Hardware Acceleration of Compression and Encryption in SAP HANA

Monica Chiosa[*]
ETH Zurich
monica.chiosa@inf.ethz.ch

Fabio Maschi[*]
ETH Zurich
fabio.maschi@inf.ethz.ch

Ingo Müller
ETH Zurich
ingo.mueller@inf.ethz.ch

Gustavo Alonso
ETH Zurich
alonso@inf.ethz.ch

Norman May
SAP SE
norman.may@sap.com

ABSTRACT

With the advent of cloud computing, where computational resources are expensive and data movement needs to be secured and minimized, database management systems need to reconsider their architecture to accommodate such requirements. In this paper, we present our analysis, design and evaluation of an FPGA-based hardware accelerator for offloading compression and encryption for SAP HANA, SAP's Software-as-a-Service (SaaS) in-memory database. Firstly, we identify expensive data-transformation operations in the I/O path. Then we present the design details of a system consisting of compression followed by different types of encryption to accommodate different security levels, and identify which combinations maximize performance. We also analyze the performance benefits of offloading decryption to the FPGA followed by decompression on the CPU. The experimental evaluation using SAP HANA traces shows that analytical engines can benefit from FPGA hardware offloading. The results identify a number of important trade-offs (e.g., the system can accommodate low-latency secured transactions to high-performance use cases or offer lower storage cost by also compressing payloads for less critical use cases), and provide valuable information to researchers and practitioners exploring the nascent space of hardware accelerators for database engines.

PVLDB Reference Format:
Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. Hardware Acceleration of Compression and Encryption in SAP HANA. PVLDB, 15(12): 3277 - 3291, 2022.
doi:10.14778/3554821.3554822

1 INTRODUCTION

The transition from traditional in-house database deployments to a Software-as-a-Service (SaaS) model has introduced architectural challenges and opportunities for the database ecosystem. While computing nodes can be time-shared by a much larger number of client instances, multi-tenancy often raises concerns and requires data to be secured at any given state. SaaS databases need to ensure that data is encrypted while in transit or at rest, with cloud vendors providing this option as a key element of cloud databases [6, 7]. Similarly, to optimize cost, data is often compressed on the way to storage [17], which also results in gains in the strict bandwidth limits in cloud deployments. If data is compressed and encrypted on the way out, it has to be decrypted and decompressed on the way in, demanding even more resources to deal with I/O.

Computing resources in the cloud are a commodity affecting the costs of the service. Even if compression and encryption are external to the database system, they compete with the engine for computing resources. Moreover, they occur on the latency-sensitive I/O path. I/O has been a crucial aspect of the design of database engines from their inception. While the reasons to optimize the I/O path have varied over time, it remains a critical aspect of a well functioning engine and plays a large role in its overall performance. In the cloud, storage disaggregation makes the I/O path even more critical due to the intervening network and the nature of cloud storage, very different from conventional local disks.

While posing these and other challenges, the cloud also offers interesting opportunities over conventional architectures. Hardware accelerators and specialized architectures coexist with traditional compute nodes. This rich environment allows different algorithms to be processed at different points in the architecture, with an increasing number of operations being offloaded away from the CPU. Among the options available today, FPGAs are increasingly being used to provide near-data and in-network processing capabilities in a variety of settings. For instance, Huang et al. [26] developed a write-optimized storage engine at Alibaba, X-Engine, that shows the efficiency of offloading the log-structured merge (LSM) tree compaction computation to an FPGA. Amazon provides a smart caching service for RedShift implemented with SSDs connected to an FPGA for SQL offloading [9]. Microsoft has used its Catapult deployment [23] to speed up access to key-value stores (KVS) via a Smart Network Interface Card (SmartNIC) implemented with an FPGA [48], and there are several proposals to use FPGAs to implement smart disks [40, 76], and smart storage [38]. Microsoft's Cipherbase project [6, 7] also analyzed the use of an FPGA as a hardware trusted module to process data inside the database engine in an encrypted form, such that parts of the data would remain encrypted while in memory.

Taking advantage of their growing availability in the cloud, in this paper we explore the use of an FPGA-based accelerator placed on the I/O path of SAP HANA to speed up compression and encryption in SAP HANA cloud deployments. The results we obtain

[*]Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554822

demonstrate the advantages of the design: we can efficiently compress and encrypt data on its way to storage (at up to 4 GB/s and 15 GB/s, respectively, and at around 4 GB/s when combined). We can also decrypt data read back from the storage at up to 15 GB/s. In all cases, we can process data at a higher rate and more efficiently than using the CPU, which can be one order of magnitude slower when having to perform both compression and encryption.

2 BACKGROUND AND RELATED WORK

In this section, we briefly describe SAP HANA, the compression and encryption algorithms involved, as well as related work.

