Data Compression Accelerator on IBM POWER9 and z15 Processors

Industrial Product

Bulent Abali[2], Bart Blaner[1], John Reilly[1], Matthias Klein[1], Ashutosh Mishra[1], Craig B. Agricola[1], Bedri Sendir[3], Alper Buyuktosunoglu[2], Christian Jacobi[1], William J. Starke[1], Haren Myneni[1], and Charlie Wang[1]

[1]IBM Systems, [2]IBM Research, and [3]IBM Cloud

***Abstract*—Lossless data compression is highly desirable in enterprise and cloud environments for storage and memory cost savings and improved utilization I/O and network. While the value provided by compression is recognized, its application in practice is often limited because it's a processor intensive operation resulting low throughput and high elapsed time for compression intense workloads.**

The IBM POWER9 and IBM z15 systems overcome the shortcomings of existing approaches by including a novel on-chip integrated data compression accelerator. The accelerator reduces processor cycles, I/O traffic, memory and storage footprint of many applications practically with zero hardware cost. The accelerator also eliminates the cost and I/O slots that would have been necessary with FPGA/ASIC based compression adapters. On the POWER9 chip, a single accelerator uses less than 0.5% of the processor chip area, but provides a 388x speedup factor over the zlib compression software running on a general-purpose core and provides a 13x speedup factor over the entire chip of cores. On a POWER9 system, the accelerators provide an end-to-end 23% speedup to Apache Spark TPC-DS workload compared to the software baseline. The z15 chip doubles the compression rate of POWER9 resulting in even much higher speedup factors over the compression software running on general-purpose cores. On a maximally configured z15 system topology, on-chip compression accelerators provide up to 280 GB/s data compression rate, the highest in the industry. Overall, the on-chip accelerators significantly advance the state of the art in terms of area, throughput, latency, compression ratio, reduced processor utilization, power/energy efficiency, and integration into the system stack.

This paper describes the architecture, and novel elements of the POWER9 and z15 compression/decompression accelerators with emphasis on trade-offs that made the on-chip implementation possible.

I. Introduction

Lossless data compression is used in computing systems to reduce the amount of data to be processed, stored and transmitted. Compression is highly valuable and desirable in enterprise and cloud environments, where it can lead to storage and memory cost savings. It also reduces I/O and network utilization and processor cycles, improving overall performance. Figure 1 illustrates yearly cost savings that may be realized on an example cloud infrastructure with 2-4x compression. Savings can be tens to hundreds of thousands of USD for petabyte of storage per year, or petabyte of network traffic or a single memory-cache server according to the published prices [1]. An on-chip compression accelerator can deliver those savings with nearly zero hardware cost. Figure 1 shows another advantage of accelerators: Compared to a system with no accelerator, an end-to-end 23% speedup is achieved for the TPC-DS on Spark workload where compression is heavily used in the data processing pipeline.

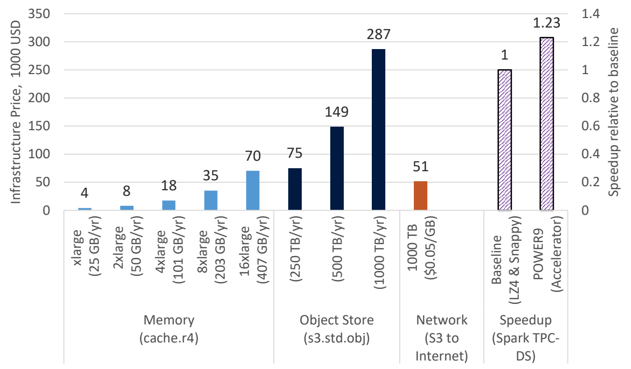

Fig. 1. Value of compression in two example environments: (A) 2x to 4x reduction in capacity or bandwidth utilization using compression would result in savings of tens to hundreds of thousands of USD per year [1]: for a single in-memory caching server, 1 PB of object storage, or for 1 PB of network traffic in the cloud, (B) The POWER9 on-chip accelerator provides 23% speedup to the Spark TPC-DS workload.

While the value provided by compression is widely recognized, its application is often limited because of the high processing cost and the resulting low throughput and high elapsed time for compression intense workloads. In the past,

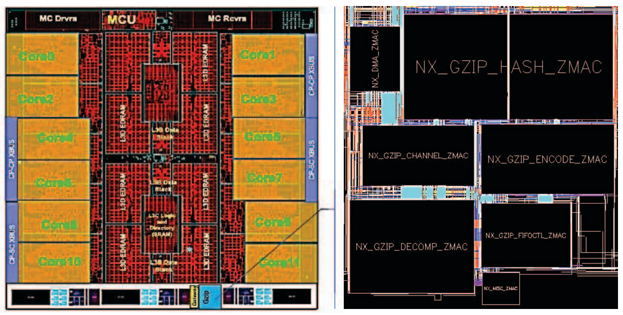

Fig. 2. z15 processor chip floorplan and detail of the NXU accelerator located at the southern edge of the chip.

This paper is part of the Industry Track of ISCA 2020's program.

IBM z13 and many other systems have used FPGA based PCIe attached compression accelerators [2]–[5]. However, the FPGA cost and limited number of PCIe slots restrict their usage to high-end servers [2] and specialized applications such as storage controllers.

IBM POWER9 (launched in 2017), and IBM z15 (launched in 2019) overcome the shortcomings of existing approaches by including a novel compression accelerator called NXU. Every z15 and POWER9 processor chip contains NXU. Product level software and libraries exploiting the accelerator are available for the IBM z/OS, zLinux, and AIX operating systems. For POWER Linux, open source drivers and libraries are currently going through the Linux community upstreaming process.

NXU is an industry-leading hardware accelerator implementing the Deflate standard with significantly higher throughput than any other implementation. On the largest z15 system topology with 20 processor chips, 20 NXU units provide 280 GB/s total throughput [6]. We would need 68 PCIe based compression cards such as [2], with 4GB/s peak throughput to match the NXU performance. Besides substantial improvements in performance, the compression function on IBM z15 features a novel approach to integrate hardware acceleration residing outside the processor core into the system stack: Although NXU utilizes classic I/O mechanisms like Direct Memory Access [DMA] in terms of data transmission and memory interactions, it is operated on behalf of applications by an architected z15 machine instruction implemented with millicode communicating with the hardware in lockstep, without operating system or hypervisor support.

The z15 and POWER9 on-chip accelerators feature several novel techniques to improve throughput and compression ratio: 1) CAM and pseudo-CAM based hybrid encoder that balances area and compression ratio in different ranges of the compression dictionary, 2) A processor-cache like area efficient hash table that uses search tokens and data tags to search up to 64 locations in the dictionary, 3) A dynamic Huffman encoder for improving compression ratio, and various other optimizations such as 2-pass, 1.1-pass modes, and cached code tables, 4) A "length limited" code table generator that enables the dynamic Huffman mode, 5) A decompressor including a parallel and speculative variable length code decoder designed for area efficiency, which breaks the 1 code per cycle limit of prior art, decoding up to 8 codes per clock cycle.

Overall, the resulting compression engine in IBM POWER9 and IBM z15 provide significant improvements in elapsed time, compression ratio, and higher capacity for existing workloads and reduces the cost of ownership. This paper describes the architecture, design, implementation, of the novel on-chip POWER9 and z15 compression and decompression accelerators with emphasis on the design trade-offs.

II. Background and Related Work

Deflate (RFC1951) is a compressed data format [7] with wide-spread support due to its inter-operability and software implementations available virtually on all platforms, the most popular being the zlib library and the gzip file compression utility [8]. Therefore, Deflate represents a natural choice for providing accelerated compression functionality in highly interconnected enterprise and cloud environments.

Deflate uses the LZ77 variant of the Lempel-Ziv (LZ) compression algorithm [9], followed by an entropy coding algorithm [7]. Few other compressed data formats and software exist making various trade-offs between the compression ratio and throughput. LZ4 has a minimum match length of 4 bytes and no entropy encoding. Snappy has a byte-oriented format and no entropy encoding. ZSTD has large search windows and uses the *tANS* encoder in addition to Huffman [10]–[13]. Compression and decompression throughput of these are ordered highest to lowest as LZ4, Snappy, ZSTD, and Deflate (as implemented by the zlib library and the gzip tool). Their average compression ratios are ordered from best to worst as Deflate (zlib/gzip), ZSTD, LZ4, and Snappy. For few exceptional files ZSTD may compress better than zlib/gzip because of its larger window size and other algorithmic improvements. We chose to implement a Deflate accelerator because it is well-established with widespread support and has the potential for good compression ratio. We compare our work to previous work in Sections VIII-IX.

LZ77 replaces duplicate strings with references to earlier copies in the most recent 32KB of the source stream [9], called the *sliding window* or *dynamic dictionary* (called "History" in this paper). An LZ reference is a distance/length pair, a copy instruction from an earlier string to the current point in the stream. Deflate distances are 1-32768 bytes and lengths are 3-258 bytes. Source bytes not encoded with an LZ reference are encoded as LZ literals. The goal of an LZ77 encoder is to find the longest matching string in the sliding window resulting in the smallest encoded output. This goal challenges both hardware and software implementations, often resulting in a tradeoff between area (or memory), performance, and compression ratio. We describe an area efficient LZ77 encoding hardware that improves state of the art in Section IV, where the related work [2]–[4], [14]–[19] are also discussed.

Deflate uses variable length prefix-free codes produced by the Huffman algorithm to compress an LZ77 stream by another 10-20% [20]. Frequent symbols are encoded with fewer bits. For example, the most frequent character 'e' in English may be encoded with 6-bits vs. the usual 8-bits. Deflate specifies a maximum code length of 15-bits, called "length limited encoding" [21]. A "Dynamic Huffman Table" (DHT) maps the LZ77 alphabet of symbols to 1 to 15-bit codes. DHTs are custom made according to LZ77 symbol statistics.

