Received September 12, 2020, accepted September 27, 2020, date of publication October 8, 2020, date of current version October 19, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3029691

An In-Sight Into How Compression Dictionary Architecture Can Affect the Overall Performance in FPGAs

MATĚJ BARTÍK[1,2], (Member, IEEE), TOMÁŠ BENEŠ[1], (Student Member, IEEE), AND PAVEL KUBALÍK[1]

[1]Department of Digital Design, Faculty of Information Technology, Czech Technical University in Prague, 160 00 Prague, Czech Republic
[2]Department of Technology for Network Applications, CESNET, 160 00 Prague, Czech Republic

Corresponding author: Matěj Bartík (bartimat@fit.cvut.cz)

This work was supported in part by the Czech Technical University Project Design, Programming and Verification of Embedded Systems under Grant SGS20/211/OHK3/3T/18, and in part by the project E-infrastructure CESNET – Modernization under Grant CZ.02.1.01/0.0/0.0/16013/0001797.

ABSTRACT This paper presents a detailed analysis of various approaches to hardware implemented compression algorithm dictionaries, including our optimized method. To obtain comprehensive and detailed results, we introduced a method for the fair comparison of programmable hardware architectures to show the benefits of our approach from the perspective of logic resources, frequency, and latency. We compared two generally used methods with our optimized method, which was found to be more suitable for maintaining the memory content via (in)valid bits in any mid-density memory structures, which are implemented in programmable hardware such as FPGAs (Field Programmable Gate Array). The benefits of our new method based on a ''Distributed Memory'' technique are shown on a particular example of compression dictionary but the method is also suitable for another use cases requiring a fast (re-)initialization of the used memory structures before each run of an algorithm with minimum time and logic resources consumption. The performance evaluation of the respective approaches has been made in Xilinx ISE and Xilinx Vivado toolkits for the Virtex-7 FPGA family. However the proposed approach is compatible with 99% of modern FPGAs.

INDEX TERMS Compression algorithm, compression dictionary, FPGA, hash table, LZ4, LZ77, memory architecture, performance comparison, status register.

I. INTRODUCTION

Lossy or lossless high-speed and low-latency compression is important for many applications in real-time networking, video transmissions or disk storage. The research in lossless compression has led to the development of new types of devices that perform compression in real-time. Over the last decade, the throughput of these devices has increased up to 44.8 Gbps (Gigabit per second) [1] from a gigabit speed. The progress has been made by improving designs step by step with new techniques or tweaks that have made these designs more efficient in terms of speed, logic resources utilization or compression ratio. We focus on the improvement of the specific area of compression algorithms – compression dictionaries and how to increase their overall performance.

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

II. BRIEF MOTIVATION AND CONTRIBUTIONS

In this paper, we focused on exploring new ways of improving the overall performance of hardware-implemented compression algorithms. We put emphasis on these design properties:

- maximum throughput,
- maximum frequency,
- resources utilization,
- computation latency,
- predictability.

The mentioned design properties have no direct impact on the compression ratio; however, they may have an indirect effect, such as using an original amount of logic resources to implement a larger dictionary when a resource-efficient architecture was selected over the original one (due to saved resources).

In some specific use cases, the compression ratio may be less important than the design throughput, latency, and predictability. These requirements are considered to be fundamental for Real-Time systems [2]. To achieve a great compression ratio, some sophisticated techniques for a match search are usually used (re-hashing principle, for example). On the other hand, these sophisticated techniques introduce variability in data processing, which results in stalls in a compression engine datapath; thus, computation latency is variable and unpredictable.

Several hardware architectures focused on maximizing the design throughput have been introduced in the past few years [1], [3], [4]. On the other hand, no current architecture has emphasized lowering an architecture overhead to reduce the respective architecture computation latency or resource utilization. It has been identified [5] that the overhead (a compression dictionary initialization) necessary for the compression process (computation phase) could require more time than the compression itself.

Our goal is to reduce the computation latency and resource utilization while increasing the operating frequency by improving the compression dictionary architecture.

The contributions of the paper are:

- We analyzed three existing techniques suitable for implementing a compression algorithm dictionary, including our new technique [5].
- We created a new methodology for evaluating the analyzed techniques.
- We performed an experimental test to obtain results.
- A conclusion has been made that our method (''Distributed Memory'') shows better results than the other techniques in terms of maximum frequency, computation latency, and amount of required logic gates for our specific use case. This statement has been supported by quantitative analysis and experimental results.

III. THEORETICAL BACKGROUND

Implementations of lossless compression algorithms in hardware (in both FPGAs and ASICs – Application Specific Integrated Circuit) appeared right after the moment when software implementations were unable to satisfy the desired performance requirements such as throughput or latency. In the last two decades, a device realizing real-time compression of network communication using IP (Internet Protocol) principles became a widespread use case. The authors would like to summarize the properties of such implementations [1], [3], [7]–[32] as follows:

- The majority of designs are based on the LZ77 algorithm [33] or derived algorithms such as LZ78 [34] or LZW [35].
- The latest designs experiment with new derived algorithms focused on better compression ratio (LZMA [12]–[14], [36]) or speed (LZ4 [4], [31], [32], [37]).
- The compression speed is improved by massive pipelining or parallelization (systolic arrays) [24] of the match searching mechanism [38].
- There is a direct proportion between the compression ratio and the size of a compression dictionary. However, most of the mentioned implementations use (FPGA) embedded memory blocks (kilobytes in size) rather than external memory [8] such as DRAM (Dynamic Random Access Memory) or SRAM (Static Random Access Memory) chips.
- Compression dictionaries use three fundamental approaches: CAM (Content Addressed Memory) [39], hash table [40], and small (shift) register array for stream operating implementations [9], [17], [32], where the dictionary stores a few processed data words.
- Many implementations have small (size of kilobytes on average) input/output buffers optimized towards a block-oriented compression that makes them suitable for IP packet oriented compression [31].

A representative example of a hardware implementation of a lossless compression algorithm has the following features: It is based on LZ77 with massive parallelization of a match search mechanism with particularly small data/compression dictionary buffers.

A. LZ77 PRINCIPLES AND THE IMPACT OF THE DICTIONARY

LZ77 is a universal compression algorithm that is asymmetrical (the compression requires more time or resources than decompression) and single pass (data to be compressed are processed only once). LZ77 is a fundamental lossless compression scheme used in many further algorithms such as DEFLATE [41] or GIF (Graphics Interchange Format). The technique of the ''Sliding Window'' [6] for searching match candidates is used by the LZ77 algorithm (see Fig. 1).

The sliding window is usually divided into a search buffer (a dictionary) and a look-ahead buffer. The longest found prefix of the look-ahead buffer starting in the search buffer is encoded as a triplet **(i, j, X)**, where **i** is the distance of the beginning of the found prefix from the end of the search buffer; **j** is the length of the found prefix; and **X** is the first character after the prefix in the look-ahead buffer. The size (and the architecture) of the dictionary has a great influence on the compression ratio. A larger or better organized dictionary improves the compression ratio of the implemented compression algorithm because of the increased probability of finding a match over the larger sliding window [5].

There are three most common architectures of a dictionary. We would like to summarize their advantages and disadvantages from the perspective of their suitability for IP packet compression.

1) SHIFT REGISTERS FOR STREAM OPERATING IMPLEMENTATIONS

This type of a dictionary focuses on maximum performance in terms of operating frequency and the implementation architecture is carefully designed to process data in a (deep) pipeline to achieve maximum throughput. This seems to be an optimal solution for IP packets aware compression (a continuous stream of IP packets) with minimal latency [9], [32], but the compression ratio is quite low compared to

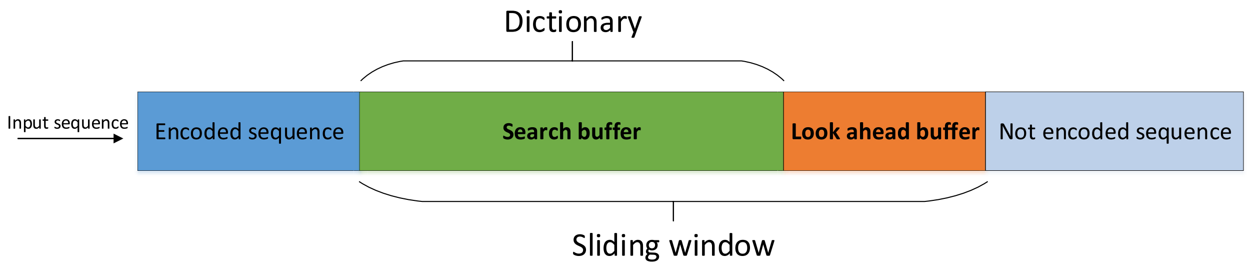


FIGURE 1. **LZ77 sliding window technique. [6].**

other approaches. The depth of the pipeline limits the sliding window of the dictionary to several words of processed data. The dictionary content does not have to be initialized because the match is only being searched in the pipelined data.

2) CONTENT ADDRESSED MEMORY (CAM)

The CAM based approach utilizes data spacial locality where the dictionary can accommodate more entries if the processed data are highly repetitive. The disadvantage is that each CAM memory cell is usually implemented by flip-flops and requires its own comparator for searching a match, requiring many logic resources. Therefore the enormous logic consumption consequently slows down the entire design by reducing the maximal operating frequency of such design [42].

Several techniques [42], [43] were introduced to improve some design properties such as reducing the amount of (some) required logic resources via the usage of other design primitives like embedded memory block. Another disadvantage is a latency of five clock cycles for a search operation.

On the contrary, a CAM based dictionary could easily be initialized via a dedicated reset/clear input of these flip-flops. Overall, we found the usage of CAM based dictionary not viable for our specific case requiring low-latency operations.

