA devices with our proposed method.

III. PARALLEL CHUNKING METHOD AND HARDWARE DESIGN

A. PROPOSED PARALLEL CHUNKING METHOD

Our parallel chunking method includes two stages: (i) determining the cut points of data segments and (ii) post-processing the chunks affected by the segment boundaries. Assuming that the CDC algorithm has the sliding window of size W. Before starting phase one, a file is partitioned into equal segments which overlap $W-1$ bytes with the preceding adjacent segments for sliding window computation. The size of each segment is flexibly set, but remember that the smaller the size is, the more segments are created, and the more post-processing operations are. After the segmentation, the segments are simultaneously fed into computational threads to find the cut points.

As our proposed method also aims for hardware implementation, all potential cut points are calculated in the first stage to avoid additional computation in the post-processing stage. However, it is desirable for another marshalling method rather than SS-CDC because the sequential post-process requires expensive memory resources. Therefore, we developed a modified version of the Dual Mode Chunking method to reduce the number of chunks for marshalling. We observed that potential cut points could be used to create new switching conditions independent of the chunk size restrictions.

A chunking thread operates in two different modes called *slow mode* and *fast mode*. Every segment is firstly processed in slow mode, where all potential cut points are calculated for the marshalling stage. The chunking process of a thread executes in slow mode until a *transition cut point* is found. When the thread switches to fast mode, it stops finding potential cut points and focuses on determining chunk boundaries. Therefore, chunks calculated in fast mode do not need to be post-processed unless they are ended by segment boundaries. A cutoff will be defined as a transition cut point if it satisfies all the following conditions:

- The cutoff is not at positions from W to $2(W-1)$.
- The distance between the last potential cutoff and the current one is greater than or equals the sliding window size W.

If the chunking thread wants to operate in fast mode, it needs to know the preceding chunk boundary to achieve accurate results. It means that determining a transition cut point is actually finding a chunk boundary without knowing the start of that chunk. Because data chunks cannot be smaller than the sliding window size, the distance between potential cutoffs might be used to determine the chunk boundaries. Fig. 6 and Fig. 7 illustrates some examples. P_1, P_2, P_3 and P_4 are potential cutoffs. The distance between two adjacent potential cut points are labeled as D_{12}, D_{23} and D_{34}. Fig. 6 assumes that $D_{12} < W$, $D_{23} < W$, $D_{34} > W$ and P_1 is a chunk boundary. Consequently, P_3 and P_4 are selected as chunk boundaries since $D_{12}+D_{23} > W$ and $D_{34} > W$. Fig. 7 indicates another case that $D_{12} > W$, $D_{23} < W$, $D_{34} > W$ and both P_1 and P_2 are chunk boundaries. Because D_{23} does not meet the minimum chunk size W, P_4 is determined as the next boundary. In both cases, P_4 is chosen as a valid cut point regardless of the state of P_3 since D_{34} is always larger than the minimum chunk size restriction.

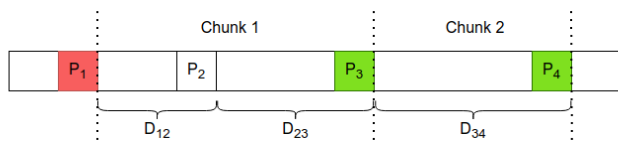

FIGURE 6. Example of chunk boundary determination. $D_{12} < W$, $D_{23} < W$, $D_{34} > W$ and P_1 is a chunk boundary.

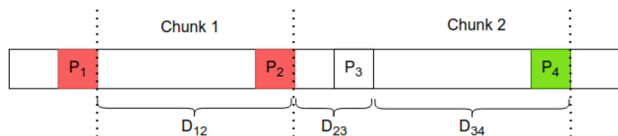

FIGURE 7. Cases of chunk boundary determination. $D_{12} > W$, $D_{23} < W$, $D_{34} > W$ and both P_1 and P_2 are chunk boundaries.

We observed that a potential cutoff could be determined as a transition cut point if the distance between it and the last potential cutoff is not less than the sliding window size. However, this is insufficient to determine a transition cut point at some positions. The first $W-1$ bytes of a segment are overlapped with the last $W-1$ bytes of the preceding one so that any potential cutoff at positions from W to $2(W-1)$ might belong to a chunk lying across multiple segments. Because the processing thread is unaware of the beginning of the chunk, it cannot determine any potential cut point in these positions as a transition cut point. In contrast, if the first potential cutoff is found after position $2(W-1)$, it is determined as a transition cut point without knowing the preceding chunk boundary position. This is because the distance between them is always larger than the window size, even in the worst case that the preceding chunk boundary is the segment boundary. Therefore, only potential cutoffs at positions from W to $2(W-1)$ are not considered as transition cut point candidates.