Producing a dynamic Huffman table (DHT) with a high throughput is a challenge. Many hardware designs do not solve this problem and instead they implement a static table with a predetermined LZ symbol distribution which typically degrades the compression ratio. In contrast, we implemented a true "Dynamic Huffman" mode in both POWER9 and z15 to achieve highest possible compression ratio, as described in Section V, where related work [2]–[4], [7], [14], [20], [22]–[26] and hardware issues are also discussed.

The prefix-free property allows concatenation of variable

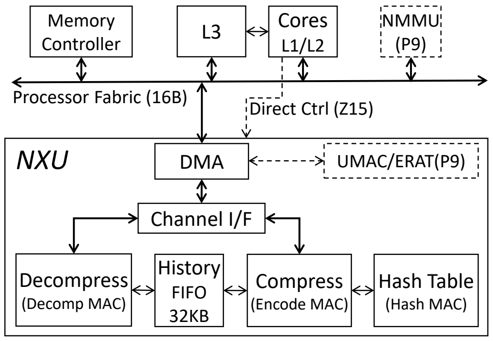

Fig. 3. Top level architecture and processor attachment of the accelerator.

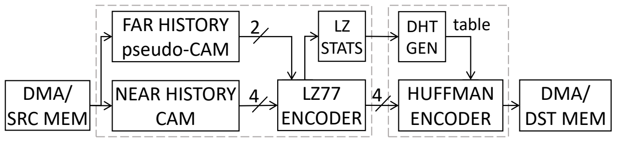

Fig. 4. Compressor data flow left to right: the hybrid design contains two types of CAM and an LZ77 encoder followed by a Huffman entropy encoder.

length codes with no delimiters in-between which challenges high throughput decoders. Simple hardware designs can handle at most one code per cycle [27], because the next code's first bit position cannot be known until the current one is decoded. We describe a speculative decoder capable of decoding 8 codes per cycle in Section VI where the related work [27], [28] is also discussed.

Low latency access to an accelerator is important to make it useful for a broad range of software applications. We developed user-mode interfaces and a user-mode page fault handler to provide low latency access, described in Section VII. Related works include [29]–[32].

IBM has been employing compression hardware and software extensively in its products. IBM z13 and POWER systems used PCIe based Deflate accelerators [2], [5], [15], [28]. POWER7+ processor implemented a proprietary on-chip memory compression accelerator [33]. The MXT system (x86 architecture based) implemented transparent compression in the memory controller [34], [35]. Compression is standard on DS8000 and Storewize SAN storage systems, and TS7700 tape storage systems [16]–[18], [36], [37].

III. High-Level Accelerator Overview

The z15 and POWER9 processor chips use 14nm process. NXU occupies less than 0.5% of the chip area. The z15 unit is slightly larger than the POWER9 unit due to differences in the hash arrays and the Huffman encoding logic. Figure 2 shows the z15 chip floor plan and the NXU unit detail at the southern edge of the chip. On both chips, NXU attach to the internal processor fabric as shown in Fig. 3. NXU handles data at a rate of 8 bytes/clock at 2 GHz on POWER9 and 2.5 GHz on z15 (NXU runs at a lower clock frequency than the processor cores). The DMA engine moves data between the processor fabric and the Compress and Uncompress macros.

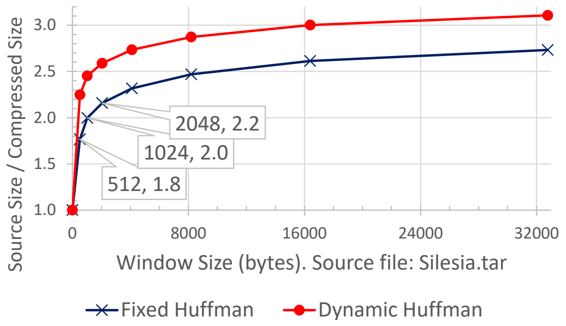

Fig. 5. Compression ratio as a function of sliding window size (zlib 1.2.11) and Silesia corpus [39]. With increasing window size the compression ratio levels off.

The History FIFO is an SRAM that stores the sliding window data. The Compress macro leverages an SRAM based hash table for searching strings in the History FIFO.

The z15 and POWER9 NXU cores are mostly identical. Since the z15 processor chip was developed later, few performance enhancements were added such as increased number of hash table ports (Section IV-D) and hardware produced code tables (Section V). The user interfaces to the accelerators were also implemented differently according to the ISA requirements of the two systems (Section VII).

Both processors enable user-mode access to NXU to eliminate OS kernel and hypervisor overheads. z15 cores use the machine instruction DFLTCC to directly control NXU [38]. POWER9 cores instead, send control messages to NXU. Software uses virtual memory addresses and page pinning is not necessary.

Figure 4 shows the compressor data flow. Source data enters from the left and compressed data exits from the right. The LZ77 stage comprises two types of Content Addressable Memory (CAM) for searching the sliding window with input phrases and an LZ77 compression encoder that outputs up to 4 LZ77 symbols per cycle (LZ references and LZ literals). Symbols are then fed to a Huffman encoder that further compresses the data using a code-table generated by DHTGEN. NXU elements are described in the following sections.

IV. Compressor: the LZ77 Stage

A. A Hybrid LZ77 Encoder

Area is an important concern in an on-chip design. The accelerator is one of many tenants on a processor chip with cores/caches. Therefore, area is at premium compared to an FPGA or an ASIC chip which may have more resources available. Trade-offs exist between silicon area, throughput and compression ratio. We designed a hybrid encoder that makes these tradeoffs by budgeting the area where it is most effective.

Few LZ encoders search for string matches using Content Addressable Memories (CAM). CAMs are typically implemented with custom circuits [19] or with latches and random logic such as in our design. Thousands of byte-wide comparators find every match of a search key in the CAM in the same cycle. However, CAMs are area and power hungry and have timing challenges at our target 2–2.5 GHz frequency. A pseudo-CAM on the other hand, is SRAM based: multiple banks of SRAM arrays function as a hash table for storing search items. A pseudo-CAM is area efficient because we estimated that SRAMs store about 14 to 16 times as many bits per unit area than latches in the 14nm CMOS process. But pseudo-CAM are imprecise and lossy: a limited number of SRAM locations may contain a search item. SRAM port count, hash and bank collisions gate throughput and accuracy.

We recognized that increasing the sliding window size has a diminishing return on the compression ratio as shown in Fig.5. Highest compression ratio change occurs in the near-history (near the origin in Fig.5) and levels off in the far-history. Reasons are two-fold: duplicate strings are often found close to each other and Deflate uses extra bits for encoding far pointers [7].

Insights in Fig.5 led to the novel ”hybrid” compressor in Fig.4 combining an area hungry true CAM for the near-history (≤ 512 bytes) with an area efficient pseudo-CAM for the far-history (513-32KB). The design budgets the silicon area between two different style of CAM according to the compression ratio benefit derived from the two ranges. Area and timing constraints didn’t permit building a larger than 512 byte near-history CAM.

B. The Near-History CAM

The CAM was implemented with random logic based on a design used in an IBM storage controller [16]–[18], shown in Fig.6. Note that this CAM is specialized for data compression reporting partial matches of a search key at any byte alignment unlike the traditional CAMs such as those used in network switches [19]. Input is latched to the $M = 8$ byte column of latches on the left edge. Previous cycle’s input bytes shift into the row of $N = 512$ byte history latches at the top, in FIFO order. $M \times N$ byte comparators compare the input to the history and their match status are reported to the Priority Encoder on the right edge (fan-in of thousands of wires are omitted in Fig.6). A column of adjacent matches indicates a match between the input and the stored bytes (illustrated by the circles). Each comparator receives a match count from the comparator above it in the same column. It increments by the count by 1 and passes down to the comparator below. Therefore, each comparator knows how many bytes matched above it (0 to 7 bytes) and the maximum match length is known in each column and row offset.

A string match longer than 8 bytes spanning multiple cycles is tracked by the the 1-bit ”continue” register C at the top, with one register allocated per column. $C = 1$ indicates that there was a match in the same column in the previous cycle. Therefore, a long match spanning the two cycles is present and the logic has deferred its encoding to this cycle. Likewise, in the current cycle the logic sets $C \leftarrow 1$ when a match ends at the bottom row. It indicates that the match may potentially continue in the next cycle.

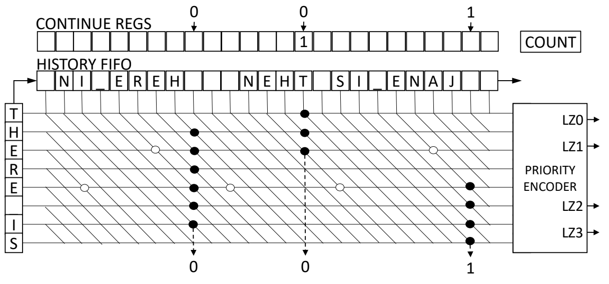

Fig. 6. Near history CAM and the 4-way parallel LZ encoder illustrating 6, 3, and 4-byte matches. The 3-byte match is a continuation from the previous cycle. The 4-byte match may continue into the next cycle.

The $COUNT$ register accumulates the length of the longest match continuing from one cycle to the next. When a string match terminates, the Priority Encoder calculates the LZ distance using the column offset and the LZ string length using the row offset plus $COUNT$.

The Priority Encoder (Fig.6) outputs up to 4 LZ symbols per cycle which was sized to sustain 8-byte per cycle input data rate on lightly compressible data with a mix of short LZ references and LZ literals, e.g. English text.