3) HASH TABLE

The hash table principle became popular when fast, modern compression algorithms (LZ4 [37], LZO [44]) appeared. The common idea of these algorithms is improving the throughput by increasing the width of the processed data word (the word width is 32 or 64 bits to match the ALU (Arithmetic Logic Unit) register width in modern processors [45]). These word widths are too large to be used as a direct address to the dictionary (the dictionary will have 4 gigabytes for the 32-bit word width). The CAM technique will make these algorithms slower [46] but fairly large dictionaries became required for a decent compression ratio. This led to implementing the dictionary as a hash table [1], [3]. The important features of a hash table implementation are the following:

- The hash algorithm can be extremely fast (just a constant multiplication in LZ4 [31], [47], the result is trimmed to an appropriate number of address bits to match the dictionary size).
- Produced hashes can collide with each other reducing the compression ratio a little (but saving memory required for the dictionary).
- Dictionaries are usually implemented in embedded memory blocks. In our particular example, the used Xilinx BlockRAMs are RAM based blocks with densities of 36 kilobits [48]. The content (a dictionary) in embedded/DRAM memory cannot be cleared in a single clock cycle [48], [49] like flip-flop (SRAM) based memory. This embedded memory block design is a trade-off between the memory capacity and the number of transistors required for the memory cell matrix [48].
- IP Packet optimized designs require clearing the entire dictionary before each run of the implemented compression algorithm (each IP packet is considered as one block).

B. REQUIREMENTS FOR THE DICTIONARY DESIGN

The requirements are set with emphasis to the particular use case: the IP protocol packet compression accelerators implementing LZ77 algorithm.

- The dictionary design should be suitable for IP packet compression (block compression oriented).
- The maximum payload will be 9 kB (the maximum size of a jumbo packet) [50].
- 10 Gbps throughput requirement leads to a 64-bit datapath clocked at 156.25 MHz at least because a design with 8-bit datapath will require a 1.25 GHz system clock which is significantly above the FPGA limits.
- The time required for loading the processed data from the buffer is 1150 clock cycles in the worst case.
- The dictionary size should be in the range of 1k–16k of entries (larger dictionary makes no sense compared to input/output buffer size).
- The dictionary will be implemented as a hash table. Therefore, we have to deal with the problem of potentially slow (re-)initialization.

The problem to be solved: the dictionary will be implemented as embedded/DRAM based memory. We have to find an efficient method (in terms of time) for (re-)initialization of the dictionary content. The efficiency in terms of logic resources can lead to a trade-off with a time sub-optimal solution.

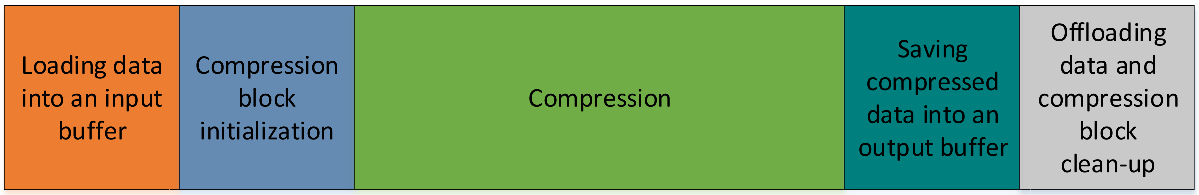

Loading data into an input buffer
Compression block initialization
Compression
Saving compressed data into an output buffer
Offloading data and compression block clean-up

FIGURE 2. Common phases of a hardware implemented compression algorithm.

C. COMPUTATION TIME OF A COMPRESSION HARDWARE BLOCK

A general (hardware) implementation of a compression algorithm has several phases where most of them are not originally related to the compression itself. However, they are needed for proper operation of the compression block. Some of these phases (see Fig. 2) can overlap each other because they were implemented [1], [3], [17], [26] in a smart way. Further details about each phase follow:

1) LOADING DATA INTO AN INPUT BUFFER

The time required for storing data into the input buffer is dependent on the type of application, and on the size and throughput of the respective buffer. The (maximum) required time [51] can easily be determined as a ratio between the throughput and the size. This phase can run in parallel with the next phase (initialization); however this phase is essential for proper operation of the compression block.

2) COMPRESSION BLOCK INITIALIZATION

The initialization of the compression block is intended to set-up default values of design registers or any other data structures like a compression dictionary, acquiring the size of the data, etc.

Despite the fact that the initialization phase can run in parallel with the data loading phase, the computational time of this phase is heavily affected by the overall architecture of the particular design. Initialization of complex data structures (used by a compression dictionary, for example) could easily be more time consuming than the data loading phase.

3) COMPRESSION

The most important phase is the compression itself, which is supposed to search for matches in a compression dictionary and encode respective output with particular examples of LZ based algorithms [38]. The maximum computation time can be estimated as the ratio between the input buffer size and the respective throughput of the compression phase. The computational time could be lower than the time of the initialization phase in our particular case (see Section III-B).

4) SAVING COMPRESSED DATA TO AN OUTPUT BUFFER

This phase is intended to store the compressed data into an output buffer. This functionality is usually implemented in the compression phase, therefore, these phases can overlap. In certain situations, the compression ended, but some data are still not copied to the output buffer. It is obvious that the computational time will be low.

D. SUMMARY

The conclusion is quite simple – calculating the overall latency is not simple, because it involves latencies of some other phases besides the compression phase as the primary function. The time (latency) required for transferring the processed data to/from input/output buffers has the same lower bound asymptotic complexity as the compression itself, the $\Omega(n)$. The question is, which compression dictionary architecture can match or decrease such lower bound asymptotic complexity, especially when the required dictionary can be larger than the buffers [5]?

IV. STATE OF THE ART – EVALUATED METHODS FOR INITIALIZING A DRAM BASED MEMORY STRUCTURES

We have selected a hash table for implementing a dictionary for a lossless compression algorithm. The choice has been made based on the analysis in the previous chapter. We are looking for a design for IP packet compression based on LZ77 with a minimum throughput of 10 Gbps per implemented block for applications in 10 gigabit ethernet networks. We put emphasis on the latency of the dictionary (re-)initialization phase.

We assume optimizations and techniques introduced by modern fast lossless compression, such as LZ4, can improve the ratio between logic gates count and throughput. This might allow implementing multiple compression blocks in a single FPGA. In the following sections, we will discuss three alternative techniques suitable for the hash table (implemented using BlockRAMs) based compression dictionary architecture. These techniques can be used in other architectures that are also BlockRAM based.

A. LINEAR PASSAGE APPROACH

The linear memory passage is a fundamental method for initializing memory to a default (constant) value [7].

The fundamental part is a counter with the same width as the memory address vector, thus all addresses are generated (including other control signals like write enable) for the (re-)initialization purpose. The data input port (vector) is multiplexed by the default value during the (re-)initialization process.

Advantages of the linear passage method are a simple and straightforward design and low resource requirements.

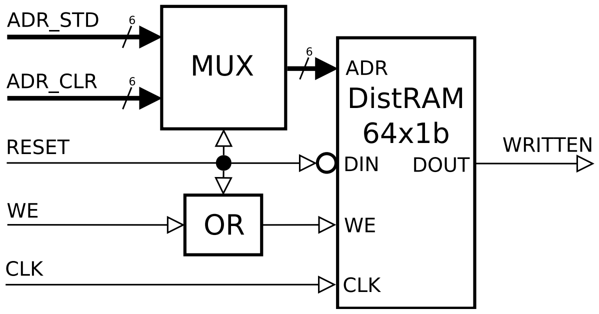


FIGURE 3. Architecture of the "Elementary Block". [5].

Disadvantages are that the initialization process requires a lot of clock cycles to pass through all addresses and that the adder used by the counter has a long carry chain that limits the maximum design frequency. These two disadvantages increase the latency of the design. This method is the only suitable approach for clearing a high density memory (with millions of entries) such as entire (external) DRAM [52] chips.

B. FLIP FLOP BASED APPROACH FOR A STATUS REGISTER

The second approach [53] uses a flip-flop based status register file, where each flip-flop preserves a single-bit wide flag indicating the status of the related memory cell (written or not written). When a memory read occurs before a write operation, the memory output will be multiplexed to the default read value.

Advantages of the flip-flop based approach are a higher operating frequency than with the linear passage and the latency of one clock cycle, because all flip-flops can be effectively cleared in parallel. Disadvantages are the exponential growth of required resources with respect to the memory address vector width. When a large memory is used, the operating frequency will drop significantly due to FPGA routing and synthesis issues.

C. DISTRIBUTED MEMORY BASED APPROACH FOR A STATUS REGISTER

We proposed an approach [5], [54] that combines previous approaches to get as many advantages (like the same asymptotic complexity [51] requiring less FPGA resources than the flip flop based approach) and to mitigate as many disadvantages from both techniques. The idea is to use an alternative way of storing data in an FPGA-based design instead of ordinary flip-flops, using the distributed memory block [55] in our particular case. This approach is comparable to the LVT (Live Value Table) [56], [57], where the idea is to split the design into two parts. Each part uses a different type of memory (flip-flop & BlockRAM) instead of a single type (flip-flop).

The flip flop based array of single-bit wide flag registers is split into small segments with the same size as a single distributed memory block (64-bits for the Xilinx 7-series architecture). The distributed memory block based design is divided into two parts: the "Elementary Block" (EB) and the "Address Control Logic" (ACL).

1) ELEMENTARY BLOCK

The EB (see Fig. 3) is composed of one distributed memory block with the size of 64 individual bits (the maximum size for a single 6-input LUT (Look-Up Table) [55] implementation). The distributed memory block has to be cleared (initialized) by the linear passage approach requiring 64 clock cycles. The linear passage approach also requires a multiplexer for switching address vectors between the standard and initialization mode (selected via the reset signal). The standard address vector input of the EB is the last (lowest) six bits of the address range for the status register. The second (initialization mode) address vector input is dedicated to the logic of the linear passage of the ACL block. The "Written" signal represents the information indicating whether a particular memory cell had a write request and the related record in a dictionary contains valid data. The default value for the initialization of the "Written" signal is logic zero (therefore, all bits in the distributed memory block).

2) ADDRESS CONTROL LOGIC

The ACL architecture (see Fig. 4) shows four individual parts of the status register:

- "CNT M64"– The 6-bits wide counter (counting as modulo 64) for generating the address vector for the elementary blocks while the initialization mode is active.
- "SPLIT"& "EB SEL" – The "SPLIT" block splits the address vector input into the upper and lower part. The lower part has six bits to match the address range of the EB. The upper part is forwarded to "EB SEL" implementing an address decoder. The address decoder is generating the "Chip Enable" (CE) encoded as one-hot value for each EB in the design. Consequently, only one EB is selected at each clock cycle.
- Output Masking – Only the output of the chosen EB is passed to the "Written" signal via AND/OR logic gates. Outputs of rest EBs are masked.