The marshalling stage selects potential cut points to generate chunks larger than the sliding window. In each post-processing thread, because the beginning position of the uncompleted chunk is known, it could select proper chunk boundaries from the set of potential cut points of the following segment to complete that chunk. If the transition cut point is not met, the post-processing thread will continue to produce chunks from the slow mode data bytes.

B. HARDWARE DESIGN ON FPGA

Fig. 8 shows the general hardware diagram of the design. The segments are fed into computational threads simultaneously. In each thread, the Chunking Module starts calculating potential cutoffs in slow mode until it meets certain conditions and switches to fast mode. In fast mode, the Chunking Module partitions the rest of the segment into chunks, which do not require additional post-processing except the last one. Because AE and RAM do not have a fixed comparing value, they cannot be used to calculate potential cut points in slow mode. Therefore, it is not suitable to apply AE and RAM in our proposed method. In this paper, we chose the PCI algorithm for our FPGA implementation because of its simplicity.

After the first chunking stage, data bytes and cut point positions are saved in two separated buffers depending on the operating mode. The buffers are called Slow Mode Buffer and Fast Mode Buffer. The Marshalling Module first read the Fast Mode Buffer data of the same thread. When the

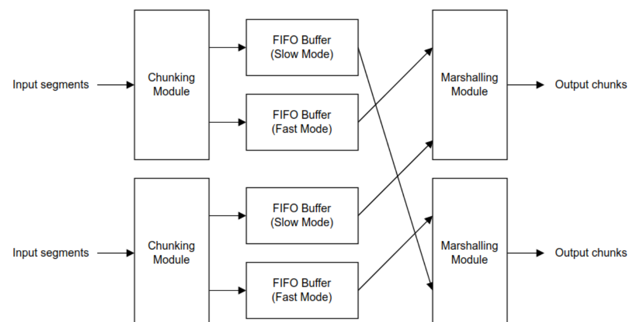

FIGURE 8. Parallel chunking design.

last chunk of a segment is read out, the Marshalling Module requests data from the next thread's Slow Mode Buffer and coalesces them with the uncompleted chunk. It keeps reading the Slow Mode Buffer and re-checking potential cut points to form precise chunks until a transition cut point is aware. Then, it switches back to the preceding thread's Fast Mode Buffer and repeats this process until the buffers are empty. The Slow Mode Buffer might overflow if a transition cut point is missed. In this case, the last byte stored in the buffer serves as the transition cut point even though it is not content-dependent. The buffer overflow probability could be reduced by increasing the buffer depth or decreasing the average chunk size. However, because the FPGA internal memory is limited, we have to trade the chunk consistency for a scalable design.

1) CHUNKING MODULE

The input segments are sent to the chunking module to find the cut points, along with some associated signals such as *start* and *end* to indicate the first and last byte of a data file.

In fast mode, the Chunking Module uses the PCI algorithm to chunk the segments. The cut points found in this mode are valid and do not need to be post-processed.

In contrast, in slow mode, the PCI algorithm is used with an adjustment to find all the potential cut points used in the Marshalling Module. The differences with the original algorithm lead to the need for a Chunking Controller to control the chunking process, as shown in Fig. 9.

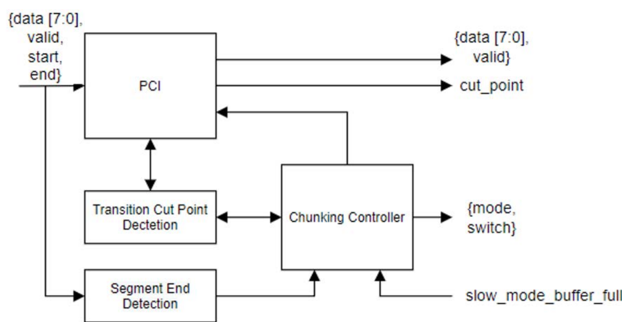

FIGURE 9. Hardware design of Chunking Module.