2.1 SAP HANA

SAP HANA [22] is a column-oriented in-memory database specifically built to integrate both analytical and transactional workloads into a single engine. It can be deployed on-premise or as part of SAP HANA Cloud[1] as a SaaS. Together with SAP HANA, the cloud platform also provides SAP IQ, a robust, traditional disk-based database. This work is part of recent efforts from SAP to enhance their on-premise products with the opportunities offered by cloud deployments. For instance, Abouzour et al. [2] detail the process of porting a disk-based data management system to cloud object-storage. Complimentary to these efforts, this work deals with compression and encryption of data on the I/O path taking advantage of the new hardware options available in the cloud.

2.2 Compression Algorithms

Like most column stores and analytical engines, SAP HANA uses light-weight compression internally to optimize memory usage, memory bandwidth, and cache efficiency. The light-weight compression scheme depends on the data type of the column, and several of them are used: dictionary, N-bit, prefix, or run-length [46, 54, 64, 67]. These schemes have in common that they preserve both (i) efficient sequential and random access for the scans and (ii) point queries of analytical and transactional workloads; at the cost of a potentially low or moderate compression rate.

Compressing data on its way to storage is typically done using different methods. Being already in use in SAP HANA [70], we focus on the DEFLATE [19] method for heavy-weight compression. This algorithm is found at the core of *gzip*, *zlib*, or *zip*. The different formats are obtained by adding a header and a checksum to the raw compressed data to produce the corresponding file format.

The DEFLATE compression method was created to be independent from the CPU architecture, operating system, or file system, and offers a solution that accepts arbitrary-sized input data, making it suitable not only for storage, but also for data communication channels. The method consists of two parts: *LZ77 compression* and *Huffman encoding*. LZ77 compression maintains a sliding window over the uncompressed data and replaces repeated occurrences of bits with references to the previously encountered same sequence of bits [80]. Huffman encoding aims to reduce the size of the data stream by encoding symbols with varied-length codes based on their frequency [27]. These codes are created using a Huffman tree as follows: the most frequently encountered symbols receive a shorter code, whereas the rarely encountered symbols receive a longer code. The Huffman tree can be created dynamically, a new Huffman tree is generated for each new input data block, or statically, the tree is generated based on statistical information over the data and used for multiple input data blocks. The trade-off between the two tree types comes down to compression ratio quality vs. computing latency at the pre-processing phase of the compression.

These methods are extensively used by database engines; however, only as software implementations. For instance, MariaDB and MySQL's storage engine, InnoDB, support the zlib library with its LZ77 compression algorithm and report a 50% reduction factor in the data size [56]. Nevertheless, they also report a performance hit for write-intensive applications such as OLTP due to the compression algorithm's high computing demands. The same compression library is used by PostgresSQL to enable compression of data dumps [66]. NoSQL databases such as MongoDB or the WiredTiger storage engine offer support for snappy[2], zlib, and zstd. Snappy is their default compression algorithm, whereas zlib requires more computing power, but yields a better compression ratio [53].

Related Work. Given the penalties imposed on both memory and communication bandwidth on the I/O path, there has been extensive work to optimize compression algorithms. Not only DEFLATE, but also other alternatives, such as lz4[3] and snappy. They, however, compromise compression ratio in benefit of performance, so they are not an alternative to DEFLATE in all use cases. Some algorithms try to optimize for both metrics by specializing to specific data types, such as FSST [12], a compression scheme that supports efficient random access to individual strings. In this paper, we aim to support the full spectrum of DEFLATE compression, without redefining the scope of data types. As we target block-based I/O operations, support for random access is not required, and higher compression ratios are preferred.

Being extremely computing expensive, and naturally placed in the hot-path of data communication, considerable efforts have been put on porting these software compression algorithms to hardware [20, 42, 72]. Microsoft [24] presented an FPGA implementation of DEFLATE where the resource utilization per compression ratio is minimized thanks to their optimized matching engine, achieving also the best public known compression throughput. Unfortunately, their implementation is not open source to allow us a direct comparison. Intel [1] proposed a very efficient open source implementation of the DEFLATE algorithm. Intel's version is the starting point for our compression module that incorporates several important optimizations (Section 3.1).

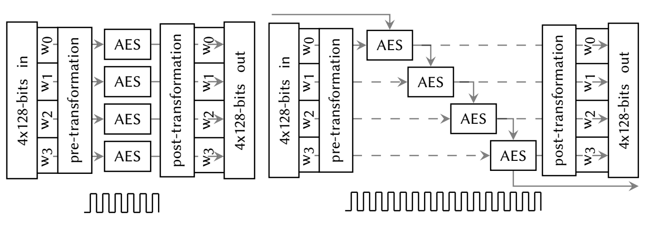
Some attempts to implement INFLATE — the decompression side of DEFLATE — on hardware have been made [44], however none of them solved the intrinsic complexity of LZ77 decompression and Huffman decoding without having to resort to a different decompression algorithm [21, 62]. In this work, we rely on the CPU implementation of the decompression.