Figure 6 contains a mock example in which the 8 byte input data matches 3 different strings with match sizes of 6, 3, and 4 bytes. The 3-match is adjacent to the previous cycle’s boundary. Since $C = 1$, the 3-match is continuation of a string match from the previous cycle. Since the 4-match is adjacent to the next cycle, its encoding may be deferred by setting $C \leftarrow 1$ and $COUNT \leftarrow 4$.

A CAM may produce hundreds of matches but not all will be encoded since some overlap each other as shown in Fig.6. The Priority Encoder chooses a subset of matches producing the smallest possible output, yielding the highest compression ratio. Matches are prioritized according to their length and taking the $COUNT$ register and C register values into account.

C. The Far-History CAM

The Far-history pseudo-CAM is constructed with SRAMs. In essence it is an 8-way set-associative cache with 8 R/W ports as detailed in this section. Area efficiency is the primary concern in this design. The 32KB sliding window is stored in 1-port SRAMs called the ”History-FIFO” in Fig.7, written sequentially from the input stream in FIFO order. Virtual 2-read ports are implemented with multiple banks of SRAM. The problem solved here is finding the longest match (3 to 258 bytes) of the input data in the History FIFO at a rate of 8-byte/cycle.

Long strings are matched through their substrings, called ”tokens” (tokens have a fixed size for ease of hardware implementation). The 8-byte input is decomposed into overlapping K-byte search tokens ($K = 5$ is the firmware default.) A token can span two cycles of input. Each token is hashed to an 11-bit index of the 2048-entry hash table (Fig.7). A hash table entry contains 8 addresses which are the candidate History-FIFO locations of the search token. Items in the hash table entry are

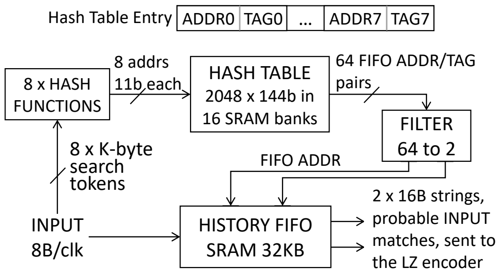

Fig. 7. Far history pseudo-CAM uses a hash table to search the recent 32KB input. Data tags eliminate false positives.

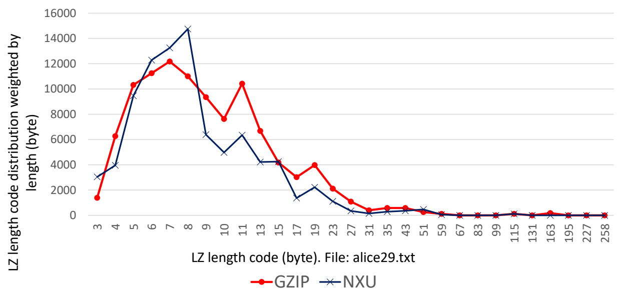

Fig. 8. LZ length code distribution weighted by length. Area under the curves are proportional to the source data compressed by two LZ77 implementations

replaced in FIFO order. In effect, the hash table is an 8-way set-associative cache directory with FIFO replacement policy and with 8 R/W ports. The 8 virtual ports are implemented with 16 banks of 1-port SRAMs in POWER9 and 2-port SRAMs in z15. Bank conflicts are resolved either by stalling the compressor pipeline or by dropping R/W requests to be discussed in Section IV-D.

Eight tokens of the input word hash to 8 different hash table entries returning 64 addresses in total. However, the History-FIFO SRAM has only 2-read ports and therefore 2 addresses out of the 64 History-FIFO addresses must be selected. The FILTER in Fig.7 uses probabilistic data tags and address cross-comparisons to identify two candidates, one of which may potentially yield the longest match.

Non-equal tokens may hash to the same table entry. False positives are reduced with yet another hash of the search token, a 6-bit digest called ”tag”, stored in the hash table (Fig.7). Tags are saved in SRAM since they are smaller than tokens (6-bits vs. K-bytes). The FILTER dismisses a large fraction of non-equal tokens simply by comparing the input tags to the stored tags. False positive probability is $1/2^6$ which may degrade compression ratio, a trade-off made in exchange for saving area. Note that false positives are not a correctness issue since any data read from the History-FIFO and the source input are eventually compared to verify a match. The FILTER uses additional heuristics for identifying long matches not detailed in this paper. The final two addresses out of the FILTER are used to read two data chunks from the History-FIFO. The two chunks are then fed to the LZ77 Encoder (Fig.4) which selects from multiple matches reported by the near and far history CAMs.

An example hybrid encoder result is shown in Fig.8. The benchmark file is compressed with the gzip tool and NXU. The LZ length distribution weighted by the length itself (minus an estimate of 2 bytes per pointer overhead) are plotted. Area under the two curves represent the source bytes compressed and eliminated by the two methods. We observe that NXU does not encode as many desirable long matches ($>$ 8 byte) as the gzip tool, evidenced by the NXU curves generally lower than gzip in that x-axis region. NXU encodes more 3-8 byte matches than gzip. NXU finds fewer 4-byte matches than gzip because 5-byte tokens cannot identify 4-byte strings as they will hash to different entries to be discussed next, in Section IV-D.

D. Discussion on Design Trade-offs and a Comparison

Originally, the hashing logic was designed for $K = 3$-byte tokens since 3-byte is the minimum string length in Deflate. A short token may identify many matches, 3-bytes and longer. But only 8 positions are available in a hash table entry. For example, consider the phrase **the** which is contained in many longer strings, **the␣**, **then␣**, **there␣**, **therefore␣** whose first 3-bytes all hash to the same entry when $K = 3$. Items are aged and lost faster in such hot table entries due to FIFO-replacement. On the other hand, with a long search token size, e.g. $K = 8$, hot entries are fewer and items age slower but they cannot find shorter ($< K$ byte) substrings of the token.

We experimentally determined that $K = 5$ performs the best on average for a range of benchmarks. Outliers exist such as the internal CUST2 benchmark dataset shown in Fig.9 for which $K = 8$ yields 10% higher compression than the default; it indicates that the Far-history range contains many 8-byte or longer matches, which $K = 8$ is better at identifying. Note that the near-history logic is not affected by the token size and it still identifies as short as 3-byte matches.

The hash table is constructed using 16 banks of 1-read and 1-write port SRAM on the POWER9 chip. With 8 requests per cycle, bank conflicts occur 87% of the time on average, stalling the pipeline to resolve the conflict with an average access latency of 2.1 cycles (1.0 ideal). A firmware configuration option is available to drop conflicted access requests: higher throughput may be achieved at the expense of increased hash table misses and therefore reduced compression ratio. The z15 processor in comparison to POWER9, adopted 2-port SRAMs to reduce bank conflicts resulting in an average 1.12 cycle access latency, however at the expense of larger SRAMs. Performance benefits of this change will be discussed later on Fig.13 in Section VIII-C.

The NXU hash table organization is quite efficient in terms of SRAM bits. Total capacity used for the sliding window is 32KB for History-FIFO plus 36KB for the hash table. Since the z15 hash table uses 2-port SRAMs; its area is larger in z15 than POWER9. Prior art, mostly FPGA based, often replicate logic to meet throughput and compression ratio objectives which is not suitable for an on-chip design. In [4], four separate compression engines work in parallel on four different

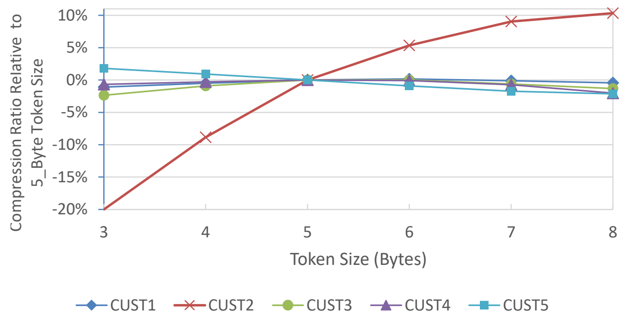

Fig. 9. Compression ratio change as a function of search token size for five datasets. Base compression ratio is for 5-byte tokens. Higher is better.

segments of the input stream and compressed outputs are concatenated to achieve high throughput, reportedly with some degradation of compression ratio. 16,177,152 bits (1.92MB) of FPGA block memory was used. In [14], 7 independent engines with 2.4 Gbit/s throughput were used to meet total throughput requirements in a multithreaded environment. 26,401,536 bits (3.15MB) of FPGA block memory was used. In comparison, 66KB of SRAM is used in our design.

In [2], [15], the 32KB sliding window data is replicated M times stored directly in M SRAMs organized as hash tables ($M = 16$). Each hash table has M-read and 1-write ports (M^2-read ports total). M overlapping substrings of the $2M$-byte two-cycle input string are searched in the M SRAM modules (M^2 reads total). Compared to our design, $O(M)$ times more storage capacity and M times as many read ports are required.

V. Compressor: Huffman Stage

The Huffman encoder (Fig.4) encodes the four LZ77 symbols received per cycle with 1 to 15-bit variable length codes. The accelerator must produce a Dynamic Huffman Table (DHT) customized according to the LZ symbol statistics which maps LZ77 symbols to the codes. The main problems of Huffman encoding are (a) generating the table (DHTGEN) fast and (b) encoding the LZ77 encoder output (Fig.4) with the DHT at a 8-byte/cycle input data rate.

The number of literals, lengths, and distances produced by the LZ77 encoder are accumulated in an SRAM array of counters at the output of the LZ77 encoder (LZ STATS, Fig.4.) Then, DHTGEN uses a length limiting (15-bit max) algorithm to generate a DHT based on those counter values. POWER9 and z15 implementations differ in this step. POWER9 NXU expects software to supply a DHT which takes longer to produce than in a hardware implementation. z15 NXU on the other hand has a self-contained hardware based DHTGEN unit. The two processors use different strategies to minimize the DHTGEN overheads as detailed in Section V-D.