V. A QUANTITATIVE ANALYSIS

This section presents a discussion about which design parameters have an impact on such compression dictionary design. The general observed properties for a hardware accelerator implementing a compression algorithm are:

- compression ratio,
- throughput,
- latency,
- operating frequency,
- amount of logic resources.

As stated earlier in Section III-A, the compression dictionary size significantly affects the respective compression ratio. In the case of a hardware accelerator, input/output buffers and compression dictionary are often implemented using embedded memory blocks (called BlockRAM/M9K

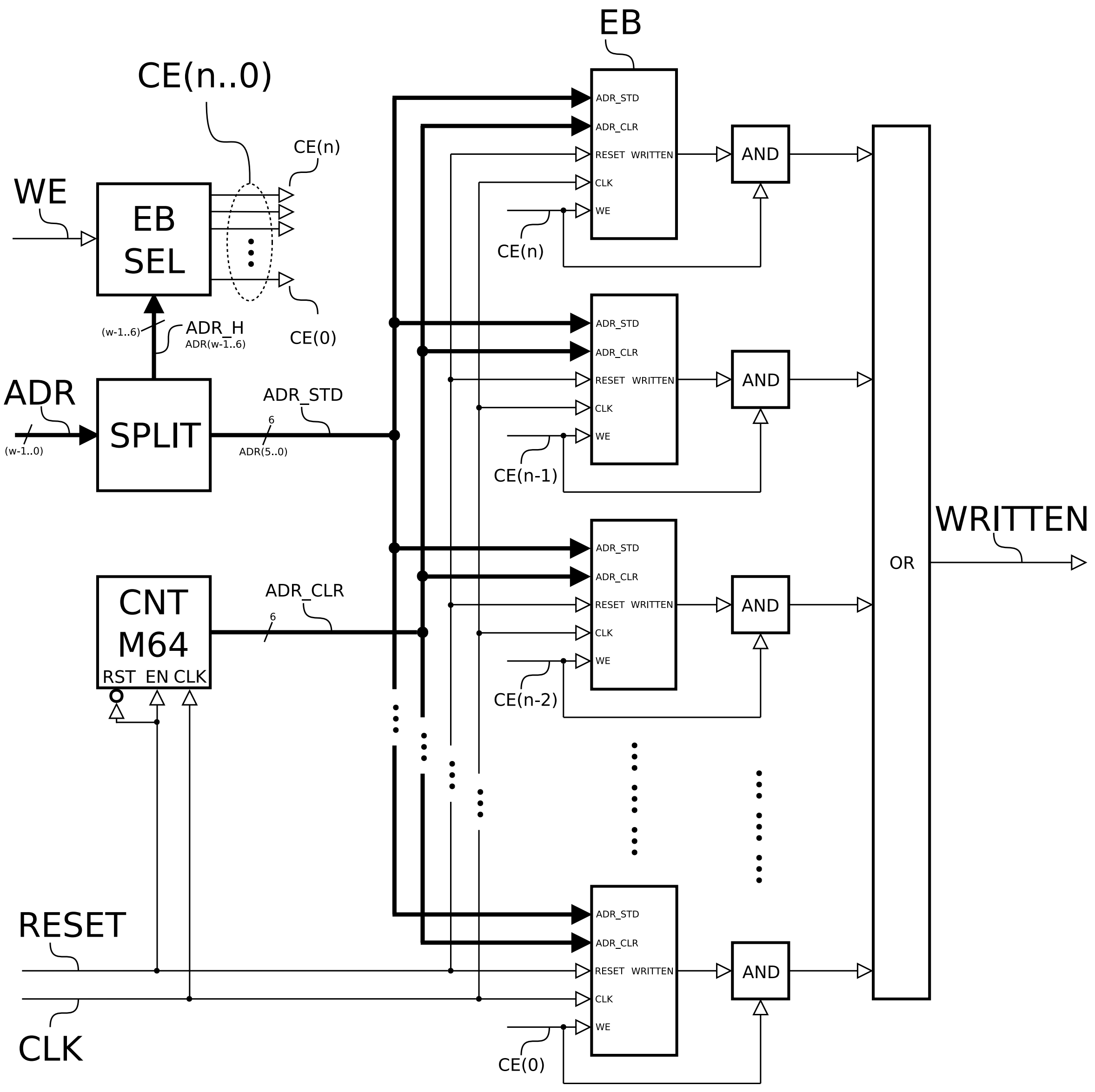


FIGURE 4. Architecture of the "Status Register" based on EBs and the ACL. [5].

in the case of Xilinx and Intel/Altera FPGAs, respectively). Therefore the sizes of (the example) dictionaries in our case are simulated by modifying the **"W"** parameter [5], which stands for a memory address bus width. The relation between memory capacity (which accommodates a compression dictionary, for example) and the address width can be expressed as formula (1). The formula assumes a digital system. Therefore, the address is a binary number, and the number of entries is also a power of two (cases where an address range does not match a number of entries are not considered because they are rare in digital design).

$$\begin{aligned} W &= log_2\left(\frac{Memory\ capacity}{Memory\ entry\ width}\right) \\ &= log_2(Number\ of\ memory\ entries) \end{aligned} \quad (1)$$

The operating frequency and the amount of used logic resources are also affected by the used FPGA type (technological parameters).

A. ARCHITECTURE INFLUENCE

The particular architecture of such a dictionary affects the remaining observed properties, which usually depend on the used FPGA. In general, the architecture complexity affects the number of logic resources needed for an implementation in hardware and properties as such the latency required for clearing them off. It is assumed that more complex architectures will require a higher amount of logic resources. There is no such assumption on (theoretical) architecture latency.

These logic resources have to be placed ("floor-planned") [58] in the 2D space of the integrated circuit and interconnected by wires. It is clear that a higher number of logic resources must occupy a larger area in a silicon. Therefore the higher the area occupied, the longer the respective wires will be, and they will contain more junctions.

Consequently, the increased wire lengths and increased logic gate output load will increase the signal propagation delay, and thus the maximum operating frequency of such accelerator will be limited by the ''slowest'' signal [59]. The reduced frequency will also reduce the respective throughput of a compression accelerator.

In the particular case of a hardware compression accelerator, it is a common attitude to design the accelerator to minimize the number of (system) clocks; only one clock signal is being used in most cases. Such a compression accelerator usually consists of several smaller ''building blocks'', for example: input and output buffers, compression dictionary, match search unit [38], encoding unit, etc. It is obvious that the lowest frequency of these ''building blocks'' will be the resulting operating frequency of the particular accelerator. Therefore, the motivation is to design an accelerator where all blocks are close to each other in terms of frequency to improve the overall accelerator frequency, thus improving the performance.

From this perspective, the architecture used for implementing a compression dictionary has an impact on the amount of required logic resources, thus frequency, and thus the overall accelerator performance. In case the respective architecture saves a lot of logic resources (against previously used compression dictionary architecture), it will increase the overall accelerator frequency. Despite the fact that there is no direct influence on the compression ratio, a more resource-efficient architecture could allow hardware designers to implement a bigger dictionary with the same amount of resources as it was originally, using to achieve a better compression ratio [5], [38]. On the other hand, the extra logic resources can also be used for implementing multiple accelerators with a higher overall throughput while keeping the same (constrained) area of an integrated circuit.

B. ESTIMATIONS

The amount of logic resources needed is affected by the **''W''** parameter and the capabilities of the used FPGA for this estimation. The frequency parameter is usually indirectly proportional to the amount of logic resources. The latency of initialization of a compression dictionary is architecture specific. The compression ratio parameter cannot be estimated in this particular case.

1) XILINX CONFIGURABLE LOGIC BLOCK (CLB) ARCHITECTURE

As an abbreviation, FPGA is quite self-explanatory. It is a giant array of fundamental blocks (CLBs [55] in the Xilinx case) interconnected by a matrix of wires (FPGA fabric) which can realize a desired logic function. This principle has been shared among all major FPGA vendors. CLB can be divided further into two slices. Each slice consists of four LUTs and eight flip-flops (registers) plus an interconnection fabric.

The most common LUT width is 6 bits in most FPGAs (LUT6). However, some older FPGAs had 4-bit LUTs only. Some LUTs have available alternative use cases such as distributed memory blocks or wide shift registers. The 6-bit LUT can usually be split further into two 5-bit LUTs, which are more suitable for implementing less complex logic functions. It seems a wider LUT is not going to be introduced by FPGA vendors in the near future. Not all elements in a CLB have to be utilized.

2) LINEAR PASSAGE APPROACH

As stated in the above text, the approach uses a counter (counter width is equal to the **''W''** parameter) generating all addresses (4), which is connected to the BlockRAMs address input via a multiplexer. The multiplexer switches the normal and reset operation addresses. The amount of logic resources can be estimated [59] in the following way: the counter will require at least **''W''** LUT5s and **''W''** registers. The multiplexer will require **''W''** LUT5s only for the implementation. Therefore the linear passage approach will likely require several LUTs (2) and registers (3) in total.

$$LUT_{Linear} = 2 * W \tag{2}$$
$$REG_{Linear} = W \tag{3}$$
$$Latency_{Linear} = 2^W. \tag{4}$$

3) FLIP-FLOP BASED APPROACH

This approach requires generating an array of registers equal to the number of entries in a dictionary (2^W in our case). An address decoder is also required to select the individual register during the operation. Therefore at least one LUT will be required for each register resulting in estimations (5) and (6)

$$LUT_{Flip-Flop} = \lceil log_6(W) \rceil * 2^W \tag{5}$$
$$REG_{Flip-Flop} = 2^W \tag{6}$$
$$Latency_{Flip-Flop} = 1. \tag{7}$$

4) DISTRIBUTED MEMORY BASED APPROACH

The numbers of required LUTs and registers are expressed in formulas (8) and (9); thus they can be described as a difference between the Distributed and the Flip-Flop based approach:

- Each EB replaces 64 flip-flops; therefore, the total number of EBs is 2^{W-6}.
- Individual address decoder for each EB is less complex because it decodes 6 fewer address bits, which are omitted by the SPLIT function.
- Each EB consists of 7 LUTs.
- The output masking function uses the 2-input AND logic gates, which can be packed into the OR logic gate ($log_6(W-6)$ originally).