More valuable than a comparison of compression efficiency and throughput over artificial corpuses, in this work we introduce the challenge of having to deal with real-world data. In fact, the majority of the related work reports only the maximum throughput achieved in the best conditions, where there is enough data to be processed

[1]https://www.sap.com/products/hana/cloud.html

[2]https://google.github.io/snappy

[3]https://lz4.github.io/lz4



(a) parallelizable

(b) non-parallelizable

Figure 1: Block diagram of one AES-256 module operating in parallelizable and non-parallelizable modes while processing a 512-bit cacheline (64 bytes).

by the hardware engine. In this paper, we explore the challenges introduced by dealing with database I/O in terms of information entropy, size variance, and irregular arrival rates.

2.3 Encryption Algorithms

The Advanced Encryption Standard (AES) [16] is the most widely used block cipher encryption standard, having its own set of dedicated instructions in both standard CPU [28] and embedded [8] architectures, and being generally adopted also by database engines. AES is a symmetric key algorithm with three possible *initial key sizes*: 128 bits, 192 bits, and 256 bits. We focus on the 256-bit implementation and define the process of encryption or decryption of a cacheline (512 bits) as AES-256.

The algorithm's design is based on a chain of substitutions and permutations, known as *transformation rounds*, which makes it suitable for both efficient software and hardware implementations. However, its block cipher modes of operation cause significant performance differences due to the resulting implementations. We show these differences by implementing three AES block cipher modes: Electronic Code Book (ECB), Counter (CTR), and Cipher Block Chaining (CBC). The amount of data that has to be encrypted or decrypted together represents the transformation granularity of a block. For CTR and CBC modes it is given by the data block size, whereas for the ECB mode, the granularity is the 128-bit data word.

Figure 1 illustrates the processing of a 512-bit cacheline, with each AES module consuming one 128-bit data word at a time. We differentiate between the parallelizable modes in Figure 1a (ECB and CTR for both encryption and decryption and CBC only for decryption), and the non-parallelizable mode in Figure 1b (CBC for encryption). ECB is the simplest of the modes, requiring only the transformation rounds for both encryption and decryption, whereas CTR and CBC modes require pre- or post-transformations of the data word before entering or after exiting the transformation rounds. For its pre-transformation, CBC encryption needs the previous encrypted word, thereby creating a data dependency between consecutive iterations. As a result, while the number of transformation rounds does not have an impact on the performance of the parallelizable modes (ECB and CTR), the encryption performance in CBC mode is significantly affected by it.

Many database systems have software implementations of encryption as well. Oracle uses all three initial key sizes (128, 192 or 256 bits) of the AES algorithm for its Transparent Data Encryption (TDE) [60], employing the 256-bit initial key only for highly sensitive data [61], and relies on Intel's intrinsic instruction set to boost encryption and decryption performance for ECB, CTR and CBC modes. MySQL (v8.0) supports AES encryption in ECB and CBC modes using all three initial key sizes, with ECB and 128-bit initial key being the default block cipher mode encryption option [55]. The same encryption algorithm, AES and 256-bit initial key, is also used by Azure's TDE for its SQL Server, Azure SQL Database, and Azure Synapse Analytics services [51]. SQL Server relies on the AES modes supported by the Microsoft encryption library: ECB, CBC, CFB (cipher feedback) and OFB (output feedback).

From data security point of view, out of the three modes, ECB is the most prone to failures under attacks due to its simple architecture that can map properties of the input word into the output word. CBC mode is the most robust of them, but at the cost of reduced parallelism. CTR mode is the intermediate point between security and performance, providing a possible solution for database encryption, as suggested by HighGo for their PostgreSQL solution [75].

SAP HANA has the option to encrypt data at rest using CBC mode on the CPU, with at most two threads allocated to the task. The number of allocated threads is such that the encryption does not interfere with query processing. This paper is part of an effort to consider modes more amenable to parallelization than CBC mode.

Related Work. Much attention has been given to AES implementations on FPGAs. Xilinx [77] proposes their own proprietary AES module focusing only on the parallelizable modes. Our open source design[4] achieves the same order of magnitude in terms of throughput performance as Xilinx's custom module. Other efficient AES implementations focus mainly on enhancing the encryption/decryption of a single 128-bit word for FPGAs [14, 15, 39, 49], or ASICs [3, 25], and put less consideration to the block cipher modes, the non-parallelizable modes and the interaction of the AES module with other computing modules. Kara et al. [41] show that AES-256 CBC decryption placed prior to a stochastic coordinate descent (SCD) computation increases the processing time by x3.6, despite the existence of dedicated CPU instructions set for decryption, and propose the offloading of AES-256 CBC decryption transformation rounds to the FPGA, while keeping the key expansion on the CPU. In contrast, in this paper we develop a complete solution for AES-256 CBC encryption and decryption, while offloading the entire computation (key expansion and transformation rounds) to the FPGA. Moreover, beside the CBC mode, we also analyze the performances of two additional modes, ECB and CTR, and we distinguish between the block sizes performance.