A. Producing Dynamic Huffman Tables

The basic Huffman algorithm runs in $O(n \log n)$ time where n is the number of symbols in the table; $n = 286$ for the literal and length symbols and $n = 30$ for the distances [20]. The algorithm is sequential and a basic hardware implementation would have a relatively long execution time negatively impacting overall throughput.

We developed a novel parallelized and length limiting Huffman algorithm in the hardware. A priority queue-based Huffman algorithm runs in $O(n)$ time when symbols are pre-sorted by their count [22], [23]. We implemented the 2-dimensional parallel *Shear-Sort* algorithm to sort n counter values in $O(\sqrt{n} \log_2 \sqrt{n})$ cycles [25], [26]. The sorter stores symbols and their counts in latches to enable $n/2$ pairs of compare-exchanges per cycle. Counts are converted to 10-bit floating point values to reduce the number of latches. Once sorted, the priority queue based Huffman algorithm is executed. Performance of this unit is discussed in Section VIII-D.

B. Length-Limited Encoding

Maximum code length permitted in the Deflate format is 15-bits. The Huffman algorithm builds a binary tree with LZ symbols at the leaves arranged according to their probabilities [20]. The path from the tree root is the binary code of the LZ symbol assigned to the leaf. In the unlimited case, the tree depth may excessively grow as much as $n - 1$ ($n = 286$ max.), which is the subject of "length-limiting" algorithms. The algorithms in the literature were found to be complex for a hardware implementation [21], [23]. Our algorithm boosts small counter values to the effect of limiting the Huffman tree depth and then uses the basic unlimited length Huffman algorithm. An upper bound on the Huffman tree depth is given as $\lfloor -\log_\phi p_{min} \rfloor$, where $\phi = (1 + \sqrt{5})/2$ and p_{min} is the probability of least frequent symbol [24]. However, this upper bound, when used as a predictor of tree depth, is pessimistic in practice and difficult to implement in hardware due to the non-integer $\log$ base. We observed that $\lfloor -\log_2 p_{min} \rfloor$ is a more accurate predictor of tree depth, although not a strict upper bound. Our algorithm iteratively normalizes LZ counts by the following integer division operation,

$$\mathit{count} \leftarrow (\mathit{count} + 1)/2$$

which has the effect of boosting the probability of small counts therefore reducing the tree depth, but at the same time nearly preserving the probability of large counts and their sorted order. Normalization is repeated until the predictor is satisfied

$$\lfloor -\log_2 p_{min} \rfloor \leq 15$$

For ease of hardware implementation, the inequality may be raised to the power of 2 and integer counts may be used (e.g. reciprocal of p_{min}). If the maximum code length exceeds 15, the normalization step and the Huffman algorithm are repeated. A second iteration occurs rarely with simulated random trees.

C. 1.1-Pass Mode for High Throughput

The second problem of Huffman encoding is making two compression passes over the source data. The first pass is needed only to collect statistics which reduces the accelerator throughput by half. In the first pass, compressed output is

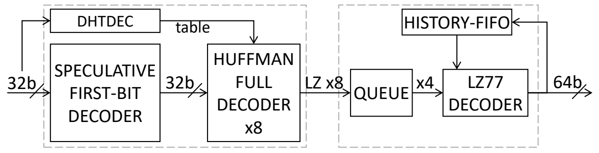

Fig. 10. The NXU decoder architecture consist of a parallel speculative Huffman decoder followed by an LZ77 decoder.

discarded but the LZ counts are kept for processing in DHT-GEN. In the second pass, the DHT is used to finally produce the actual Huffman encoded compressed output (recall that the DHT maps the LZ symbol alphabet to 1 to 15-bit codes.)

To avoid two passes, we hypothesized that sampling the source data might be sufficient to obtain representative LZ symbol frequencies. Most data streams have ”locality” in a small window of few hundred KB: literals, length and distance symbols are similarly distributed throughout a data stream. For example, if English text is used at the beginning the same is likely for the rest. Therefore, the z15 implementation makes the first pass only on the first 32KB of input data (sample size is configurable.) Resulting DHT is used in the second compression pass for up to 256KB of source data. We refer to this as the *1.1-pass* method; it increases throughput from 50% to ideally 256/(256+32) = 89% of the hardware peak. We experimentally determined that sampling first 32KB is satisfactory in most cases. Although, as a future optimization different regions of sampling and sample sizes may be considered. Compression ratio impact of the 1.1-pass over the 2-pass method is presented in Section VIII-B.

D. Discussion on Design Trade-offs

Many hardware designs implement a static table with an assumed LZ symbol distribution which typically degrades compression ratio [2]–[4], [14]. For example, Fig.5 shows the compression ratio difference between static and dynamic tables. IBM z13 adapter uses a ”pseudo-dynamic” approach where one of 15 different static Huffman tables yielding the smallest output is selected at run time [2]. We implemented in contrast, a true ”Dynamic Huffman” mode for both POWER9 and z15 to achieve highest possible compression ratio.

On POWER9, software must generate a DHT [40] which often takes long time to compared to the actual compression time. As a result, recently used DHTs are stored in a software cache which amortizes the software overhead over multiple jobs [40]. Based on the POWER9 lessons learned, the z15 system added a DHTGEN hardware unit that eliminated the software overheads. The DHTGEN hardware unconditionally produces a new DHT for each job.

VI. Decompressor

Deflate’s 1 to 15-bit variable length codes are concatenated with no delimiters in-between. In principle, codes must be decoded serially one at a time, since the next code’s starting bit position cannot be known before the current code’s length is determined. However, our design requires decoding up to 4 LZ references per cycle (4 length/distance pairs = 8 codes/cycle.)

We designed a parallel speculative Huffman decoder (Fig.10) to overcome the bottleneck. Compressed source data enters 32-bit/cycle from the left and uncompressed raw data exits 64-bit/cycle from the right. At one extreme, the 32-bit input word may contain 32 codes 1-bit each. At the other extreme, it may contain 2 codes 15-bits each and some fraction of a third code extending into the next cycle.

The speculative decoder, without knowing the code lengths, assumes that 32 different codes of any length start at every bit offset of the 32-bit input word. It decodes them in parallel, 32 codes per cycle. Wrongly speculated codes are invalidated later in the decoder pipeline since their assumed first-bit positions are wrong.

The design was inspired by an FPGA based decoder described in [28]. We reduced the SRAM and area consumption of that design which made the on-chip integration possible. Our main insight was that 32 full-decoders were not necessary. Each full-decoder requires a large SRAM mapping 1 to 15-bit codes to 286 length and literal symbols and 30 distance symbols. We instead implemented a speculative ”First-Bit Decoder” (FB) which only identifies the starting bit positions of codes in the 32-bit input. The FB decoder does not store any LZ symbols and therefore it is smaller than a full-decoder. The FB decoder identifies the first bit position of the first Huffman code in the input word and hands it off to one of eight full-decoders shown in Fig.10. The 8 full-decoders work independently and in parallel starting at their respective first-bit positions and they produce LZ77 references which are then queued at the LZ77 decoder’s input. Finally, the LZ77 decoder (Fig.10) processes up to 4 LZ references per cycle.

VII. Integration of the Accelerator into the System Stack

A. User Interface

Traditionally, hardware devices are managed by kernel-mode drivers performing tasks such as memory protection, address translation, page pinning, and coordinating shared access. User/kernel mode task switching penalizes throughput for short running device tasks. POWER9 and z15 both have user-mode accelerator interfaces to reduce software overheads which we describe in this section. A novel user-mode page fault handler is also used in POWER9. Main goal of the user-mode page fault handler is to have low latency access to the accelerator by eliminating system calls, privileged instructions, and pinned pages. Retry capability of the accelerator helped accomplish that goal on POWER9. Performance of the user-mode interfaces are detailed in Section VIII-D.

The z15 architecture includes the ”Deflate Conversion Call” (DFLTCC) unprivileged instruction [38]. DFLTCC enables user-mode access to NXU with no operating system or device driver support. DFLTCC was implemented in low level firmware called millicode running on the CISC type processor cores [41]. Millicode interacts with the NXU on the same chip and performs essential functions such as checking access rights, virtual to absolute address translation, and exception handling.

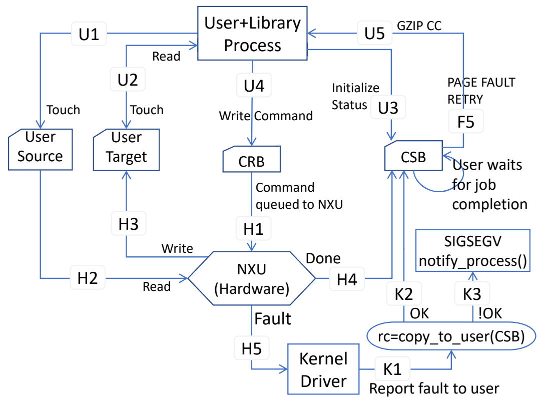

Fig. 11. User-mode page fault handler control flow

Since compression is a complex operation with many arguments, both z15 and POWER9 architected their own message formats for communicating with NXU. On z15, the DFLTCC instruction's general purpose register arguments contain the user source and target buffer addresses/lengths, and the sliding window address, and a Deflate specific parameter-block [38]. The POWER architecture, in contrast, uses a 128 byte length message format called *Coprocessor Request Block* (CRB). CRB has fields for device function codes, source, target, and parameter-block memory addresses, similar to that of z15. User threads atomically post CRBs to NXU job queues, using the POWER9 "copy-paste" instruction pair bypassing the OS kernel [42], [43].