TABLE 1. Estimated properties for all techniques depending on the "W" parameter.

LUTs										
Address width "W"	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Linear	12	14	16	18	20	22	24	26	28	30
Flip-Flop	64	256	512	1024	2048	4096	8192	16384	32768	65536
Distributed	15	22	38	71	135	263	519	1159	2311	4615

Registers										
Address width "W"	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Linear	6	7	8	9	10	11	12	13	14	15
Flip-Flop	64	128	256	512	1024	2048	4096	8192	16384	32768
Distributed	6									

Latency [Cycles]										
Address width "W"	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Linear	64	128	256	512	1024	2048	4096	8192	16384	32768
Flip-Flop	1									
Distributed	64									

- CNT64 requires only 6 LUTs and 6 registers in total. However, the counter itself might be replicated several times by a synthesis tool (principle of locality [60]).

Therefore, the number of logic resources for the Distributed approach can be expressed in the following way:

$$\begin{aligned} LUT_{Dist} &= \lceil log_6(W-6) \rceil * 2^{(W-6)} + 7 * 2^{(W-6)} \\ &\quad + 2 * \lceil log_6(W-6) \rceil + 6 \\ &\approx \lceil log_6(W-6) \rceil * 2^{(W-6)} \qquad (8) \\ REG_{Dist} &= 6 \qquad (9) \\ Latency_{Dist} &= 2^6 = 64. \qquad (10) \end{aligned}$$

5) ESTIMATION DISCUSSION AND SUMMARY

We estimated several properties (latency, resource utilization) of respective techniques (see Table 1) by using formulas mentioned in the previous section. Therefore we discussed our expectations (design complexity and frequency) for individual techniques.

The Linear Passage technique requires the least amount of logic resources and also has the potential to reach high frequencies. However, the latency will grow exponentially, and this prevents this technique from being suitable for compression dictionaries unless a large external memory is used (memory is initialized only once during a power-up phase, for example), and the latency of initialization is not an issue.

Therefore, the Flip-Flop and Distributed based techniques were found suitable for implementations requiring dictionaries to be (re-)initialized before each run of a compression accelerator where the low latency is one of the requirements.

The Flip-Flop based approach having the best latency of one clock cycle is redeemed by enormous logic consumption (both LUTs and registers), which grows exponentially. The Distributed memory based approach seems to have the same advantage (constant latency) and disadvantage (the number of logic resources growing exponentially), however, the amount of required logic resources is decreased by a factor of 64. On the other hand, the latency is increased by the same factor to 64 clock cycles.

TABLE 2. Brief estimations and expectations.

Technique	**Linear**	**Flip-Flop**	**Distributed**
Resources	Minimal	Enormous	Moderate
Design complexity	Low	Moderate	High
Frequency	High	Moderate	High
Latency	Enormous	Minimal	Fair
Suitability[1]	Low	Fair	Great

Note 1: For hash table based compression dictionary architecture.

Thus the Distributed memory technique consumes less logic resources than the Flip-Flop technique, and it is assumed the frequencies will be higher in favor of the Distributed memory technique. The respectively increased latency will not be an issue in our case because the compression dictionary initialization could run in parallel with the loading data phase (see section III-C1). We assume this phase will take more clock cycles than the compression dictionary initialization phase for both Distributed and Flip-Flop techniques.

The general expectations for all approaches are summarized in Table 2. We assume the combination of latency, logic resources consumption, and frequency in the "sweet spot" range [5] (**"W"** between 6 to 15) will favor the Distributed technique. It is assumed that the final properties and results of individual techniques will change after implementation due to the various optimization used by synthesis tools [58], [59], such as resource sharing [61], [62], logic duplication, or register balancing [63] may be applied.

VI. THE DISADVANTAGE OF THE PREVIOUSLY USED METHODOLOGY

The initial set of measurements [5] was performed on a single computer with the Xilinx ISE toolset using the same initial conditions as those discussed in the following section VII-B. We used the "Random PAR" (Place & Route) mode in ISE, which allows to synthesize and PAR the design without setting-up physical constraints such as FPGA pins assignment. The creation of timing constraints such as a clock period is not affected by this mode. The disadvantage of this procedure is that designs (representing different approaches) with the same value of the **"W"** parameter have different pin placements. The observed randomness of the pin placements may affect the process of the synthesis, the PAR, and the STA in the final consequence. This might make an advantage (pins can be placed closer to an evaluated design) for one approach and penalize other approaches. This led us to prepare a new workflow to prevent this issue and to be supported by both ISE and Vivado.

VII. OUR APPROACH

We decided to choose a universal FPGA (in term of support in the Xilinx tools) to perform objective measurements. Thus we have selected the Xilinx Virtex-7 690T (XC7V690T-2FFG1158) due to its size and being a representative

TABLE 3. Computer systems used for the evaluation.

System/Platform	CPU	C/T[1]	RAM
Sun SunFire X4150	2x Intel Xeon L5420	8/8	64 GB
Supermicro H8DME-2	2x AMD Opteron 2382	8/8	64 GB
HP DL360 G6	2x Intel Xeon E5530	8/16	64 GB
Dell PowerEdge T620	2x Intel Xeon E5-2640	12/24	48 GB
ASRock B450M PRO4	1x AMD Ryzen 7 1700X	8/16	64 GB

Note 1: Cores/Threads

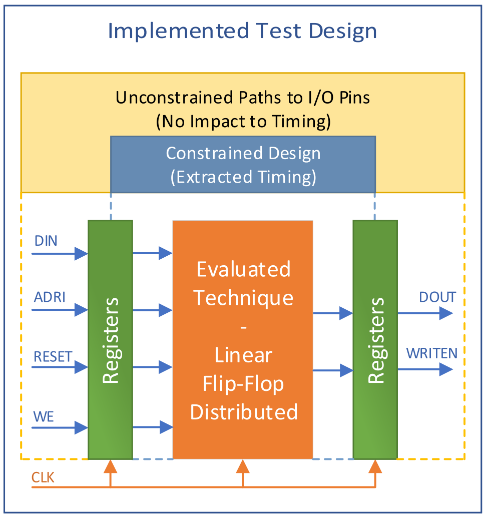

Implemented Test Design
Unconstrained Paths to I/O Pins (No Impact to Timing)
Constrained Design (Extracted Timing)
DIN
ADRI
RESET
WE
CLK
Registers
Evaluated Technique - Linear Flip-Flop Distributed
Registers
DOUT
WRITEN

FIGURE 5. Architecture of the design wrapper.

example of the Xilinx 7-Series FPGAs. The 7-Series architecture is the basis for the latest Xilinx FPGAs, such as the UltraScale(+) [64] platform. An advantage of Virtex-7 is that it is supported by both Xilinx development toolsets: the Xilinx ISE (Integrated System Environment) and the Xilinx Vivado in latest versions (version 14.7 and 2017.2 respectively). The reason to chose both tools is that the research started in 2015 (on Virtex-6 platform) when Vivado wasn‘t recognized as a ‘‘mature’’ product. Usage of the ISE also allows us to evaluate and compare the synthesis process against Vivado without violating Xilinx ISE license [65]. All tests were performed on several computer systems (see Table 3) representing trends in the last decade.

The synthesis process uses randomized algorithms; however, each system has it‘s own ‘‘seed’’. Therefore, we included the information that the used computer systems were different (and their respective configuration).

All systems have been following the requirements for Xilinx ISE [66] and Xilinx Vivado [67].

A. EXPERIMENTAL SETUP

We developed the design implementing all three mentioned approaches (linear passage, flip-flop array and distributed memory based approach) with a unified interface (see Program 1) in VHDL language.

The implemented approach and some other parameters (such as LUT size and the address vector width **‘‘W’’**) are set by generic constants. We assume the following design properties:

- Xilinx 7-Series architecture with 6-input LUTs.
- A memory cell width of 36 bits (one of the native BlockRAM widths [48], intended for simulating a 32-bit memory pointer like software version of LZ4 does. This dictionary cell width has also been used elsewhere [16]),
- **‘‘W’’** parameter stands for the intended memory address vector width and defines the memory capacity ($36{*}2^W$ bits).

As a precaution, we designed a test ‘‘Wrapper’’, which embeds an evaluated technique into a register array. The architecture (see Fig. 5) of the ‘‘Wrapper’’ will prevent the paths between physical FPGA I/O pins and the respective logical signals to have any impact on timing analysis. Therefore any path length could be virtually unlimited. Thus the technique designs can be substantially dense and floorplanned almost anywhere in an FPGA.

Program 1 A Unified VHDL Interface for All Implementation Types. [5]

```
entity top is port (
clk, reset, we: in std_logic;
adri: in  std_logic_vector(W-1 downto 0);
din: in  std_logic_vector(35 downto 0);
dout: out std_logic_vector(35 downto 0);
written: out std_logic);
end top;
```

B. ADDITIONAL SETTINGS FOR SYNTHESIS TOOLS

We changed some of the parameters from the defaults for Xilinx ISE and Xilinx Vivado to force the tools to favor the design speed instead of area. Some additional parameters were set to overcome some of the synthesis issues, such as a memory overflow.

1) Xilinx ISE [63]
   - Synthesis – Optimization Effort = Fast (Synthesis consumes less memory allowing synthesis of larger designs without a crash of Xilinx XST).
   - Synthesis – Register Balancing = Yes
   - Map – Register Duplication = On
   - Map – Allow Logic Optimization Across Hierarchy = Yes
2) Xilinx Vivado (Strategies) [68]
   - Synthesis – PerfOptimized_High
   - Implementation – Performance _ExplorePostRoutePhysOpt

C. OUR TEST METHODOLOGY WITH THE LINEAR PASSAGE APPROACH AS AN EXAMPLE

The new workflow improves the original workflow [5] by removing of the observed randomness of physical constraints via fixing the used constraints across all implemented approaches. The new workflow is depicted in Fig. 6, and some of these phases will be described in an example (the Linear Passage technique in our case) in a more detailed way.