2.4 Hardware Acceleration for Databases

Field programmable gate arrays (FPGAs) have emerged as one type of accelerator for data processing [73]. They consist of matrices of logical elements whose behavior can be reconfigured over time. Having a spatial architecture, FPGAs allow algorithms to be executed with a great degree of parallelism at line-rate, for example when attached to the network as SmartNICs. Cloud computing made FPGAs not only affordable, but also convenient to deploy close to traditional systems as they are migrated to the cloud [37].

[4]https://github.com/fpgasystems/hw-acceleration-of-compression-and-crypto

Related Work. Ringlein et al. [68] show the advantages (cost reduction, tail-latency minimization, execution efficiency) of using FPGA-based Function-as-a-Service into disaggregated cloud infrastructures, and Zha and Li [78] build an abstraction framework for virtualizing heterogeneous cloud FPGAs. Key-value stores also benefit from FPGAs with new memcached architectures [11], native computational storage [74], and hardware-managed transactions [36]. Kara et al. [41] show how FPGAs can be used for coupling column-store ML algorithms with on-the-fly data transformation, such as decryption and delta-encoding decompression, while Lasch et al. [43] accelerate Re-Pair compression algorithm using FPGAs.

Zheng et al. [79] propose that by offloading the physical storage utilization efficiency to the modern SSDs (with built-in transparent compression), a DBMS can obtain higher performance and simpler data structures and algorithm complexity. Zuck et al. [81] study various approaches of implementing transparent compression in the firmware of SSD devices, either in the context of database systems or in that of filesystems. Complementary to them, we show how encryption can be added on top of compression, and offloaded to an intermediate device that can be positioned between compute and storage, leading to a more modular and flexible system.

Lee et al. [45] show how offloading online analytical processing (OLAP) operations to a near-memory accelerator (an FPGA board with attached DIMMs) that sits between CPU and main memory eliminates the performance degradation of online transactional processing (OLTP) workloads when co-existing with OLAP workloads. While Lepers et al. [47] show that the CPU is the bottleneck and not the storage device for tree compaction in an LSM KVS on NVMs SSDs, Huang et al. [26] address this issue by accelerating compaction computation on the FPGA.

In the cloud, Microsoft Azure uses FPGAs to offload network management functionality [23] and accelerate key-value store applications [48], and Amazon Aqua offloads parts of SQL to a network-attached FPGA-based caching layer [9]. Intel's QuickAssist Technology [34] offer acceleration solutions for lossless data compression and symmetric and asymmetric data encryption, but they do not give details about design, implementation, or performance.

3 DESIGN

In this section, we describe the implementation of compression and encryption on an FPGA, as well as the architectural strategies needed to achieve an efficient and compact pipeline that combines both designs. Since we are interested in exploring the effects of acceleration, we focus on the compression and encryption operations *per se* and explore the throughput they can achieve regardless of the type of storage used (local disk or network-attached storage).

3.1 Compression/Decompression

We took the design proposed by Abdelfattah et al. [1] as our starting point given its high-level language implementation and the best compression throughput of all open source solutions. Then we modified the design to adapt it to a database context and combine it with the encryption module.

The base implementation decomposes the DEFLATE execution into five OpenCL kernels (Figure 2). FPGA external memory accesses are handled by dedicated kernels (*Read data*, *Load Huffman tree*, *Send to DDR4*), so FPGA resource allocation can be optimized for those specific read and write kernels, and a high operating frequency can be achieved [35]. The communication and synchronization between kernels are done through low-latency channels acting as blocking FIFOs.

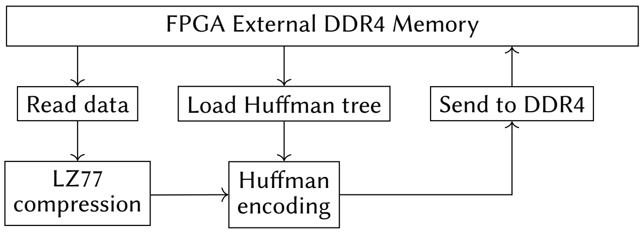


Figure 2: Compression block diagram.

The compression of a client payload, i.e., a database page, is sliced into several transactions. Different kernels can process different transactions from the same client payload in parallel. The payload is sent to the FPGA external memory, and all kernels receive several control parameters, notably the memory pointers and the input size to be processed. To maximize memory read bandwidth, the execution is carried out by transactions of 256 bits. Each computing kernel (*LZ77 compression*, *Huffman encoding*) receives a transaction from the input FIFO, processes it, and sends the resulting transaction to the next kernel via the output FIFO. The execution finishes as soon as the storage kernel (Send to DDR4) finishes writing the last piece of compressed transaction into the main memory.