According to the state machine in Fig. 10, when the chunking module starts processing the input segments, by default, it always operates in slow mode. However, because the first segment of a file is not contiguous to any other segment, Chunking Controller enters fast mode as soon as it receives the *start* signal. For the remaining segments, Chunking Module processes in slow mode until it finds a transition cut point or the buffer overflows with a large data chunk.

When operating in fast mode, the Chunking Controller continuously checks the signals indicating the end of a segment or a file and will switch back to slow mode if either signal is active. The *mode* signal in Fig. 9 represents the current state of the Chunking Controller. It decides which

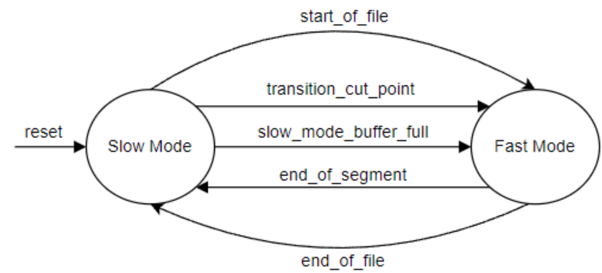

FIGURE 10. Chunking Module operation.

buffer to write data. In addition, a *switch* signal is also added to determine when to change the operating mode.

2) MARSHALLING MODULE

The hardware design of the Marshalling Module is presented in Fig. 11. A Marshalling Module connects to a Slow Mode Buffer and a Fast Mode Buffer of two different threads, as shown in Fig. 8, but only reads data from either of them in each clock cycle.

Fig. 12 details the operation of reading data from two buffers. Initially, in the Idle state, the Marshalling Module checks the status of the Fast mode Buffer and reads data from this buffer as soon as it is informed that the buffer is not empty. The data bytes and cut points read from the Fast Mode Buffer will be sent straight to the output without further processing. If a received signal *switch* is active high, the Marshalling Module switches to reading data from the Slow Mode Buffer. Data and the corresponding potential cut points will be post-processed to achieve valid chunks.

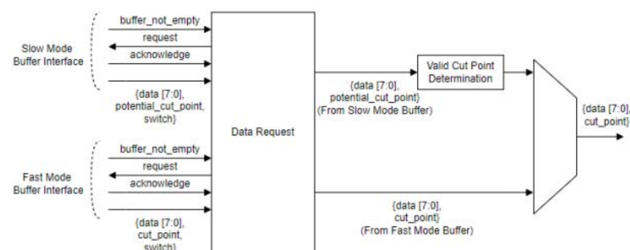

FIGURE 11. Hardware design of Marshalling Module.

While reading data from the Slow Mode Buffer, the Marshalling Module changes its request state when it receives an active high *switch* signal. If the Fast Mode Buffer is not empty, the Marshalling Module will switch to read data from this buffer. Otherwise, it will return to the Idle state.

C. TESTING MODEL

In this section, we discuss the verification model illustrated in Fig. 13, which is used for checking our parallel chunking method. The research is novel when parallelizing the PCI algorithm and adding a new solution to recover the chunks affected by segment boundaries. Therefore, before implementing the design on FPGA devices, it is necessary to check

TABLE 2. List of datasets used in the study.

Dataset	Descriptions	Size (bytes)
Dataset 1: Operating system installation images	Linux Mint 20.3 Cinnamon 64-bit Debian 11.3 AMD64 Archlinux 2022.06.01 x86 64-bit	3 551 180 800
Dataset 2: Network traffic	Data capture files from Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC) 2012	3 451 850 240
Dataset 3: Media files	25 random MP4 video files	3 360 570 368

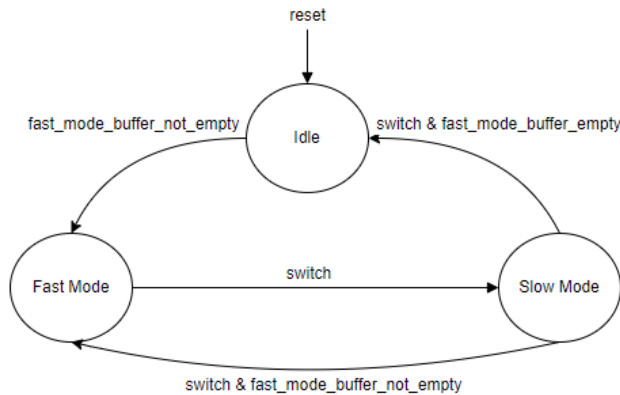

FIGURE 12. Marshalling Module requests for data from Slow Mode Buffer and Fast Mode Buffer.