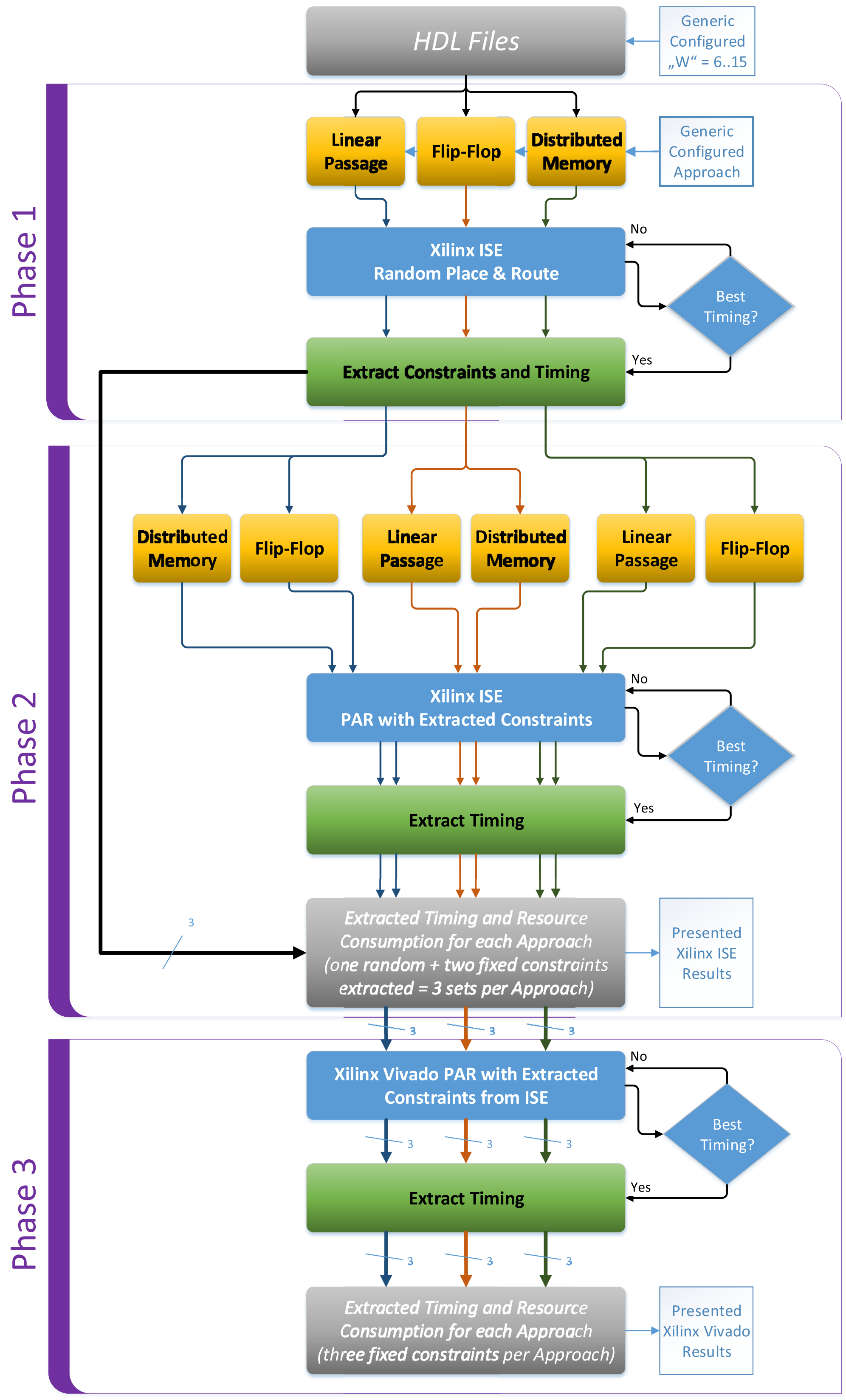

HDL Files
Generic Configured „W" = 6..15
Phase 1
Linear Passage
Flip-Flop
Distributed Memory
Generic Configured Approach
Xilinx ISE Random Place & Route
No
Best Timing?
Yes
Extract Constraints and Timing
Phase 2
Distributed Memory
Flip-Flop
Linear Passage
Distributed Memory
Linear Passage
Flip-Flop
Xilinx ISE PAR with Extracted Constraints
No
Best Timing?
Yes
Extract Timing
3
Extracted Timing and Resource Consumption for each Approach (one random + two fixed constraints extracted = 3 sets per Approach)
Presented Xilinx ISE Results
3
Phase 3
Xilinx Vivado PAR with Extracted Constraints from ISE
No
Best Timing?
Yes
Extract Timing
3
Extracted Timing and Resource Consumption for each Approach (three fixed constraints per Approach)
Presented Xilinx Vivado Results

FIGURE 6. The test flow used for experimental measurements.

The respective workflow paths for Linear Passage, Flip-Flop, and Distributed methods are highlighted with colors (blue, orange, and green).

1) PHASE ONE

The source files (HDL and initial timing constraints) are generated for implementation in Xilinx ISE. The best-reached clock period is extracted from the STA (Static Timing Analysis) report of the fully (randomly) routed design. The clock period is decreased by 0.05 ns (the smallest available step recognized by the Xilinx ISE) for the next iteration. The previous step is repeated until timing errors occur. The physical constraints (pin placement) and other reports (such as STA and PAR resources utilization report) are extracted at the end of phase one.

2) PHASE TWO

Phase two uses the extracted physical constraints (of the Linear Passage) to perform the implementation of the two remaining approaches (Flip-Flop and Distributed Memory based Status Register). The search for the best timing is the same as in phase one. The result of phase two is a set of three designs representing all approaches with their timing & physical constraints and resources utilization. These best frequencies (of the measured designs) are averaged over all approaches to reduce the influence of the random pin placement.

3) PHASE THREE

The collected ISE constraints are converted to a constraints format suitable for Xilinx Vivado. The designs are evaluated in the same manner as in phase one and phase two in Xilinx ISE.

D. COLLECTED DATA SETS

Nine data sets were collected after all three phases in the Xilinx ISE. Each approach had its own subset of three measurements:

- Linear Passage
  - Native constraints set (randomly generated)
  - Flip-Flop constraints set (fixed)
  - Distributed memory constraints (fixed)
- Flip–Flop
  - Native constraints set (randomly generated)
  - Linear Passage constraints set (fixed)
  - Distributed memory constraints (fixed)
- Distributed Memory
  - Native constraints set (randomly generated)
  - Linear Passage constraints (fixed)
  - Flip-Flop constraints set (fixed)

Each measurement had 10 fully routed designs with the STA report ranging the addresses with the **"W"** parameter from 6 to 15 bits. The measured datasets from the Xilinx Vivado had the same structure as the Xilinx ISE datasets.

TABLE 4. Properties of measured designs like logic gates count, frequency, etc. depending on the "W" parameter.

Linear passage approach										
Address width "W"	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Latency [Cycles]	64	128	256	512	1024	2048	4096	8192	16384	32768
Xilinx ISE										
Slices	70	92	98	90	103	84	111	112	114	109
LUTs	159	164	170	174	197	201	206	210	215	219
Registers	86	88	90	92	58	60	62	64	66	68
Frequency [MHz]	402	402	402	402	593	590	588	585	582	578
Xilinx Vivado										
Slices	18	25	26	27	26	36	36	49	52	60
LUTs	50	52	54	56	58	60	62	64	66	69
Registers	38	47	50	45	52	61	51	51	53	54
Frequency [MHz]	571	534	483	557	481	441	298	229	238	258

Flip–flop based approach										
Address width "W"	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Latency [Cycles]	1									
Xilinx ISE										
Slices	92	134	157	304	577	932	2275	4506	10026	17503
LUTs	221	309	478	840	1544	3250	6495	13912	31747	57586
Registers	145	210	339	615	1073	2167	4147	8244	16461	32955
Frequency [MHz]	402	373	395	297	260	245	218	204	176	161
Xilinx Vivado										
Slices	42	69	121	234	418	792	1563	2978	5888	12342
LUTs	109	174	317	596	1095	2145	4237	8312	16584	33573
Registers	89	178	340	713	1361	2690	5316	10577	21025	42303
Frequency [MHz]	472	292	295	232	230	234	224	194	207	207

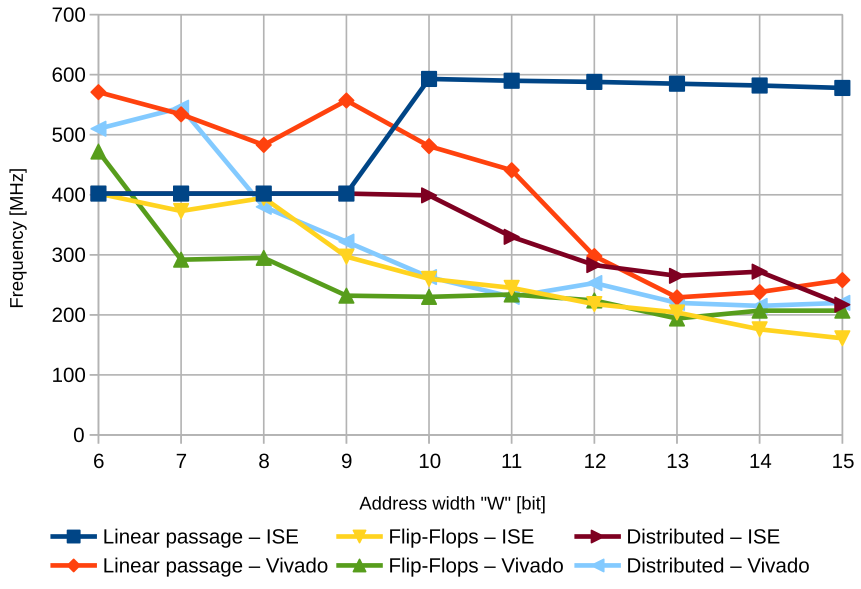
Distributed memory based approach										
Address width "W"	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Latency [Cycles]	64									
Xilinx ISE										
Slices	61	82	96	118	144	255	416	716	1382	2608
LUTs	51	58	71	96	145	245	439	839	1635	3235
Registers	87	94	107	132	145	242	435	820	1589	3126
Frequency [MHz]	402	402	402	402	399	330	283	265	272	217
Xilinx Vivado										
Slices	18	23	36	55	93	179	333	631	1206	2422
LUTs	142	158	189	255	372	605	1037	1991	3696	7491
Registers	16	32	62	123	245	494	986	1971	3941	7866
Frequency [MHz]	510	545	380	322	263	230	253	220	215	220