From this base implementation, we enhanced a few aspects of the design. Firstly, we introduced a stateless control flow to different kernels. In this way, each kernel processes each transaction individually, acting as a steady stream processor. Note that, by the nature of compression, both the LZ77 compression and Huffmann encoding kernels output less data than they receive as input. Secondly, to achieve a stream dataflow architecture, we make the execution independent of the client payload size by augmenting the transaction data with a control signal that indicates the end of the payload processing. As long as the new transaction does not carry the *last* flag, the kernel executes it as being part of the same payload as the previous one.

The original and traditional method to invoke FPGA kernels requires synchronization and communication overheads between the host and the FPGA observes the following pattern: (i) the host sets the control parameters of all kernels; (ii) the host sends the data payload to the FPGA external memory; (iii) the host invokes the kernels; (iv) kernels execute; (v) the host polls the FPGA to detect the end of the execution; (vi) the host reads back the output payload.

By having stateless control flow in the FPGA, introduced by the *last* flag, the intermediate computing kernels can run continuously and process any transaction of data they receive without the need of synchronization with the host. These are called *autorun* kernels [30]. Here, the utilization of a high-level language such as OpenCL creates a common challenge to this type of modification: the actual hardware implementation generated by the compiler is far away from the high-level abstraction modeled by the language, and programmers lose all the control of the final hardware

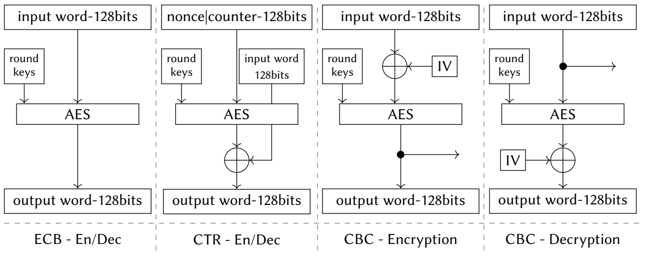


Figure 3: AES block cipher modes for Encryption (En) and Decryption (Dec).

topology [18]. By simply transforming a traditional kernel into an autorun kernel, the compiler can introduce drastic changes to the topology and ruin any optimization attempt. We succeed to transform two kernels (LZ77 compression, Huffman encoding) as autorun kernels by re-designing the memory initialization to a static section of the autorun module.

With these two modifications, a stateless control flow of autorun kernels allows us to reduce the overhead of calling the FPGA by batching calls: the CPU does not need to send individual I/O requests to the FPGA; it can buffer a variable amount of them and invoke the FPGA only once (Section 4.6). SAP HANA already does this to a certain extent and the mechanism described makes the implementation of compression suitable for such batching.

3.2 Encryption/Decryption

Through our encryption/decryption implementation, we present a comparison between three different AES block cipher modes, parallelizable and non-parallelizable, and analyze how they interact with a compression module inside an FPGA processing pipeline.

Our encryption/decryption FPGA design aims to achieve the following properties: (i) Reusability and modularity of the code, irrespective of the AES block cipher mode and initial key size, thereby making the code base less error-prone and more tractable; (ii) Control over the operations happening at each pipeline stage of the design to maximize the overall throughput; (iii) Flexibility regarding the initial key size and the number of 128-bit input data words that can be processed in a transaction to ensure wider applicability.

AES processing involves a sequence of transformations: substitution, shift, bitwise operation, and polynomial multiplication (known as a transformation round), that are consecutively applied to each 128-bit input data word. The initial key size determines the number of transformation rounds needed to encrypt or decrypt an input data word. For a 256-bit initial key size, 14 transformation rounds are needed, and each transformation round requires a dedicated 128-bit *round key* for its internal bitwise XOR operations. All the required round keys are obtained from the initial 256-bit key through a key expansion process that is independent of the 14 transformation rounds. Together with the pre- or post- data transformation operations, the 14 transformation rounds represent the processing *datapath* of encryption/decryption.

On the FPGA, we implement each AES module shown in Figure 1 in a pipeline fashion, with each stage of the pipeline applying a transformation round to the 128-bit data word. Therefore, the depth of the pipeline equals the total number of transformation rounds, namely 14 stages. In Figure 3 we illustrate the block diagrams for the three AES block cipher modes we implement (ECB, CTR, CBC) and emphasis their common points: (i) transformation rounds (grouped under the AES name) and (ii) round keys (generated by the key expansion); and differences (presence or absence of the pre- or post-processing data transformation operations - XOR operations).