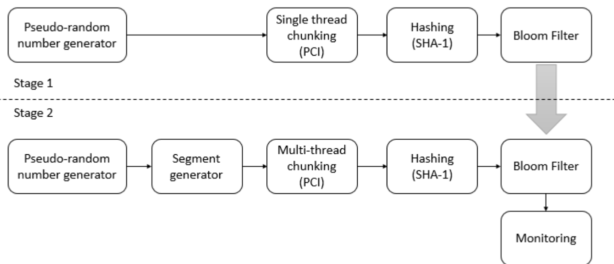

FIGURE 13. Testing model for the multi-thread chunking system.

the correctness of the parallel chunking method with software languages. Thanks to the considerable cache memory of our test system, we can temporarily ignore the overflow issue and verify the multi-thread chunking results with single-thread generated chunks. The single-thread PCI algorithm firstly processes the testing datasets. The results are hashed and updated into the Bloom Filter [20]. The same datasets are divided into segments that are overlapped *W-1* bytes with each other. These segments are then processed by our parallel chunking method, and the generated chunks will be compared with single-thread results through a sharing Bloom Filter. Several hash functions calculate the input chunk fingerprints of the Bloom Filter. Each function returns a different value that points to the corresponding position of the filter array, and its value will be changed from 0 to 1. The input chunk is new if one of the pointed positions is not set to 1. Otherwise, this chunk might already exist [20].

This testing model can be applied for verifying hardware design. Firstly, we chunk the datasets sequentially and update the Bloom Filter array. Then, the testbench reads the datasets, prepares segments and passes them into the design under test. The chunks calculated by hardware design will be hashed and checked for their existence by the single-thread generated Bloom Filter. For behavioral verification, the size of the buffers will be set large enough to ignore the overflow problems. However, the buffer size must be set to an appropriate value to achieve high throughput and good scalability in practice.

IV. IMPLEMENTATION AND EVALUATION

This section performs experiments to examine the influence of the buffer size on the chunk inconsistency. Moreover, we discuss and evaluate the proposed design implemented on FPGA Cyclone V 5CSXFC6D6F31C6.

A. EXPERIMENTAL DATASETS

We choose three real-world datasets with various data types for experiments. The first dataset includes Linux installation images from different vendors [21]–[23]. The second one, the network traffic dataset, contains data capture files from Mid-Atlantic Collegiate Cyber Defense Competition in 2012 [24]. Because the total captured data is enormous, we only used a subset of approximately 3.5 GB of data. Finally, the third dataset consists of 25 random MP4 video files. Table 2 lists explicitly the content of each dataset used in the study.

B. THE IMPACT OF THE BUFFER DEPTH ON THE CHUNK INCONSISTENCY

In our experiments, the PCI algorithm's window size W and preset threshold V are configured differently for each dataset to achieve the average chunk size of about 1 kB.

Besides the parameters of the algorithm, the size of the buffer also needs to be considered. In slow mode, the data will be stored entirely in the buffer before being read out by the Marshalling Module. If a chunk is larger than the buffer depth, the buffer will overflow and undesired chunks will be created. The chunks are defined as undesired chunks if they are created or affected by the buffer overflow. In other words, they are the chunks inconsistent with sequential chunking results.

As the maximum chunk size of a file is unpredictable, it is difficult to find an accurate buffer size that eliminates the overflow problem. When parallel processing is performed, we must accept that some undesired chunks will be created.

TABLE 3. Comparisons between the PCI algorithm and the proposed method implemented on FPGA.

Dataset	Chunking method		Total chunks	Duplicated bytes	Undesired chunks
Dataset 1 (W = 5, V = 31)	PCI		2 283 359	178 822 602	0
	Proposed method	2 segments	2 283 359	178 822 602	0
		500 segments	2 283 472	178 199 365	226
		1000 segments	2 283 586	177 680 014	454
Dataset 2 (W = 5, V = 30)	PCI		3 090 635	178 287 661	0
	Proposed method	2 segments	3 090 636	178 287 661	2
		500 segments	3 090 869	178 287 661	468
		1000 segments	3 091 103	178 287 661	936
Dataset 3 (W = 5, V = 30)	PCI		3 638 227	196 230	0
	Proposed method	2 segments	3 638 227	196 230	0
		500 segments	3 638 243	196 230	32
		1000 segments	3 638 263	196 230	72

However, the buffer depth can affect the number of unwanted chunks. With the average size of data chunks about 1 kB, we examine the number of new chunks created when the buffer depth is set at 1024. Then, we gradually increase the buffer size. The test results are shown in Fig. 14.