VIII. EXPERIMENTAL RESULTS

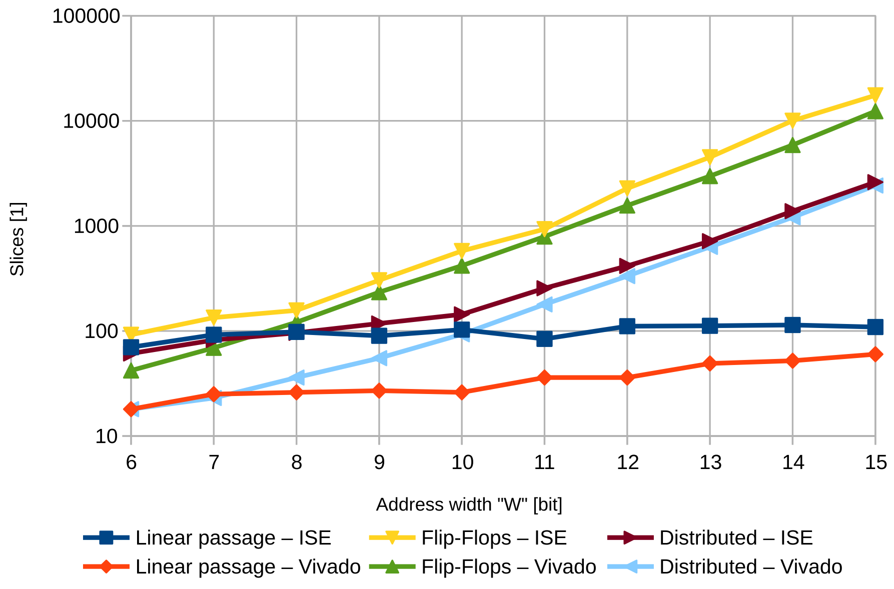
We evaluate and comment on the measured data (see Table 4) in terms of design speed, FPGA resources utilization, and suitability for compression dictionary design. These results are also visualised as graphs (Figures 7a, 7b, 8a, 8b).

A. LINEAR PASSAGE APPROACH

The Xilinx Vivado seems to be a better tool for implementing the linear passage approach in terms of chip area, thus consuming less FPGA resources in all cases. A synthesis or implementation issue appears to be in the Xilinx ISE. There is an unexpected "step" in the ISE frequencies, and results do not scale down in relation to the **"W"** parameter. The use of fast carry logic between neighboring slices might be the reason when no DSP48 [69] blocks are used



(a) Reached maximum frequency for all approaches (both ISE and Vivado).



(b) Slice consumption for all approaches (both ISE and Vivado).

FIGURE 7. Overall results, part I.

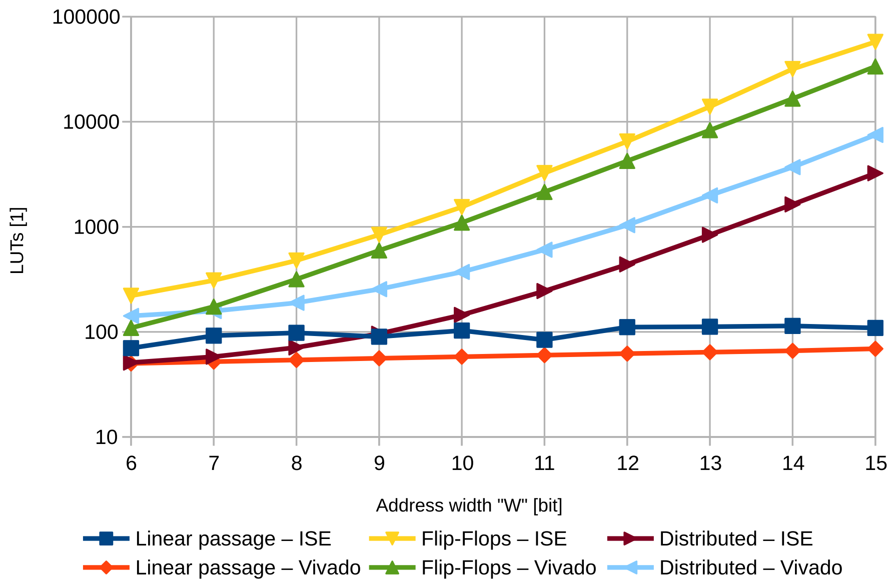
(the number of DSP48 blocks used is zero after PAR). However we were unable to determine the actual reason for the "step" even with a detailed analysis of respective floor-planned designs. It seems the implementation of such small designs in the such a large FPGA can lead to unpredictable results.

The initial results [5] showed that higher (better) frequencies were reached for the entire range while the graph curve is converging down for the Xilinx Vivado frequencies. This behavior needs to be further analyzed by performing the measurements on multiple computers.

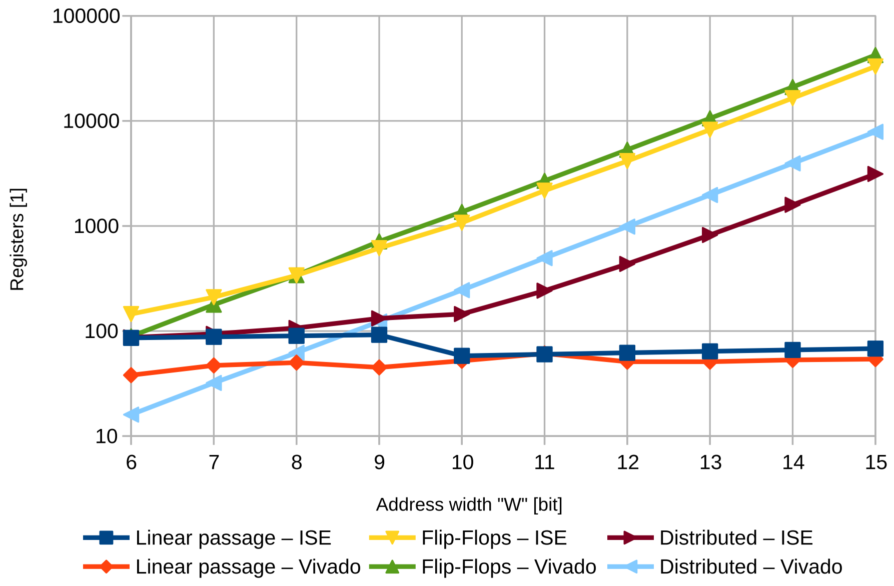
This approach is not suitable for larger dictionaries due to the enormous latency (growing exponentially) required for memory initialization. The approach is optimal in terms of used chip area.

B. FLIP-FLOP APPROACH

Implemented designs based on the flip-flop approach are comparable in terms of frequency for both Xilinx toolsets. The Xilinx ISE frequencies start on a lower point than Vivado, but convergence is slower compared to the Vivado results. The Xilinx Vivado is a better tool in terms of used FPGA resources saving 15%–55% of used slices with 32.5% on average. The solution of the flip-flop based status register is optimal in terms of latency (a single clock cycle, and it is constant for the entire range of the **"W"** parameter).



(a) LUT consumption for all approaches (both ISE and Vivado).



(b) Register consumption for all approaches (both ISE and Vivado).

FIGURE 8. **Overall results, part II.**

C. DISTRIBUTED MEMORY APPROACH

The last evaluation deals with the distributed memory approach of a status register. We prepared an additional table (see Table 5) to show the differences between the Xilinx ISE and the Xilinx Vivado results (see Fig. 9). The formula (11) used for calculating the differences follows

$$Difference = \left(1 - \frac{Vivado}{ISE}\right) * 100\ [\%]. \qquad (11)$$

The Xilinx ISE results show that the frequency start lower than Vivado for the first two smallest designs, but for the rest of the measured range, ISE is better than Vivado in terms of speed. On the other hand, the Xilinx Vivado has better results in terms of resource consumption over the entire range, but the advantage of Vivado is gradually diminishing (we assume the Xilinx ISE will be better for **"W"** parameter above the value of 15). There is no obvious winner in the end, because any single advantage of ISE or Vivado converges to zero while the **"W"** parameter rises (see Fig. 9).

D. SUITABILITY FOR A DICTIONARY DESIGN

We evaluated all approaches with respect to the requirements discussed in Section III-B. All approaches satisfy the requirement of the minimum design speed of 156.25 MHz. We have to exclude the linear passage approach due to its latency (it didn't meet the requirements, and it was too high compared

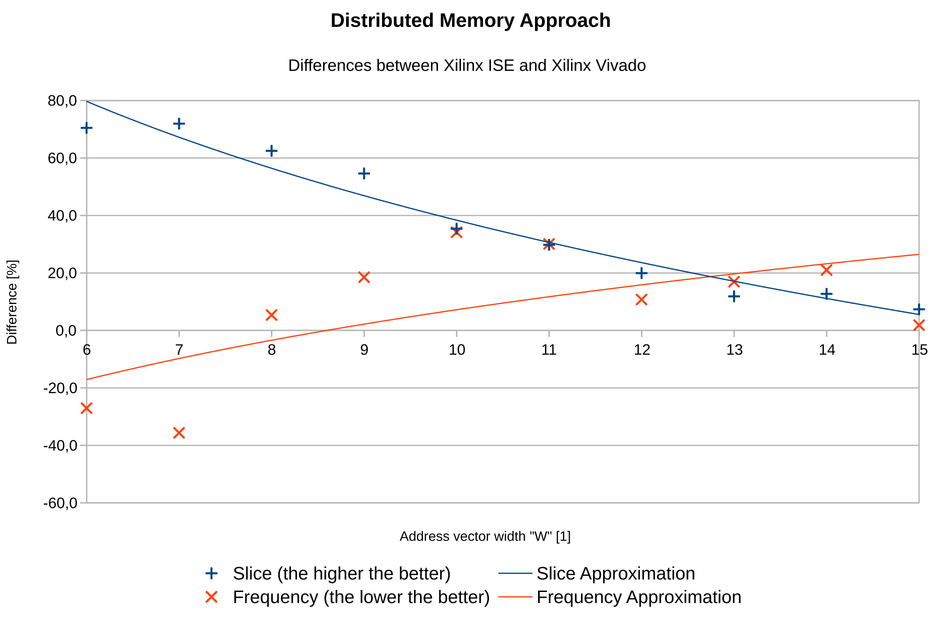


FIGURE 9. Differences for distributed memory approach.