We offload the entire encryption and decryption computation to the FPGA, namely both transformation rounds and key expansion, and design their corresponding modules as parameterizable modules in RTL (Register Transfer Level). We use VHDL as modeling language to maximize the throughput [52] for the transformation rounds. All the modules, transformation rounds for encryption and/or decryption and the key expansion for encryption and/or decryption, are part of an OpenCL library and can be instantiated into OpenCL FPGA kernels as function calls, abstracting away the VHDL implementation from the user. The user can set at FPGA compilation time the following programmable parameters: *OPERATION* (0-encryption, 1-decryption); *N_WORDS* (number of 128-bit words processed in parallel), four 128-bit words (512-bits) represent our default choice and the granularity for FPGA external memory read and write operation; *KEY_WIDTH* (128, 192, or 256), for the purpose of this work we use only the 256-bit initial key size; and *MODE* (0-ECB, 1-CTR, 2-CBC), for choosing the block cipher mode.

Listing 1: Connection between OpenCL function call and the corresponding VHDL module.

```
<RTL_SPEC>
    <FUNCTION name="aes_256" module="aes_user_intel">
        ...
        <PARAMETER name="OPERATION" value="0" />
        <PARAMETER name="MODE" value="2" />
    </FUNCTION>
    <FUNCTION name="aes_key_256" module="key_user_intel">
        ...
        <PARAMETER name="OPERATION" value="0" />
    </FUNCTION>
    <FUNCTION name="aes_256_decrypt" module="aes_user_intel">
        ...
        <PARAMETER name="OPERATION" value="1" />
        <PARAMETER name="MODE" value="2" />
    </FUNCTION>
    <FUNCTION name="aes_key_256_decrypt" module="key_user_intel">
        ...
        <PARAMETER name="OPERATION" value="1" />
    </FUNCTION>
</RTL_SPEC>
```

Listing 1 illustrates how the connection between the OpenCL function call and the VHDL module is done through an XML kernel description file.*The <FUNCTION>...</FUNCTION>* element defines the scope of each function, transformation rounds or key expansion for encryption or decryption. Within the opening tag, the *name* attribute identifies the function call used in the OpenCL kernel and the *module* attribute identifies the VHDL module that describes the function implementation (e.g., *name* = *aes*_256 and *module* = *aes_user_intel* represent the transformation rounds function call and the VHDL module, respectively, for encryption). Through Listing 1 we want to emphasize the programmability of the FPGA design. Note that for different OpenCL function call names we use the same VHDL module name with different settings for the programmable parameters, e.g., for decryption we use the same *module = aes_user_intel* and a different function call *name = aes_256_decrypt*.

Beside the programmable parameters that can be set using the *PARAMETER* element, the XML file also exposes the characteristics of each VHDL module: expected latency, stall free or not, stateful or stateless, the communication interface between the VHDL module itself and the other operations inside the OpenCL kernel [33], and the list of all RTL files that are describing the module's behavior. Since the OpenCL kernel is implemented into hardware as a pipeline similar to the instruction pipeline of processors, all these VHDL module details are necessary such that the module's pipeline itself is integrated into the OpenCL kernel pipeline.

The datapath for AES-256 is stateless, input data being streamed through the encryption or decryption pipelines. For each stage of the pipeline, two stateful objects are needed: an 8-bit substitution box for adding randomness to the transformation round, and a round key from the key expansion. The information for the round keys is stored in registers, whereas the 8-bit substitution box preserves its state using look-up tables.

The latency of the AES-256 FPGA module is mainly determined by whether a parallelizable or a non-parallelizable block cipher mode is used. ECB and CTR modes have 15 clock cycles of latency for processing one cacheline (4x128-bit input data words), for both encryption and decryption. For the same cacheline, the CBC mode needs 15 clock cycles for decryption, which is parallelizable, and 57 clock cycles for encryption, which is non-parallelizable. The latency for one AES module can be manually computed, one clock cycle for each transformation round plus one extra clock cycle for delivering the results. The key expansion computation takes 11 clock cycles on hardware, but since it is independent of the datapath, it can be ignored when computing the latency of the results. Seen from the host, the key expansion latency is 62.90 µs and includes beside the 11 clock cycles of hardware latency, also the invocation of the kernel from the CPU when an I/O request is made.

3.3 Compression and Encryption

We take advantage of the OpenCL environment versatility to combine modules written in different languages. Compression is built as an OpenCL kernel and encryption is built in VHDL, but integrated in OpenCL as a library and exposed to the system kernels as a function call. The result is a combined operator that compresses data and then encrypts it when invoked by the database (Figure 4).

When used in isolation, the compression module sends the results directly to the FPGA external memory. When plugged upstream the encryption module, the compressed transaction is forwarded via FIFO buffers to the AES-256 module(s). The encryption module uses the *last* flag that goes along with the compressed transactions to determine the granularity of the operation, i.e., a compressed payload is encrypted as a single unit until the last flag is met. When the last flag is met, the encryption module resets its nounce and counter for CTR mode, and restarts from the initialization vector for CBC mode. The last flag has no reset effects on ECB mode, since its granularity is always 128-bit word size.