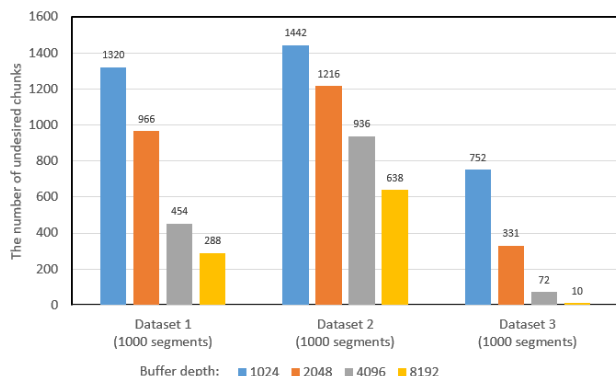

FIGURE 14. The number of undesired chunks when performing parallel chunking process with different buffer sizes.

The number of new chunks is quite high when the buffer depth is close to the average chunk size and this number decreases as the buffer size increases. When we set the buffer depth roughly the same as the average chunk size, it means about half of the chunks are larger than the buffer. Since each long chunk is cut by half whenever the buffer overflows, we could see that above 70% segment boundaries of dataset 1 and dataset 2 create undesired chunks in the case of the buffer depth of 1024. The number of undesired chunks in media file datasets is lower than in two other datasets but the proportion of undesired chunks is still high, at nearly 40%. The quantity of undesired chunks drops significantly with the buffer depth of 8192. However, such a buffer size is quite large and will encounter hardware resource constraints if the design scales up. The buffer depth of 4096 results in slightly more unwanted chunks but this setting reduces half of the internal memory consumption. In the case that the parallel design supports up to 1 kB average chunk size, the buffer depth of 4096 is a reasonable choice as it is not only hard to overflow but also saves more hardware resources.

C. THE IMPACT OF THE INPUT SEGMENT QUANTITY ON THE CHUNK INCONSISTENCY

The undesired chunks are created when chunks processed in slow mode are too long and make the Slow Mode Buffer overflow. This means the number of input segments affects the number of undesired chunks, not the number of threads. We configure the buffer depth of 4096 and implement experiments with various input segment quantities to observe the influence on the number of undesired chunks. Because the number of threads does not affect the number of undesired chunks, the experiments are only executed with a 2-thread design. Each dataset is divided into two equal segments, which is the minimum quantity of input segments in multi-thread implementation, to show the effect of the small number of segments on undesired chunks. Then, we increase the number of segments to 500 and 1000 segments to observe the growth of the undesired chunk quantity.

As can be seen in Table 3, the undesired chunks rarely appear when the number of input segments is minimum. Even if the undesired chunks are created, the quantity is minimal. When the number of segments increases, the undesired chunk quantity also increases. Because at least two undesired chunks are created when the Slow Mode Buffer overflows, the total chunks rise as the number of undesired chunks increases.

Suppose a long duplicated chunk in the dataset causes the Slow Mode Buffer overflow. In that case, that chunk is divided into two undesired chunks, and our proposed design finds fewer duplicated bytes. These cases happen in experiments with dataset 1. With 500 and 1000 input segments, the deduplication performance of the proposed implementation reduces by 0.35% and 0.64%, respectively, compared to the original algorithm. The loss of deduplication performance rises when the number of segments increases. In experiments with dataset 2 and dataset 3, the deduplication performance of the proposed design is the same as that of the PCI algorithm because long duplicated chunks do not create the undesired chunks.

D. THE IMPACT OF THE BUFFER SIZE ON THE CHUNKING THROUGHPUT

Because each Marshalling Module reads data from 2 buffers of different threads, the number of data processed by each

thread might not be the same. A Marshalling Module bypasses data read from the Fast Mode Buffer of the same thread and processes data from the Slow Mode Buffer of the following thread. The total bytes processed by a Marshalling Module might be higher than the total bytes of input segments of that thread. The variance of processed data between Marshalling Modules results in the throughput reduction. In this section, we discuss the influence of the buffer size on the design throughput. We run the proposed implementation with the parallel thread quantity of 2, 4, 8, 16 and 32. The buffers are configured with various sizes to observe the impact on the chunking throughput.