POWER9 NXU has two 256 deep hardware managed job queues (high and low priority). The queues are mapped to the user address space with the mmap() system call. Synchronization among multiple processes and threads is handled in hardware; semaphores and mutexes are not necessary to share NXU. Multiple processes have fair access to the NXU thanks to a "job-limit" register that suspends long running jobs. When suspended, NXU dumps its internal state to the parameter-block in caller's memory. The dumped state includes partial checksums and stream offsets. The software thread resumes a job by submitting it again with the NXU state restored through the same parameter-block.

Another difference between z15 and POWER9 is that a POWER9 processor thread can send jobs to any other chip's NXU which can be used for balancing accelerator workloads and increasing overall throughput. The POWER9 processor includes a "Nest MMU" (NMMU), and an "effective to real address translation" table (ERAT) enabling virtually addressed NXU [42].

A notable design goal of these features and functions in both z15 and POWER9 is that NXU are accessed in the user-mode without making system calls and without page pinning which are necessary for low latency access (any system calls are made once per device initialization.)

B. User mode page fault handling

While z15 does not need OS or device drivers, POWER9 needs kernel support to handle page faults caused by NXU. To minimize Linux kernel code changes, we implemented a user-mode page-fault handler in the compression library [40]. Figure 11 shows the handler control/data flow. The user library touches one byte in source and target data pages to make them present in memory; no new kernel support is required for this step. Posting a CRB to the NXU queue starts the job (steps U1-U4 in Fig.11). NXU depends on NMMU (Fig. 3) and ERAT for virtual/physical address translation. In the common case, job executes without page faults (steps H1-H4, U5). Hardware interrupts are available through Linux signals but currently not used in the library.

When the system free page counts are low, the operating system may page-out user pages before a queued job starts. In the simple-fault case, NXU reports a missing page via the Command Status Block (CSB) found in the user memory (Fig. 11). Kernel is not involved in this code path, H4. The problem is if the CSB containing page was also paged out. Then, NXU interrupts the kernel driver (the H5 path in Fig. 11). The kernel driver pages in the CSB containing page and then copies the fault status back to the user thread waiting on job completion (steps H5, K1, K2, F5.) In both the H4 and H5 scenarios, the CSB status indicates that the user thread must retry the job.

To adapt to the low memory condition and to prevent infinite retries, the library shrinks the source and target buffer lengths and then retries the NXU job, effectively executing the original job using multiple smaller jobs. Suppose, an accelerator job requires 100 resident pages but some of them were paged out due to insufficient number of free pages. Through the retry mechanism, the library may cut the original job size down to 3 resident pages only, namely the source, target, and parameter-block pages, therefore making forward progress. The benefit of this approach is that the OS kernel is not involved in compression specific operations and page pinning is not required therefore enabling low latency access.

The value of user-mode access and page fault handling is also recognized in [29] for network adapters. User-mode page fault handlers have been used in other contexts. The *Userfaultfd* and *CRIU* mechanisms were proposed for live migration of virtual machines [44] and for applications to optimize access (e.g. caching, prefetching) to backing stores [45]. Our method is for managing page faults caused by a physical device and has no relation to those mechanisms.

C. Software Exploitation

Zlib API compliant libraries were implemented for leveraging the accelerators transparently [40], [46]–[48]. z/OS, z/Linux and AIX product level implementations are available. Power Linux implementation is going through the open-source upstreaming process. Existing zlib enabled applications can link to the accelerator enabled libraries without a new programming effort. New applications wanting to use compression can be coded for the zlib library virtually available on every O/S distribution. The accelerator can be leveraged when hardware is available.

Since Deflate is a widely adopted standard, general purpose cores and the on-chip accelerators may be employed in parallel for higher throughput. For example, one can choose to deploy the accelerator for compression-only since it can be much faster than an entire set of cores, whereas cores can be used for decompression-only.

VIII. Results: Microbenchmarks

Performance evaluation of the POWER9 and z15 on-chip accelerators is presented in this section. Accelerator throughput with a single thread and multiple threads and latency were experimentally determined with micro-benchmarks. End-to-end application performance of the POWER9 compression accelerator using the Spark TPC-DS benchmark is presented in Section IX.

A. Experimental Setup

1) Hardware: The POWER9 system model AC922 with two sockets (2 chips) is used. The system has 20 cores per chip operating at 3.8 GHz in simultaneous multi-threading (SMT4) mode with a total of 80 threads per chip. The POWER9 chip has one NXU accelerator operating at 2.0 GHz. Results reported here are for a single chip. The z15 experimental setup is an 8-core logical machine partition (LPAR) allocated from a single z15 chip. NXU runs at 2.5 GHz.

2) Software: Both POWER9 and z15 ran with Ubuntu 18.0.4 OS. POWER9 was booted with a Linux kernel and firmware that is customized for NXU [46], [47]. The POWER9 setup uses a zlib API compliant library written from ground up for supporting NXU [40]. The z15 setup instead uses the original zlib library with NXU support added [48].

The compdecomp_th.c utility was used for microbenchmarking [40]. For comparison, we linked the utility either with the original zlib 1.2.11 on general purpose cores, or with the accelerator enabled zlib-API compliant library. Single NXU throughput is measured in all cases.

For compression ratio measurements the GZIP, ZSTD v1.4.5, and LZ4 v1.9.2 command line utilities were used with default compression levels and options [8], [10], [12]. Silesia, Canterbury, and Calgary compression benchmark datasets were used [39], [49].

B. Compression Ratio

Figure 12 compares the compression ratio of LZ4, ZSTD, GZIP tools and libraries and the z15 NXU accelerator. (Note that results may vary depending on future revisions of hardware and software library). NXU is used in both the 2-pass and 1.1-pass modes for collecting symbol statistics as described in Section V-C. First observation is that the NXU results are 4-5% larger than GZIP -6, the default compression level.

The NXU 1.1-pass method (Section V-C) works well compared to the 2-pass method for all files except for *ptt5* and *pic*. Inspection of the two files reveal that the first 32KB in both files consist of almost entirely zeros which do not represent the remainder of the file contents. Therefore, the 1.1-pass method does not get a good sample and has worse compression ratio than the 2-pass method for those two files.

NXU performs worse than GZIP for *nci* and *xml*. *Nci* contains long runs of the triple "0␣␣". Since each hash table entry can only store 8 addresses in FIFO order (Section IV-C), the hash table loses track of long string matches. Likewise, *xml* contains many repetitive XML tags for which the NXU hash table does not perform well.

The LZ4 tool [10] has been optimized for speed and not for compression ratio which is also apparent in Fig.12. ZSTD compression ratio is generally comparable to that of GZIP except for *kennedy.xls*. ZSTD uses a 2MB sliding window (vs. 32KB in GZIP, ZLIB, and NXU, all Deflate based), which increases the probability of finding and encoding longer matches for some files, such as *kennedy.xls*. Additionally, ZSTD uses an entropy algorithm called *tANS* that can encode some highly compressible files with fewer bits per symbol than Huffman codes [13].

C. Throughput

Figure 13 shows the compression throughput of POWER9 and z15 NXU, and the zlib linked compdecomp_th utility running on the POWER9 cores in the SMT4 mode (1-80 threads). In the NXU case, processor threads share a single NXU. POWER9 NXU has a total throughput in excess of 7 GB/s for 64KB source size and larger. Since the zlib results are quite low, to highlight the ratio between zlib on cores and NXU we also present a set of speedup calculations on Table I (for 256KB source size) which show that NXU is 388x times faster than a POWER9 thread and 13x times faster than 80 threads on a POWER9 chip. In other words, a single NXU has the equivalent Deflate throughput of 13 processor chips full of cores which is substantial considering that it occupies less than 0.5% of the chip area.

z15 NXU compression throughput at 15 GB/s peak is double the POWER9 NXU throughput (Fig.13). As described in Section IV-D, z15 reduces compressor pipeline stalls by using 2-port SRAMs in the hash table, and the z15 NXU operating frequency is higher than POWER9 NXU.

Finally, Fig.14 shows the decompression throughput of POWER9 and z15 NXU and the zlib library running on the POWER9 and z15 cores. Table I shows that POWER9 and z15 NXU have 34 and 30-times speedup over a single thread, respectively. On POWER9, NXU decompression throughput is about 75% of the total throughput of 80-threads for large data. Note however that NXU is still faster than 1-thread zlib decompress for all data sizes.

D. Latency

Low latency is as important as high throughput to make an accelerator useful for a broad range of software applications. On Figs.13-14 we observe that with a single thread, POWER9 NXU and z15 NXU compression throughputs are low for small data sizes, then reach their peak at about 256 KB data size. Although a detailed accelerator model LogCA [50] exists, we used a simpler model to estimate the data granularity

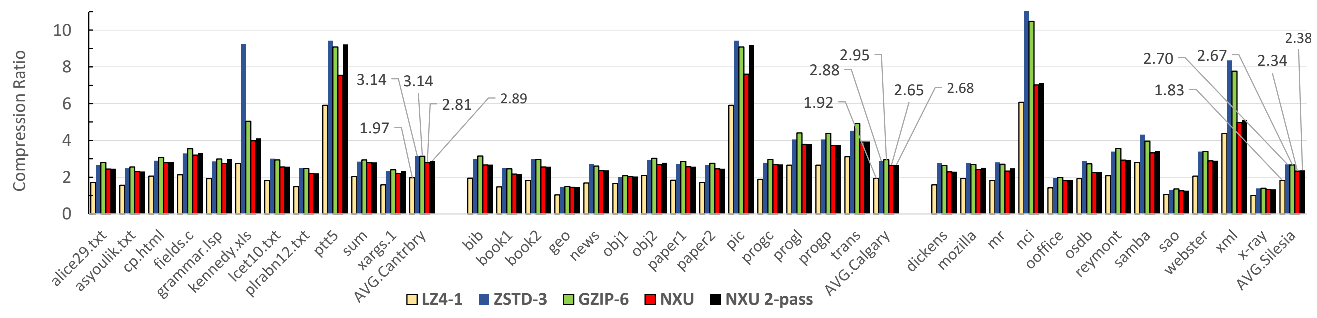

Fig. 12. Compression Ratio of three benchmark suites, Canterbury, Calgary, and Silesia with 5 different compression methods. (Note that both the GZIP utility and NXU use the Deflate compressed data format.)