TABLE 5. Differences between Vivado and ISE for distributed memory approach related to the "W" parameter – positive values represents the Vivado advantage.

Distributed Memory Approach										
"W"	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Slices [%]	70.5	72.0	62.5	54.6	35.4	29.8	20.0	11.9	12.7	7.4
Frequency [%]	-27.0	-35.6	5.4	18.5	34.2	30.1	10.8	17.0	21.0	1.8

TABLE 6. Advantage of distributed memory approach compared to flip-flop for the Xilinx Vivado results.

FPGA Resources – Distributed Memory vs. Flip-Flop										
"W"	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Area [%]	57.1	66.7	70.2	76.9	77.8	77.4	78.7	78.8	79.5	80.3

to the other two approaches). The flip-flop approach and the distributed memory approaches are comparable in terms of latency. The asymptotic complexity of both approaches is constant (1 respectively 64 clock cycles). The minor disadvantage of the distributed memory approach (the latency is slightly higher, but meets the requirements) is balanced by its better speed (higher frequencies are reached over the entire range), and less FPGA resources are consumed (see Table 6).

The average saving was 75% in terms of FPGA resources for the distributed memory approach. The issue of the flip-flop approach is the FPGA resources required for implementations. For example, the flip-flop approach will consume almost all resources in the Xilinx Virtex-7 690T FPGA for the **"W"** parameter equal to 20 (such design will support 1 million entries in a dictionary). The flip-flop approach cannot be used effectively with larger embedded memory such as the Xilinx UltraRAM feature [70], which increases the amount of FPGA embedded memory available by 600% from previous FPGA generations.

E. THE "DISTRIBUTED MEMORY" METHOD COMPATIBILITY WITH OTHER FPGA VENDORS

We considered four different FPGA vendors (Xilinx, Intel/Altera, Lattice, Microsemi/Actel) for compatibility with our method. Xilinx seems to have the most comprehensive support of LUT based "Distributed Memory" feature since the introduction of the very first Virtex/Spartan FPGAs [71].

Intel/Altera's FPGA support for the "Distributed Memory" feature varies across families. In general, modern and expensive FPGAs [72] do support the feature, while low-cost oriented and older families lack the support for the feature [73], [74]. Some FPGAs of Lattice and Microsemi/Actel also support the feature in certain more recent families [75], [76]. Due to the fact that these four vendors have 99% market share [77], our technique can be ported to nearly every modern FPGA.

IX. CONCLUSION

We presented a comprehensive analysis of three methods (linear passage, flip-flop, and distributed memory) suitable for initializing memory oriented data structures, including our distributed memory based approach. A performance comparison was performed in terms of the maximum reached operating frequency, FPGA resources consumption, and the requirements of the lossless compression dictionary design.

All approaches were measured over the range of hash table sizes suitable for IP compression devices. We presented the new test flow to support the Xilinx ISE and the Xilinx Vivado toolkits in the measurement process.

According to our methodology, the distributed memory approach shows the best combined performance against remaining techniques. This method is probably the only technique to satisfy the needs of future FPGA based implementations of various compression algorithms where a larger embedded memory such as the Xilinx UltraRAM feature is used. The presented technique is compatible and can be ported to any modern SRAM based FPGA architecture.

REFERENCES

[1] J. Fowers, J. Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on FPGAs," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2015, pp. 52–59.

[2] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd ed. New York, NY, USA: Springer, 2011.

[3] B. Sukhwani, B. Abali, B. Brezzo, and S. Asaad, "High-throughput, lossless data compresion on FPGAs," in *Proc. IEEE 19th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2011, pp. 113–116.

[4] T. Beneš, M. Bartík, and P. Kubalák, "High throughput and low latency LZ4 compressor on FPGA," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2019, pp. 1–5.

[5] M. Bartík, S. Ubik, and P. Kubalík, "A novel and efficient method to initialize FPGA embedded memory content in asymptotically constant time," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Nov. 2016, pp. 1–6.

[6] P. E. Bender and J. K. Wolf, "An improved sliding window data compression algorithm based on the Lempel-Ziv data compression algorithm [magnetic recording]," in *Proc. Global Telecommun. Conf. Exhib. Commun. Connecting Future (GLOBECOM)*, Dec. 1990, pp. 1773–1777 vol. 3.

[7] S. Rigler, W. Bishop, and A. Kennings, "FPGA-based lossless data compression using Huffman and LZ77 algorithms," in *Proc. Can. Conf. Electr. Comput. Eng.*, Apr. 2007, pp. 1235–1238.

[8] S. Rigler, "FPGA-based lossless data compression using GNU Zip," M.S. thesis, Dept. Elect. Comput. Eng., Univ. Waterloo, Waterloo, ON, Canada, 2007. [Online]. Available: http://hdl.handle.net/10012/2692

[9] R. Mehboob, S. A. Khan, Z. Ahmed, H. Jamal, and M. Shahbaz, "Multigig lossless data compression device," *IEEE Trans. Consum. Electron.*, vol. 56, no. 3, pp. 1927–1932, Aug. 2010.

[10] R. Mehboob, S. A. Khan, and Z. Ahmed, "High speed lossless data compression architecture," in *Proc. IEEE Int. Multitopic Conf.*, Dec. 2006, pp. 84–88.

[11] I. Shcherbakov, C. Weis, and N. Wehn, "A high-performance FPGA-based implementation of the LZSS compression algorithm," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum (IPDPSW)*, May 2012, pp. 449–453.

[12] E. J. Leavline and D. A. A. G. Singh, "Hardware implementation of LZMA data compression algorithm," *Int. J. Appl. Inf. Syst.*, vol. 5, no. 4, pp. 51–56, 2013.

[13] B. Li, L. Zhang, Z. Shang, and Q. Dong, "Implementation of LZMA compression algorithm on FPGA," *Electron. Lett.*, vol. 50, no. 21, pp. 1522–1524, Oct. 2014.

[14] P. M. Parekar and S. S. Thakare, "Hardware implementation of lossless LZMA data compression algorithm," *Prog. In Sci. Eng. Res. J.*, vol. 2, no. 3, pp. 201–205, May-Jun. 2014.

[15] M. Morales-Sandoval and C. Feregrino-Uribe, "On the design and implementation of an FPGA-based lossless data compressor," in *Proc. Sociedad Mexicana Ciencias Computación (ReConFig)*, 2004, pp. 29–38. [Online]. Available: https://www.tamps.cinvestav.mx/~mmorales/research.html

[16] S. Naqvi, R. Naqvi, R. A. Riaz, and F. Siddiqui, "Optimized RTL design and implementation of LZW algorithm for high bandwidth applications," *Przegląd Elektrotechniczny Elect. Rev.*, vol. 87, no. 4, pp. 279–285, 2011.

[17] J. L. Nunez, S. Jones, and S. Bateman, "X-MatchPRO: A high performance full-duplex lossless data compressor on a ProASIC FPGA," in *Proc. Int. Workshop Intell. Data Acquisition Adv. Comput. Syst. Technol. Appl.*, 2001, pp. 56–60.

[18] J. L. Nunez and S. Jones, "Gbit/s lossless data compression hardware," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 11, no. 3, pp. 499–510, Jun. 2003.

[19] J. L. Nunez, C. Feregrino, S. Bateman, and S. Jones, "The X-MatchLITE FPGA-based data compressor," in *Proc. 25th EUROMICRO Conf.*, vol. 1, 1999, pp. 126–132.

[20] J. L. Nunez and S. Jones, "Lossless data compression programmable hardware for high-speed data networks," in *Proc. IEEE Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2002, pp. 290–293.

[21] M. Milward, J. L. Nunez, and D. Mulvaney, "Design and implementation of a lossless parallel high-speed data compression system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 481–490, Jun. 2004.

[22] J. L. Nunez-Yanez and V. A. Chouliaras, "Gigabyte per second streaming lossless data compression hardware based on a configurable variable-geometry CAM dictionary," *IEE Proc.-Comput. Digit. Techn.*, vol. 153, no. 1, pp. 47–58, Jan. 2006.

[23] Helion Technology. *LZRW Compression Cores*. Accessed: Aug. 21, 2020. [Online]. Available: http://www.heliontech.com/comp_lzrw.htm

[24] M. A. A. El ghany, A. E. Salama, and A. H. Khalil, "Design and implementation of FPGA-based systolic array for LZ data compression," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2007, pp. 3691–3695.

[25] M. A. A. El ghany, A. E. Magdy, and A. E. Salama, *Design and Implementation of FPGA-Based Systolic Array for LZ Data Compression*. Rijeka, Croatia: InTech, 2010, pp. 75–92.

[26] K. Papadopoulos and I. Papaefstathiou, "Titan-R: A multigigabit reconfigurable combined compression/decompression unit," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 2, pp. 7:1–7:25, May 2010.

[27] W. J. Huang, N. Saxena, and E. J. McCluskey, "A reliable LZ data compressor on reconfigurable coprocessors," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, Apr. 2000, pp. 249–258.

[28] M. Morales-Sandoval and C. Feregrino-Uribe, "A hardware architecture for elliptic curve cryptography and lossless data compression," in *Proc. 15th Int. Conf. Electron., Commun. Comput. (CONIELECOMP)*, Feb. 2005, pp. 113–118.