Fig. 15, Fig. 16, Fig. 17, Fig. 18 and Fig. 19 illustrate the experimental results of the proposed design with various thread quantities. The chunking throughput decreases as the buffer size increases. However, because the buffer size is small compared with a segment size, the variance of processed data between threads is also small. Consequently, the chunking throughput decreases slightly when the buffer depth changes from 1024 to 8192.

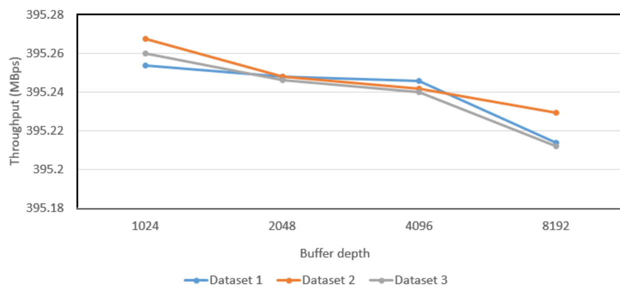

FIGURE 15. The throughput of 2-thread implementation with various buffer depth configurations.

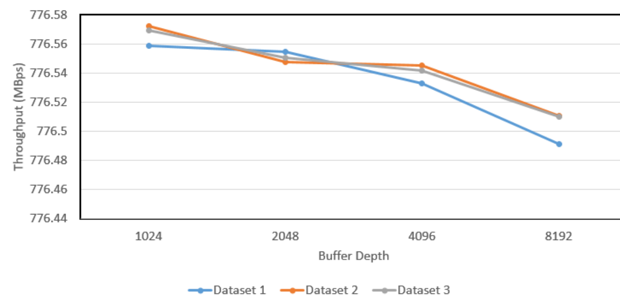

FIGURE 16. The throughput of 4-thread implementation with various buffer depth configurations.

E. HARDWARE EVALUATION

We use Quartus software to synthesize the design with the FPGA Cyclone V 5CSXFC6D6F31C6. After setting the depth to 4096 for the buffers in the design, the synthesis results of the parallel chunking design are shown in Table 4.

The results show that the proposed design can operate at high frequencies, nearly 200 MHz. As the design scales up

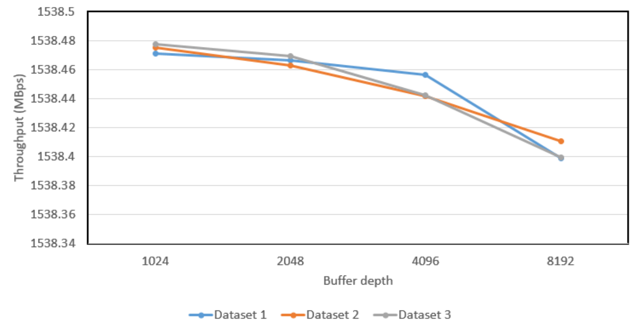

FIGURE 17. The throughput of 8-thread implementation with various buffer depth configurations.

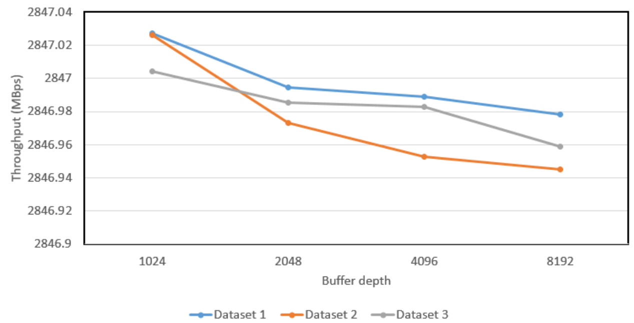

FIGURE 18. The throughput of 16-thread implementation with various buffer depth configurations.