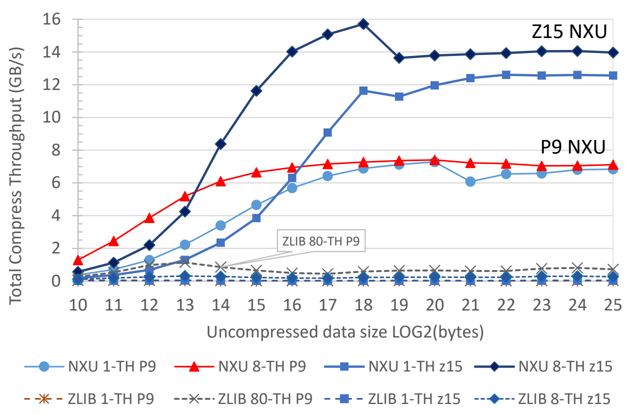

Fig. 13. Compression throughput of zlib on general purpose cores and POWER9 NXU and z15 NXU accelerators

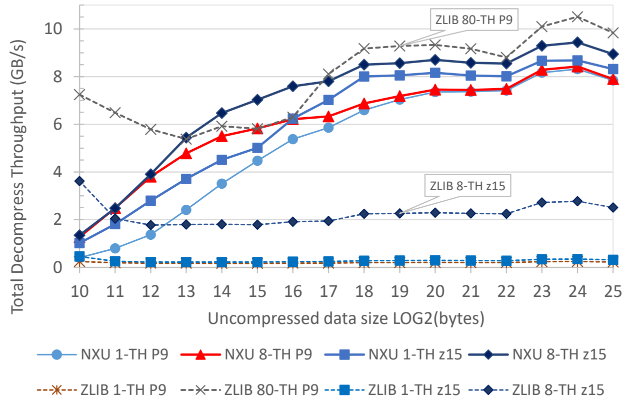

Fig. 14. Decompression throughput of zlib on general purpose cores and POWER9 NXU and z15 NXU accelerators

for efficient accelerator utilization. We model the accelerator execution time T with

$$T = T_0 + Data\_Size/PeakBandwidth$$

where the constant T_0 is the setup time which is the software and hardware initialization delays before the accelerator starts processing data. *Half_Peak_Data_Size* is a useful accelerator figure-of-merit (smaller is better.) When the data size is

$$Half\_Peak\_Data\_Size = T_0 \times PeakBandwidth$$

TABLE I

NXU SPEEDUP OVER ZLIB WITH 1, 8, 80 THREADS, AND LZ4 AND ZSTD TOOLS. SPEEDUP IS DEFINED AS THE SOFTWARE COMPLETION TIME DIVIDED BY THE NXU COMPLETION TIME

Arch.	Function	ZLIB 1th	8th	80th	ZSTD	LZ4
P9	Compress	388	49	13	32	19
z15	Compress	574	72	-	78	38
P9	Decomp.	34	4.6	0.75	8.3	2.8
z15	Decomp.	30	3.8	-	10	3.5

we get $T = 2T_0$ which means that half the time is spent in the setup and the other half in the accelerator doing actual processing. In other words, the user thread receives 50% of the ideal accelerator bandwidth when $Data\_Size = T_0 \times PeakBandwidth$. With smaller data, the accelerator is underutilized because the setup time T_0 dominates. With small data, one way to increase the accelerator utilization is by multiple threads sharing it. On Figs.13-14 we can determine the half-peak bandwidths and the half-peak data sizes and then arrive at T_0, given in Table II.

z15 compress single thread has the highest latency of $5.2\mu s$, higher than $2.2\mu s$ of POWER9. The z15 NXU support library is a port of the existing zlib source code which contains some software overheads not found on the POWER9 implementation of the zlib-API compliant library supporting NXU.

The setup time T_0 can be decomposed into two parts, the software setup time on core and the setup time in the NXU hardware. In a multithreaded access to NXU, software can run in parallel on different processor threads. Whereas the NXU components of the threads must run sequentially on a single NXU. Therefore, with multithreading software delays nearly disappear in the total delay calculations and the 8-thread latencies in Table II reveal the actual NXU hardware latencies in the $0.55 - 0.73\mu s$ range.

E. Power and Energy Efficiency

Figure 15 shows active power measurement of a POWER9 system [51] with the accelerator and with cores-only. One or 80 independent copies of the zlib/zpipe.c utility running on a single POWER9 chip are used. Active power is defined as total system power minus the idle system power when no workload is running. Throughput includes file system overheads. Energy

TABLE II
ACCELERATOR SETUP TIMES AND HALF-PEAK BANDWIDTH DATA SIZES

Arch.	Function	#Threads	HalfPkDataSz(KB)	Lat. T_0 (μs)
P9	Compress	1	16	2.2
z15	Compress	1	64	5.2
P9	Compress	8	4	0.55
z15	Compress	8	12	0.78
P9	Decomp.	1	24	2.9
z15	Decomp.	1	16	1.9
P9	Decomp.	8	6	0.73
z15	Decomp.	8	6	0.65

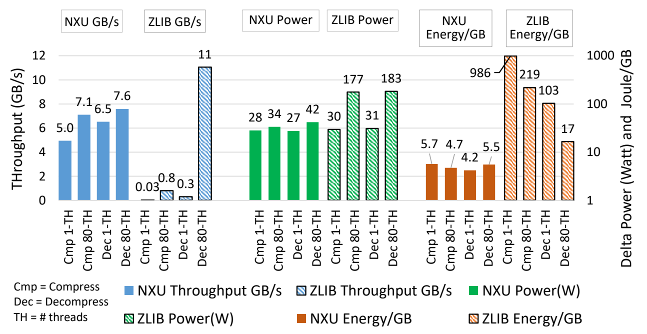

Fig. 15. Power/energy utilization of 1 NXU and 1 and 80 SMT4 threads on a single POWER9 chip are compared. Total system power minus the system idle power is shown. Energy unit is Joules per GB and calculated as Power/Throughput, the delta energy required to compress or decompress 1 GB of raw data.

per unit of data processed (Joule/GB) is calculated by dividing active power consumption with throughput. System power increased by 27 to 42 watts with the accelerator. A large fraction of that is due to processor, cache, and memory activity during the compress and decompress operations. Power and energy characteristics of the accelerator are very favorable compared to the cores. Active energy is up to 2 orders of magnitude less as the results show.

IX. Results: End to End Application Performance

Data compression is a vital component in large scale big data applications for improving storage space efficiency and application performance [52].

In this section, we first describe the integration of NXU into two components of the Hadoop ecosystem: Apache Spark [53], a popular distributed computing framework, and Apache Parquet [54], a columnar file format. Then, we evaluate the performance of NXU on Apache Spark with a popular industry-standard decision support benchmark, TPC-DS [55].

The accelerators speed up Spark by reducing storage I/O utilization and network traffic among the cluster nodes during the ”shuffle” and ”HDFS read/write” steps with a trivial amount of processor cycles. Terabytes of data are moved to/from storage and among Spark servers for each TPC-DS query. Spark uses software compression by default to reduce data amounts. Without acceleration, state of the art software compression codecs must use many processor cycles for compression therefore slowing down queries. Alternatively, compression quality is degraded to save processor cycles which results in higher I/O cost. Accelerators address both of these problems therefore speeding up Spark queries.

In order to integrate NXU to Java-based big data applications, we implement the common compression interfaces used by these applications in a thin JNI library that communicates with the NXU user-level libraries. Our JNI library implements two common streaming classes from Java standard library: `OutputStream` with a configurable block size to adjust maximum number of bytes to compress at once and `InputStream` for decompression. Additionally, it implements gzip compliant header and trailer for each block during compression and validates the integrity of uncompressed block using the NXU generated CRC at the time of decompression.

The stream classes are built on top of `compress` and `decompress` functions. For input and output buffers used in compression and decompression, implemented functions accept both primitive byte buffers that are allocated on JVM heap and off-JVM-heap allocated memory buffers. Such functions and stream interfaces provided in the JNI library allow us to integrate NXU to Apache Spark, Apache Hadoop (MapReduce and Hadoop Distributed File System), Apache Parquet (columnar storage format), Apache Kafka (a distributed streaming platform), Apache Cassandra (NoSQL database) and many other applications.

A. *TPC-DS Benchmark on Apache Spark*

TPC-DS is a de-facto standard benchmark in academia [52], [56]–[58] and industry [59]–[61] for evaluating performance of big data SQL platforms and underlying hardware infrastructure. The TPC-DS benchmark models a data warehouse for a large retail product supplier. It offers a dataset generator and 99 distinct SQL queries that cover the entire dataset. TPC-DS benchmark includes analytical queries from different workload classes, such as iterative OLAP, data mining and reporting. These queries are designed to perform an intense activity to stress the I/O, memory and CPU subsystem of the target database server.

TPC-DS benchmark interfaces with Spark SQL [62], a Spark module for relational data processing. Spark SQL integrates with various data sources. In our evaluation, we use Parquet files as data source because of its compatibility with Spark SQL, efficiency [63] and performance [64]. In the TPC-DS benchmark, we store the Parquet files in the Hadoop Distributed File System (HDFS) replicated across the cluster. HDFS is typically used as a storage backend for Apache Spark due to its high resiliency and scalability.