[29] K. Papadopoulos and I. Papaefstathiou, "Titan-R: A reconfigurable hardware implementation of a high-speed compressor," in *Proc. 16th Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2008, pp. 216–225.

[30] A. Khu. (Sep. 25, 2001). *Xilinx FPGA Configuration Data Compression and Decompression*. Accessed: Aug. 21, 2020. [Online]. Available: http://www.xilinx.com/support/documentation/white_papers/wp152.pdf

[31] M. Bartík, S. Ubik, and P. Kubalík, "LZ4 compression algorithm on FPGA," in *Proc. IEEE Int. Conf. Electron., Circuits, Syst. (ICECS)*, Dec. 2015, pp. 179–182.

[32] S. M. Lee, J. H. Jang, J. H. Oh, J. K. Kim, and S. E. Lee, "Design of hardware accelerator for Lempel-Ziv 4 (LZ4) compression," *IEICE Electron. Exp.*, vol. 14, no. 11, p. 6, 2017.

[33] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.

[34] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, vol. IT-24, no. 5, pp. 530–536, Sep. 1978.

[35] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, Jun. 1984.

[36] I. Pavlov. (2016). *LZMA SDK*. Accessed: Aug. 21, 2020. [Online]. Available: http://www.7-zip.org/sdk.html

[37] Y. Collet. (May 26, 2011). *LZ4 Explained*. Accessed: Aug. 21, 2020. [Online]. Available: http://fastcompression.blogspot.cz/2011/05/lz4-explained.html

[38] T. Beneš, M. Bartík, and P. Kubalák, "Design of a high-throughput match search unit for lossless compression algorithms," in *Proc. IEEE 9th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Jan. 2019, pp. 0732–0738.

[39] W. A. Crofut and M. R. Sottile, "Design techniques of a delay-line content-addressed memory," *IEEE Trans. Electron. Comput.*, vol. EC-15, no. 4, pp. 529–534, Aug. 1966.

[40] F. J. Burkowski, "A hardware hashing scheme in the design of a multiterm string comparator," *IEEE Trans. Comput.*, vol. C-31, no. 9, pp. 825–834, Sep. 1982.

[41] D. Salomon, *Data Compression: The Complete Reference*. Berlin, Germany: Springer-Verlag, 2007.

[42] Z. Ullah, M. K. Jaiswal, and R. C. C. Cheung, "Z-TCAM: An SRAM-based architecture for TCAM," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 2, pp. 402–406, Feb. 2015.

[43] Z. Ullah, K. Ilgon, and S. Baeg, "Hybrid partitioned SRAM-based ternary content addressable memory," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 59, no. 12, pp. 2969–2979, Dec. 2012.

[44] M. F. Oberhumer. *Lempel-Ziv-Oberhumer*. Accessed: Aug. 21, 2020. [Online]. Available: http://www.oberhumer.com/opensource/lzo/

[45] J. Kane and Q. Yang, "Compression speed enhancements to LZO for multi-core systems," in *Proc. IEEE 24th Int. Symp. Comput. Archit. High Perform. Comput.*, Oct. 2012, pp. 108–115.

[46] O. Fiedler, "LZ-family data compression methods," M.S. thesis, Dept. Theor. Comput. Sci., CTU FIT, Prague, Czechia, 2014.

[47] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3, 2nd ed. Redwood City, CA, USA: Addison Wesley, 1998.
[48] Xilinx Inc. (2019). *UG473 (v1.14)—7-Series FPGAs Memory Resources*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf
[49] F. E. Goetting and T. J. Bauer, "Block RAM with reset," U.S. Patent 6 101 132 A, Feb. 3, 1999. [Online]. Available: https://patents.google.com/patent/US6101132
[50] M. Bencivenni et al, "Performance of 10 gigabit Ethernet using commodity hardware," *IEEE Trans. Nucl. Sci.*, vol. 57, no. 2, pp. 630–641, Apr. 2010.
[51] J. Hartmanis and R. E. Stearns, "On the computational complexity of algorithms," *Trans. Amer. Math. Soc.*, vol. 117, pp. 285–306, May 1965.
[52] C. Yoo, "High-speed DRAM interface," *IEEE Potentials*, vol. 20, no. 5, pp. 33–34, Dec. 2002.
[53] M. Stohanzl and Z. Fedra, "The FPGA implementation of dictionary; HW consumption versus latency," in *Proc. 36th Int. Conf. Telecommun. Signal Process. (TSP)*, Jul. 2013, pp. 82–85.
[54] M. Bartík and S. Ubik, "System for implementation of a hash table," U.S. Patent 10 262 702, May 3, 2019. [Online]. Available: https://patents.google.com/patent/US10262702B2/en
[55] Xilinx Inc. (2016). *UG474—7-Series FPGAs Configurable Logic Block*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf
[56] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for FPGAs," in *Proc. 18th Annu. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, New York, NY, USA, 2010, pp. 41–50.
[57] C. E. LaForest, Z. Li, T. O'Rourke, M. G. Liu, and J. G. Steffan, "Composing multi-ported memories on FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 3, pp. 16:1–16:23, Sep. 2014.
[58] V. Sklyarov, I. Skliarova, A. Barkalov, and L. Titarenko, *Synthesis and Optimization of FPGA-Based Systems*. Cham, Switzerland: Springer, 2014.
[59] C. Woods and B. Holdsworth, *Digital Logic Design*, 4th ed. Burlington, MA, USA: Newnes, 2002.
[60] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann, 2017.
[61] B. Ronak and S. A. Fahmy, "Improved resource sharing for FPGA DSP blocks," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2016, pp. 1–4.
[62] S. Hadjis, A. Canis, J. H. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski, "Impact of FPGA architecture on resource sharing in high-level synthesis," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, New York, NY, USA, 2012, pp. 111–114.
[63] Xilinx Inc. (2013). *UG627-XST User Guide for Virtex-4, Virtex-5, Spartan-3 and Newer CPLD Devices*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/xst.pdf
[64] Xilinx Inc. (2017). *DS890—UltraScale Architecture and Product Data Sheet: Overview*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf
[65] *Xilinx ISE 14.7 EULA*, Xilinx, San Jose, CA, USA, 2013.
[66] Xilinx Inc. *FPGA Memory Recommendations Using the ISE Design Suite 14*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/ise-design-suite/memory.html
[67] Xilinx Inc. *FPGA Memory Recommendations Using the Vivado Design Suite*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/memory.html
[68] Xilinx Inc. (2017). *UG904—Vivado Design Suite User Guide—Implementation*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug904-vivado-implementation.pdf
[69] Xilinx Inc. (2018). *UG497—7 Ser. DSP48E1 Slice User Guide*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf
[70] Xilinx Inc. (2016). *WP447—UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf
[71] Xilinx Inc. (2005). *XAPP464 (v2.0)—Using Look-Up Tables as Distrib. RAM in Spartan-3 Generation FPGAs*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp464.pdf
[72] Altera Corporation. (2011). *Logic Array Blocks and Adaptive Logic Modules in Stratix V Devices*. Accessed: Aug. 21, 2020. [Online]. Available: http://www2.engr.arizona.edu/~ece506/readings/project-reading/6-cad/stx5_51002.pdf
[73] N. Pramstaller and J. Wolkerstorfer, "A universal and efficient AES co-processor for field programmable logic arrays," in *Field Programmable Logic and Application*, J. Becker, M. Platzner, and S. Vernalde, Eds. Berlin, Germany: Springer, 2004, pp. 565–574.
[74] Altera Corporation. (2011). *Memory Blocks Cyclone IV Devices*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-iv/cyiv-51003.pdf
[75] Lattice Semiconductor Corporation. (2013). *Technical Note TN1201—Memory Usage Guide for MachXO2 Devices*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.latticesemi.com/-/media/LatticeSemi/Documents/ApplicationNotes/MO/MemoryUsageGuideforMachXO2Devices.ashx?document_id=39082
[76] Lattice Semiconductor Corporation. (2018). *AC476 Application Note—Design Migration Guidelines From Xilinx 7-Series to PolarFire*. Accessed: Aug. 21, 2020. [Online]. Available: https://www.microsemi.com/document-portal/doc_download/1243552-ac476-design-migration-guidelines-from-xilinx-7-series-to-polarfire
[77] J. Johnson. (Jul. 15, 2011). *List and Comparison of FPGA Companies*. Accessed: Aug. 21, 2020. [Online]. Available: http://www.fpgadeveloper.com/2011/07/list-and-comparison-of-fpga-companies.html

MATĚJ BARTÍK (Member, IEEE) was born in Prague, Czech Republic. He received the B.Sc. degree in computer science from the Faculty of Electrical Engineering, Czech Technical University in Prague, in 2012, and the M.Sc. degree in informatics from the Faculty of Information Technology, Czech Technical University in Prague, in 2014, where he is currently pursuing the Ph.D. degree.

He joined CESNET, as an FPGA and an Electronics Designer, in 2014. His research interests include electronic and PCB design, embedded system design, including embedded safety and security topics, and low-level FPGA optimizations.

TOMÁŠ BENEŠ (Student Member, IEEE) was born in Prague, Czech Republic. He received the B.Sc. and M.Sc. degrees in informatics from the Faculty of Information Technology, Czech Technical University in Prague, in 2017 and 2019, respectively, where he is currently pursuing the Ph.D. degree.

Since 2016, he has been an FPGA Hardware Designer with the Research and Development Department, CESNET. His research interests include hardware design, low-level optimization, and hardware network monitoring.

PAVEL KUBALÍK was born in Hořice, Czech Republic. He received the M.Sc. and Ph.D. degrees in informatics from the Faculty of Electrical Engineering, Czech Technical University in Prague, in 2002 and 2007, respectively.

From 2004 to 2009, he was an Assistant Professor with the Faculty of Electrical Engineering, Czech Technical University in Prague. Since 2009, he has been an Assistant Professor with the Faculty of Information Technology, Czech Technical University in Prague. His research interests include fault-tolerant design in FPGA, digital design in FPGA, self-testing circuits-based on FPGA, HW design of networks, high-speed wireless networks, arithmetic in FPGA, error control, and self-repair codes in FPGA.