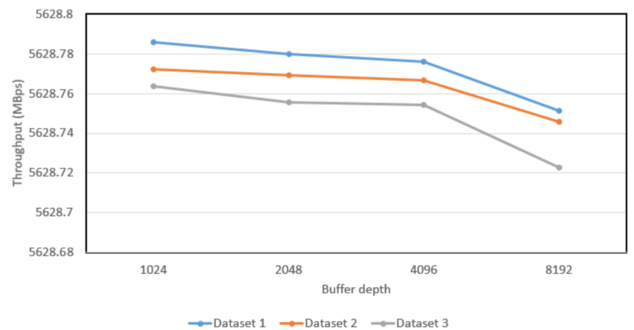

FIGURE 19. The throughput of 32-thread implementation with various buffer depth configurations.

to 8 threads, the hardware resource consumption increases approximately linearly. However, the clock rates are not considerably changed thanks to the low logic utilization of individual threads. The designs with more than 16 parallel threads have more noticeable changes in operating clock frequency due to the increase in overall resource consumption. While our design costs a few logic resources and registers, it requires many BRAMs for building buffers. Consequently, the number of BRAMs in the FPGA will be used up while leaving a lot of logic cells and registers if the design continues to expand. By trading off the BRAM resource costs against the number of undesired chunks, the design can continue to scale up to

TABLE 4. Hardware design synthesis results for Intel FPGA Cyclone V.

Number of parallel threads	Maximum frequency (MHz)	Resource utilization		
		BRAM (M10K)	Register	Logic utilization (ALM)
2	197.63	20	1162	497
4	194.14	40	2317	999
8	192.31	80	4640	1991
16	177.94	160	9269	3986
32	175.90	320	18526	7962

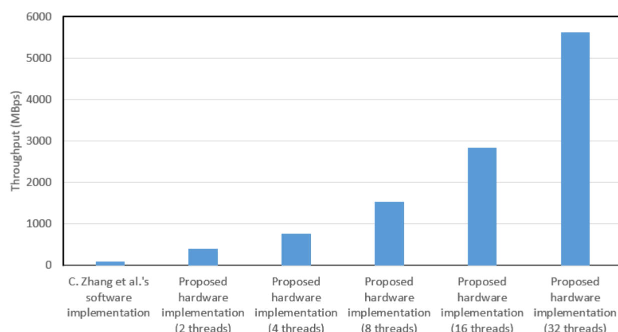

FIGURE 20. Chunking throughput comparisons.

achieve higher throughput.

$$Throughput = f_{max} \times threads \quad (1)$$

C. Zhang *et al.* performed experiments with datasets generated by the Mersenne Twister Pseudo-Random Number Generator [25] to evaluate the chunking speed of the PCI algorithm. The experimental results show that the throughput of the PCI algorithm was about 90MBps with software implementation [14]. In this paper, because each chunking thread calculates a byte per cycle, the hardware implementation throughput is calculated by multiplying the maximum frequency, denoted as f_{max}, and the number of parallel threads. The calculated results shown in Fig. 20 indicate that our proposed design with only two parallel threads could process roughly four times faster than the sequential software implementation. Moreover, as our chunking design on FPGA devices is scalable, the chunking throughput increases approximately linearly with the number of processing threads.

V. CONCLUSION

In this study, we investigate several Content-defined Chunking algorithms, evaluate the advantages and disadvantages, and explain why PCI is a suitable algorithm for our parallel chunking system. A new marshalling method is proposed to improve the multi-thread chunking system performance. We also implement a FPGA-based design based on the PCI algorithm and our marshalling method. The results show that the parallel hardware design in this study significantly improves the data chunking speed. The scalability of the design is also a prominent advantage when implemented on FPGA devices, nevertheless, it cannot maintain the chunk consistency. Another drawback is that the generated chunks are out of order at the system output. This disadvantage makes it difficult for the indexing and storage management process in a data deduplication system and will need further research in the future.

ACKNOWLEDGMENT

The authors acknowledge Ho Chi Minh City University of Technology (HCMUT), VNU-HCM for supporting this study.

REFERENCES

[1] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proc. IEEE*, vol. 104, no. 9, pp. 1681–1710, Sep. 2016.

[2] S. Singhal, P. Sharma, R. K. Aggarwal, and V. Passricha, "A global survey on data deduplication," *Int. J. Grid High Perform. Comput.*, vol. 10, no. 4, pp. 43–66, Oct. 2018.

[3] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplication-large scale study and system design," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2012, pp. 285–296.

[4] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Trans. Storage*, vol. 7, no. 4, pp. 1–20, Jan. 2012.

[5] S. Quinlan and S. Dorward, "Venti: A new approach to archival data storage," in *Proc. Conf. File Storage Technol. (FAST)*, 2002, pp. 1–13.

[6] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. Fast*, vol. 8, Feb. 2008, pp. 269–282.

[7] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proc. 18th ACM Symp. Operating Syst. Princ.*, Oct. 2001, pp. 174–187.