For the two parallelizable encryption modes, ECB and CTR, a single AES-256 module suffice to consume data coming from the compression module without putting back-pressure on the upstream

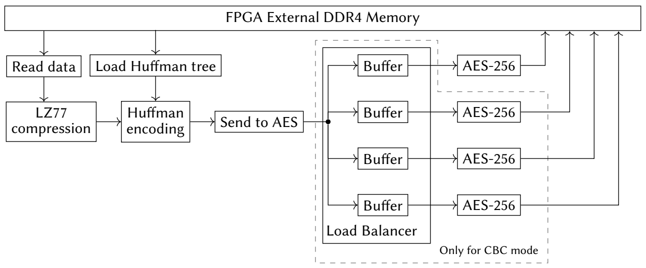


Figure 4: Compression and encryption pipeline block diagram for the three block cipher modes.

module, as it is later shown in Section 4.5. The sequential nature of the encryption in CBC mode creates back-pressure on the compression module, that is, the encryption — the second stage in the pipeline — is slower than the compression — the first stage in the pipeline. We minimize this back-pressure by using multiple AES-256 modules and a load balancer that round-robins the input between them. The goal is to eliminate the back-pressure by matching the compression throughput with several AES-256 modules. However, as a spatial architecture, we can only put several of them into a given FPGA. For the FPGA we use, we are able to deploy up to four parallel AES-256 modules in CBC mode.

3.4 FPGA resource consumption

We implement our designs on an Intel FPGA PAC D5005 Acceleration Card. Table 1 summarizes the FPGA resource utilization, including the resources allocated for the static part of the FPGA, called Board Support Package (BSP), and the dynamic part of the FPGA, called User Kernel System (where our designs reside). The interaction between these two parts is illustrated in Figure 5.

The BSP represents the FPGA logic dedicated to the FPGA interaction with external elements, host processor and DDR4 memory, and consists of the host interface, the external memory controller, and the global memory interconnect. The User Kernel System part contains the logic dedicated to the compute kernels, e.g., AES-256, LZ77 compression or Huffman encoding, and to the kernel communication with the on-chip FPGA memory.

We report the working frequency for each standalone and combined implementations. Unlike the CPU that operates in a GHz frequency range, the FPGA operates in the MHz frequency range. Our designs have a working clock frequency around 240 MHz. Despite this enormous frequency range gap, the FPGA is able to achieve

Table 1: FPGA Resource Consumption.

FPGA Design	ALUT	Logic Utilization	RAM	Op. Freq.
CBC AES-256	259,353	256,438 (27%)	667 (6%)	248 MHz
CTR AES-256	236,211	252,822 (27%)	899 (8%)	248 MHz
ECB AES-256	252,587	258,754 (28%)	899 (8%)	232 MHz
DEFLATE	282,464	263,087 (28%)	2,161 (11%)	287 MHz
DEFLATE + 1 CBC	458,061	428,244 (46%)	2,298 (20%)	243 MHz
DEFLATE + 4 CBC	712,081	678,698 (73%)	2,460 (21%)	243 MHz
DEFLATE + CTR	435,919	409,742 (44%)	2,298 (20%)	244 MHz
DEFLATE + ECB	451,274	415,508 (45%)	2,298 (20%)	236 MHz

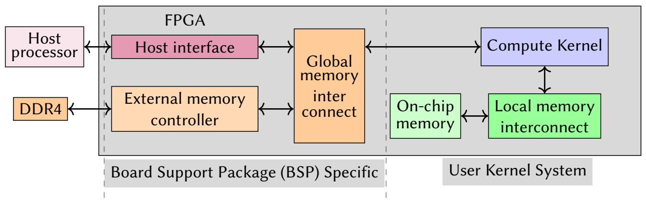

FPGA
Host processor
Host interface
DDR4
External memory controller
Global memory inter connect
Compute Kernel
On-chip memory
Local memory interconnect
Board Support Package (BSP) Specific
User Kernel System

Figure 5: OpenCL FPGA framework overview.

better performance than the CPU when the FPGA is implementing algorithms that can benefit from its spatial, parallel architecture and low-latency on-chip memory accesses, providing a MIMD (Multiple Instructions, Multiple Data) analogy. The CPU, however, must operate at a much higher frequency to overcome several overheads imposed by its architecture [10], and the best parallelism it can achieve is restricted to SIMD (Single Instruction, Multiple Data).