In Parquet files, data is horizontally partitioned into rows. Collections of rows form a row group and a group consists of a column chunk for each column. Parquet divides column chunks into column pages which is the smallest unit that must be read fully to access a single record. Compression and decompression are performed per column page. Hadoop libraries for Parquet, by default, define the page size as 1 MB and Snappy as the default compression codec.

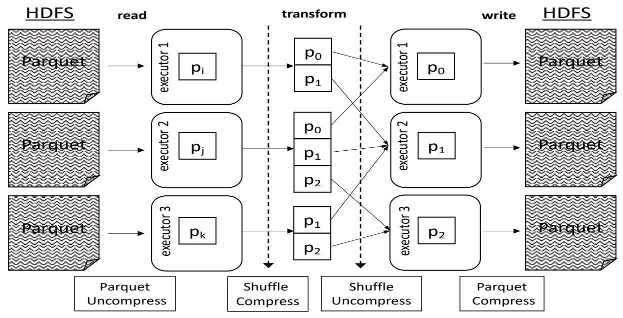

Fig. 16. Data compression pipeline in Spark and HDFS.

Fig. 16 shows data processing pipeline in Spark. Compression and decompression are used heavily in multiple components in the pipeline. Firstly, input and output files on HDFS are compressed and decompressed to save storage space and reduce I/O bandwidth usage. HDFS files are read and written sequentially on the disk. Secondly, compression is used to compress internal data such as in-memory data partitions, broadcast variables, and shuffle outputs.

Each dataset in Spark is represented as a data structure called resilient distributed dataset (RDD) [65], which is a fault-tolerant collection of partitions that can be operated on in parallel. Spark partitions RDDs into multiple partitions. A partition in Spark represents an atomic chunk of data. When a transformation (e.g join, groupbykey) is applied to the Spark dataset, the transformation is applied to each of its partition which will run inside the executor (JVM). Spark may require data from other partitions residing in another executor. Spark ”shuffles” the data to compute the transformation. In this case, Spark will collect the required data from each partition and combine it into a new partition. Shuffle operations involve heavy data copying between executors and random reads and writes to disk. During a shuffle operation, data is first written to disk and transferred across the network. In order to speed up the workload by reducing the disk and network I/O bandwidth consumption, Spark compresses the shuffle data while writing to the disk. Compressed data is transferred over the network and uncompressed at the destination executor. Since TPC-DS performs many complex transformative queries on the dataset, shuffle compression plays a critical role in end-to-end performance.

B. Experimental Setup

We run our Spark TPC-DS experiments on the following software and hardware stack:

- 4 IBM POWER9 LC922 servers (1 master, 3 worker nodes), 512 GB RAM, 2 chips (2 sockets), 20 cores per chip in SMT2 mode, 10x8TB 7200 RPM SAS 12Gb/s HDDs, 10Gbps private network
- Hadoop version 3.1, Apache Spark version 2.3.3
- TPC-DS version 1.4, 4 concurrent streams of 99 queries

On each chip on a worker node, we configure a Spark executor for each of the four TPC-DS streams. We exclusively allocate five cores per executor (JVM) and configure the executors to use the NXU local to the chip that they are running on. Therefore, we use all the cores and NXU units in the cluster to process the workload requests. We pick this Spark executor allocation and pinning strategy to run the TPC-DS workload because it delivered the best throughput for the default Spark in our experiments.

TABLE III
GENERATING 3TB TPC-DS DATASET(`SCALE FACTOR=3000`) WITH VARIOUS COMPRESSION CODECS.

Codec	Compressed(GB)	Ratio	Duration(hour)
Snappy	1019	2.94	4.1
Zlib	761	3.94	3.26
P9 NXU	804	3.73	2.9

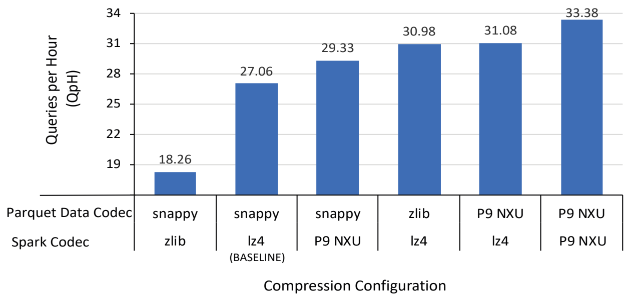

Fig. 17. Running TPC-DS benchmark (4 concurrent streams) on Spark SQL with various compression configurations. Each stream contains 99 queries in the order specified by TPC-DS specification.

1) TPC-DS Data Generation: Table III shows the elapsed time of generating the TPC-DS dataset with the scale factor of 3000 using Apache Spark as the driver and Apache Parquet as the output format. Snappy compression codec is the default codec used for Parquet files. Compared to Zlib, Snappy is faster but provides lower compression ratio. We observe that heavy disk write I/O to HDFS is the bottleneck in TPC-DS data generation. Compared to Snappy, Zlib reduces the execution time by 20% during the data generation and provides 34% better data compression. This is because Zlib reduces the I/O bottleneck by shrinking the size of the data at the cost of CPU cycles – shifting workload bottleneck to CPU and compression speed. Using POWER9 NXU provides 29% speed-up over Snappy by alleviating the I/O bottleneck and improving the compression speed during the data generation.

2) Running TPC-DS benchmark queries: In our 3 data nodes cluster, each TPC-DS query stream is processed by 6 Spark executors running concurrently, which would unavoidably cause Spark in-memory data partitions to be shuffled within the Spark executors and possible data spills to the disk due to memory pressure. Also, during the execution of the queries, each stream may read and write a few thousand partitions depending on the tables used by the TPC-DS query which would cause heavy read and write I/O to the disk.

Fig. 17 shows queries executed per hour while running 99 queries in the TPC-DS benchmark with Spark SQL with 4 concurrent streams. time of the slowest stream to calculate the workload duration. We observe that using a strong compression algorithm for the Parquet data speeds up the workload execution time. Compared to the baseline, using Zlib as Parquet codec and LZ4 for Spark shuffle provides around 14% speed-up in execution time. However, using POWER9 NXU for both Parquet data and Spark provides 23% faster execution time compared to baseline by significantly reducing the I/O contention. Using POWER9 NXU for both Parquet data and Spark reduce the data transferred (read/write) during the shuffle by 32%.

X. Conclusions

While the value provided by compression is widely recognized, its application is often limited because of the high processing cost and the resulting low throughput and high elapsed time for compression intense workloads. To this end, this paper demonstrates POWER9 and IBM z15 novel on-chip compression accelerators that overcome the shortcomings of existing approaches. Overall, the on-chip accelerators advance the state of the art in terms of area, throughput, latency, compression ratio, reduced processor utilization, integration into the system stack, cost, and ease of use.

We see several opportunities as part of future work. One concern with a hard-coded accelerator is not being able to implement new compression algorithms. New algorithms are typically implemented to overcome the low throughput of general-purpose cores, however they often result in degraded compression quality. This paper's accelerator design with two orders of magnitude speedup over a core made the performance problem irrelevant. Furthermore, Deflate is a widely used standard which justified implementing a hard-coded accelerator on the chip. Improving compression quality however may still need a configurable/programmable accelerator or a compression-optimized ISA which we consider as valuable future research work.

In addition, we see plenty of other compression opportunities for capacity and bandwidth growth as part of future work. Caches, memory and I/O links are prime candidates for compression that can result in considerable performance improvement. For example, compression may be used to increase cache [66] and memory capacity [34], [35] and memory bandwidth [67]. Compression can also be leveraged to improve bandwidth in I/O links and on-chip/off-chip coherent fabrics.

Acknowledgements

We thank Charles Lefurgy for power/energy discussions, to K. Ekanadham for his formal verification efforts, and to the IBM engineers who took part in the design, verification, and software efforts.

References

[1] "Amazon S3 pricing and Amazon ElastiCache (US East)," https://aws.amazon.com/s3/pricing/, accessed: 2020-01-09.

[2] A. Sofia, C. Lewis, C. Jacobi, D. A. Jamsek, D. F. Riedy, J.-S. Vogt, P. Sutton, and R. S. John, "Integrated high-performance data compression in the IBM z13," *IBM Journal of Research and Development*, vol. 59, no. 4/5, pp. 7:1–7:10, 2015.

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

[4] W. Qiao, J. Du, Z. Fang, M. Lo, M.-C. F. Chang, and J. Cong, "High-throughput lossless compression on tightly coupled CPU-FPGA platforms," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 37–44.

[5] A. Martin, D. Jamsek, and K. Agarawal, "FPGA-based application acceleration: Case study with gzip compression/decompression streaming engine," *ICCAD Special Session C*, vol. 7, 2013.

[6] C. Berry, B. Bell, A. Jatkowski, J. Surprise, J. Isakson, O. Geva, B. Deskin, M. Cichanowski, D. Hamid, C. Cavitt, G. Fredeman, A. Saporito, A. Mishra, A. Buyuktosunoglu, T. Webel, P. Lobo, P. Parashurama, R. Bertran, D. Chidambarrao, D. Wolpert, and B. Bruen, "IBM z15: A 12-core 5.2GHz microprocessor," in *IEEE International Solid-State Circuits Conference (ISSCC)*, February 2020, pp. 54–55.

[7] P. Deutsch, "RFC1951 DEFLATE compressed data format specification version 1.3," https://tools.ietf.org/html/rfc1951, accessed: 2020-01-02.

[8] "Zlib: a massively spiffy yet delicately unobtrusive compression library," https://github.com/madler/zlib, accessed: 2020-01-13.

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

[10] Y. Collet *et al.*, "LZ4: Extremely fast compression algorithm," https://github.com/lz4/lz4, accessed: 2020-01-15, version v1.9.2.

[11] "A fast compressor/decompressor," https://github.com/google/snappy, accessed: 2020-01-13.