[8] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "WAN-optimized replication of backup datasets using stream-informed delta compression," *ACM Trans. Storage*, vol. 8, no. 4, pp. 1–26, Nov. 2012.

[9] K. Eshghi and H. K. Tang, "A framework for analyzing and improving content-based chunking algorithms," Hewlett-Packard Labs, Palo Alto, CA, USA, Tech. Rep. TR, 2005, vol. 30.

[10] M. O. Rabin, *Fingerprinting by Random Polynomials*. Cambridge, MA, USA: Center for Research in Computing Technology, Harvard Univ., 1981.

[11] N. Bjørner, A. Blass, and Y. Gurevich, "Content-dependent chunking for differential compression, the local maximum approach," *J. Comput. Syst. Sci.*, vol. 76, nos. 3–4, pp. 154–203, May 2010.

[12] Y. Zhang, H. Jiang, D. Feng, W. Xia, M. Fu, F. Huang, and Y. Zhou, "AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 1337–1345.

[13] R. N. S. Widodo, H. Lim, and M. Atiquzzaman, "A new content-defined chunking algorithm for data deduplication in cloud storage," *Future Gener. Comput. Syst.*, vol. 71, pp. 145–156, Jun. 2017.

[14] C. Zhang, D. Qi, W. Li, and J. Guo, "Function of content defined chunking algorithms in incremental synchronization," *IEEE Access*, vol. 8, pp. 5316–5330, 2020.

[15] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Z. Wang, ''P-dedupe: Exploiting parallelism in data deduplication system,'' in *Proc. IEEE 7th Int. Conf. Netw., Archit., Storage*, Jun. 2012, pp. 338–347.

[16] Y. Won, K. Lim, and J. Min, ''MUCH: Multithreaded content-based file chunking,'' *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1375–1388, May 2015.

[17] P. Bhatotia, R. Rodrigues, and A. Verma, ''Shredder: GPU-accelerated incremental storage and computation,'' in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, p. 14.

[18] D. Li, Q. Yang, Q. Wang, C. Guyot, A. Narasimha, D. Vucinic, and Z. Bandic, ''A parallel and pipelined architecture for accelerating fingerprint computation in high throughput data storages,'' in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, pp. 203–206.

[19] F. Ni, X. Lin, and S. Jiang, ''SS-CDC: A two-stage parallel content-defined chunking for deduplicating backup storage,'' in *Proc. 12th ACM Int. Conf. Syst. Storage*, May 2019, pp. 86–96.

[20] B. H. Bloom, ''Space/time trade-offs in hash coding with allowable errors,'' *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[21] *Linux Mint 20.3 'una' Linux Mint*. Accessed: Jun. 26, 2022. [Online]. Available: https://linuxmint.com/edition.php?id=292

[22] *Arch Linux Downloads*. Accessed: Jun. 26, 2022. [Online]. Available: https://archlinux.org/download/

[23] *Installing Debian Via the Internet*. Accessed: Jun. 26, 2022. [Online]. Available: https://www.debian.org/distrib/netinst.en.html

[24] *Pcap Files From the US National Cyberwatch Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC)*. Accessed: Jun. 26, 2022. [Online]. Available: https://www.netresec.com/?page=MACCDC

[25] M. Matsumoto and T. Nishimura, ''Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator,'' *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, Jan. 1998.

HUNG VUONG received the B.S. degree in electronics and telecommunications engineering from the Ho Chi Minh City University of Technology (HCMUT), Vietnam, in 2019. He is currently pursuing the M.S. degree in electronics engineering with the Ho Chi Minh City University of Technology–VNU HCM.

HUNG NGUYEN received the B.S. and M.S. degrees in electronics and telecommunications engineering from the Ho Chi Minh City University of Technology (HCMUT), Vietnam, in 2019 and 2022, respectively. Currently, he is working with the Faculty of Electrical-Electronics Engineering, Ho Chi Minh City University of Technology–VNU HCM.

LINH TRAN received the B.S. degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign, in 2005, and the M.S. and Ph.D. degrees in computer engineering from Portland State University, in 2006 and 2015, respectively. Currently, he is a Lecturer with the Faculty of Electrical-Electronics Engineering, Ho Chi Minh City University of Technology–VNU HCM. His research interests include quantum/reversible logic synthesis, computer architecture, hardware-software co-design, efficient algorithms and hardware design targeting FPGAs, and data analysis.