[12] Y. Collet and M. Kucherawy, "RFC8478: Zstandard compression and the application/zstd media type," https://tools.ietf.org/html/rfc8478, https://github.com/facebook/zstd, accessed: 2020-01-13, v1.4.5.

[13] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp, "The use of asymmetric numeral systems as an accurate replacement for Huffman coding," in *IEEE Picture Coding Symposium (PCS)*, 2015, pp. 65–69.

[14] J. Y. Kim, S. Hauck, and D. Burger, "A scalable multi-engine xpress9 compressor with asynchronous data transfer," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2014, pp. 161–164.

[15] K. B. Agarwal, H. P. Hofstee, D. A. Jamsek, and A. K. Martin, "High bandwidth compression to encoded data streams," Apr. 22 2014, US Patent 8,704,686.

[16] G. J. Cockburn and A. J. Hawes, "Method and arrangement for data compression according to the LZ77 algorithm," Jun. 19, 2007, US Patent 7,233,265.

[17] ——, "Circuit and method for use in data compression," Apr. 18, 2006, US Patent 7,030,787.

[18] ——, "Data parsing and tokenizing apparatus, method and program," Mar. 3 2009, US Patent 7,500,103.

[19] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[20] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[21] L. L. Larmore and D. S. Hirschberg, "A fast algorithm for optimal length-limited Huffman codes," *Journal of the ACM (JACM)*, vol. 37, no. 3, pp. 464–473, 1990.

[22] J. Van Leeuwen, "On the construction of Huffman trees," in *ICALP*, 1976, pp. 382–410.

[23] R. L. Milidiú and E. S. Laber, "The Warm-up algorithm: A Lagrangian construction of length restricted Huffman codes," *SIAM Journal on Computing*, vol. 30, no. 5, pp. 1405–1426, 2000.

[24] M. Buro, "On the maximum length of Huffman codes," *Information Processing Letters*, vol. 45, no. 5, pp. 219–223, 1993, (Note: see Theorem 2.1, "In the case that we only know a lower bound $p > 0$ on the minimum probability.").

[25] I. D. Scherson, S. Sen, and A. Shamir, "Shear-Sort: A true two dimensional sorting technique for VLSI networks," in *The International Conference on Parallel Processing*, 1986, pp. 903–908.

[26] I. D. Scherson, S. Sen, and Y. Ma, "Two nearly optimal sorting algorithms for mesh-connected processor arrays using Shear-Sort," *Journal of Parallel and Distributed Computing*, vol. 6, no. 1, pp. 151–165, 1989.

[27] H. Yu, H. Franke, G. Biran, A. Golander, T. Nelms, and B. M. Bass, "Stateful hardware decompression in networking environment,"

in *4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008, pp. 141–150.

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

[29] I. Lesokhin, H. Eran, S. Raindel, G. Shapiro, S. Grimberg, L. Liss, M. Ben-Yehuda, N. Amit, and D. Tsafrir, "Page fault support for network controllers," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 449–466, 2017.

[30] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, 2012, pp. 335–348.

[31] J. Dike, "A user-mode port of the Linux kernel." in *Annual Linux Showcase & Conference*, 2000.

[32] A. W. Appel and K. Li, "Virtual memory primitives for user programs," in *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 96–107.

[33] B. Blaner, B. Abali, B. M. Bass, S. Chari, R. Kalla, S. Kunkel, K. Lauricella, R. Leavens, J. J. Reilly, and P. A. Sandon, "IBM POWER7+ processor on-chip accelerators for cryptography and Active Memory Expansion," *IBM Journal of Research and Development*, vol. 57, no. 6, pp. 3:1–3:16, 2013.

[34] B. Abali, H. Franke, D. E. Poff, R. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith, "Memory expansion technology (MXT): software support and performance," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 287–301, 2001.

[35] B. Abali, H. Franke, X. Shen, D. E. Poff, and T. B. Smith, "Performance of hardware compressed main memory," in *Seventh International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2001, pp. 73–81.

[36] L. Coyne, J. Dain, E. Forestier, P. Guaitani, R. Haas, C. D. Maestas, A. Maille, T. Pearson, B. Sherman, C. Vollmar *et al.*, *IBM Private, Public, and Hybrid Cloud Storage Solutions*. IBM Redbooks, 2018. [Online]. Available: http://www.redbooks.ibm.com/redpapers/pdfs/redp4873.pdf

[37] D. Harnik, R. Kat, D. Sotnikov, A. Traeger, and O. Margalit, "To zip or not to zip: Effective resource usage for real-time compression," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013, pp. 229–241.

[38] IBM, Ed., *IBM z/Architecture Principles of Operation, Thirteenth Edition, SA22-7832-12*. IBM, 2019.

[39] "Silesia compression corpus," http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia, accessed: 2020-01-02.

[40] "NX-GZIP: POWER9 gzip engine documentation and code samples," https://github.com/libnxz/power-gzip, accessed: 2020-03-19.

[41] L. C. Heller and M. S. Farrell, "Millicode in an IBM zSeries processor," *IBM Journal of Research and Development*, vol. 48, no. 3.4, pp. 425–434, 2004.

[42] L. B. Arimilli, B. Blaner, B. C. Drerup, C. Marino, D. Williams, E. N. Lais, F. A. Campisano, G. Guthrie, M. S. Floyd, R. Leavens, S. M. Willenborg, R. Kalla, and B. Abali, "IBM POWER9 processor and system features for computing in the cognitive era," *IBM Journal of Research and Development*, vol. 62, no. 4/5, pp. 1:1–1:11, 2018.

[43] "IBM Power ISA version 3.0B," https://openpowerfoundation.org, accessed: 2020-01-03.

[44] M. Rapoport and J. Nider, "User space memory management for post-copy migration," in *10th ACM International Systems and Storage Conference*, 2017, pp. 1–1.

[45] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale, "UMap: Enabling application-driven optimizations for page management," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 2019, pp. 71–78.

[46] "NX-GZIP: Linux source code," https://github.com/hmyneni/linux/tree/4.19-nx-gzip-v2, accessed: 2020-01-03.

[47] "NX-GZIP: SKIBOOT firmware source code," https://github.com/hmyneni/linux/tree/4.19-nx-gzip-v2, accessed: 2020-01-03.

[48] "IBM Z hardware-accelerated Deflate," https://github.com/iii-i/zlib/tree/dfltcc, accessed: 2020-01-03.

[49] "Canterbury and Calgary compression corpora," http://corpus.canterbury.ac.nz/descriptions/, accessed: 2020-01-17.

[50] M. Altaf and D. A. Wood, "LogCA: A high-level performance model for hardware accelerators," in *44th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA 2017)*, 2017, pp. 375–388. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3079856.3080216

[51] E. J. Fluhr, R. M. Rao, H. Smith, A. Buyuktosunoglu, and R. B. Monfort, "IBM POWER9 circuit design and energy optimization for 14-nm technology," *IBM Journal of Research and Development.*, vol. 62, no. 4/5, pp. 4:1–4:11, 2018.

[52] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*, 2015, pp. 293–307. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789791

[53] "Apache Spark," https://spark.apache.org, accessed: 2020-01-01.

[54] "Apache Parquet," https://parquet.apache.org/, accessed: 2020-01-01.

[55] TPC, "TPC-DS website," http://www.tpc.org/tpcds/, accessed: 2020-01-01.

[56] M. Poess, T. Rabl, and H.-A. Jacobsen, "Analysis of TPC-DS: The first standard benchmark for SQL-based big data systems," in *2017 Symposium on Cloud Computing (SoCC'17)*, 2017, p. 573–585. [Online]. Available: https://doi.org/10.1145/3127479.3128603

[57] M. Poess, R. O. Nambiar, and D. Walrath, "Why you should run TPC-DS: A workload analysis," in *33rd International Conference on Very Large Data Bases (VLDB-07)*, 2007, p. 1138–1149.

[58] A. Dholakia, P. Venkatachar, K. Doshi, R. Durgavajhala, S. Tate, B. Schiefer, M. Sheard, and R. S. Sagar, "Designing a high performance cluster for large-scale SQL-on-Hadoop analytics," in *2017 IEEE International Conference on Big Data*, Dec 2017, pp. 1701–1703.

[59] "Apache Spark SQL TPC-DS on IBM POWER9," https://developer.ibm.com/linuxonpower/perfcol/perfcol-bigdata/, accessed: 2020-01-09.

[60] "Benchmarking big data SQL platforms in the cloud," https://databricks.com/blog/2017/07/12/benchmarking-big-data-sql-platforms-in-the-cloud.html, accessed: 2020-01-09.

[61] "Spark SQL adaptive execution at 100 TB," https://software.intel.com/en-us/articles/spark-sql-adaptive-execution-at-100-tb, accessed: 2020-01-09.

[62] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and et al., "Spark SQL: Relational data processing in Spark," in *2015 ACM SIGMOD International Conference on Management of Data*, 2015, p. 1383–1394. [Online]. Available: https://doi.org/10.1145/2723372.2742797

[63] "Spark + Parquet in depth," https://databricks.com/session/spark-parquet-in-depth, accessed: 2020-01-09.

[64] T. Ivanov and M. Pergolesi, "The impact of columnar file formats on SQL-on-Hadoop engine performance: A study on ORC and Parquet," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 5, p. e5523, 2020.

[65] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[66] S. Hong, B. Abali, A. Buyuktosunoglu, M. B. Healy, and P. J. Nair, "Touche: Towards ideal and efficient cache compression by mitigating tag area overheads," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*, 2019, pp. 453–465.

[67] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K.-H. Kim, and M. Healy, "Attache: Towards ideal memory compression by mitigating metadata bandwidth overheads," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'51)*, 2018, pp. 326–338.