The OpenCL compiler determines the highest possible operating frequency based on (i) the design characteristics, and (ii) the placement as physical components (a non-deterministic operation called *place and route*). For example, even if different block cipher modes of the AES-256 modules have similar designs, their final resource allocation, and therefore placing and routing, is unlikely to be equivalent, leading to different working frequencies. The highest working frequency is obtained by the compression module, 287 MHz. The logic utilization in Table 1 represents the number of adaptive logic modules the design needs, with each adaptive logic module being used either as a 2-combinational Adaptive Look-Up Table (ALUT), as a 2-bits full adder or as four registers [29]. In all cases, the resources used leave enough room for additional logic (e.g., a network stack [69] to send the data to cloud storage through the network), even when using 4 AES-256 CBC encryption modules.

4 EVALUATION

We evaluate the performance of each module in isolation and when combined. As baseline, we run their equivalent software implementations using typical configurations currently used in SAP HANA. For the modules on the write path of the I/O request, we focus on the compression and the encryption on blocks ranging from 4 KiB to 16 MiB. For the modules placed on the read path of the I/O request, we conduct the decryption and the decompression on blocks ranging from 4 KiB to 64 MiB.

Our observations represent an evolution towards larger logical pages compared to a study of Chen et al. [13] from 2010, who observe that “a commercial DBMS” accesses almost exclusively logical pages of size 8 KiB for both reads and writes, or MySQL’s default page size of 16 KiB [57]. The same direction towards larger logical pages is also reported by Umbra [58], with the smallest page size being 64 KiB, by Snowflake [71], with 100 MB to 250 MB page sizes, or by Vertica, with Hadoop FS Block Size being set to 64 MB. Antonopoulos et al. [5] report initial log sizes between 3 GB-38 GB as they attempt to reach constant time recovery in Azure SQL Databases by truncating the initial log sizes to 200 MB.

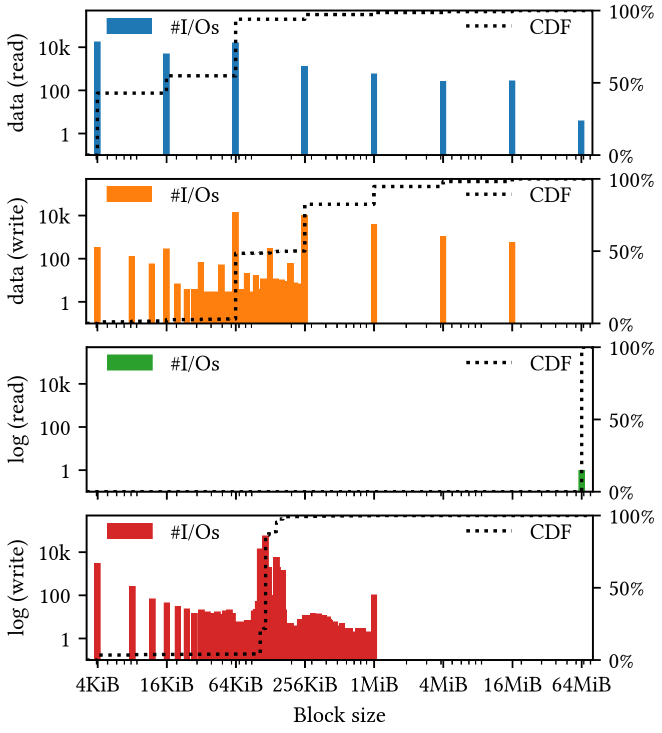


Figure 6: Cumulative distribution function (CDF) and histogram of I/O sizes in I/O trace.

4.1 Real-world I/O Trace

SAP HANA supports a wide variety of workloads. We focus on a concrete use case that generates significant I/O traffic and which is especially common in the cloud: loading data into the system. During this and other write-intensive workloads, SAP HANA initially applies updates to the write-optimized store of each table residing in memory. For durability, changes are also persisted in the transaction log on storage. Depending on the configuration of the system, the write-optimized store is merged into the read-optimized store in order to maintain high read performance. When this process completes, the new read-optimized store (i.e., the snapshot of the table at the point of the merge) is persisted on disk and outdated snapshots are deleted [4]. The snapshots and the log entries caused by data import, which is typically committed in batches of a few thousand rows, result in a regular stream of relatively large writes to storage, and are thus potentially well-suited for block-based compression.

The payload of our compression and encryption modules concerns data from (i) the transaction log, and (ii) from current snapshots of the read-optimized partition of tables. Since such snapshots are only read at column granularity, block-based schemes are suitable for this use case.

To understand the granularity of the I/O transfers in a real system, we analyzed an I/O trace using SAP HANA 2.0 SPS 04 Database Revision 045. This version has a similar I/O behavior as the newer SAP HANA releases, e.g., SAP HANA 2.0 SPS 06 Database Revision 061. SAP HANA uses Linux asynchronous I/O subsystem for logs and data files, so we intercept calls to that subsystem (`io_submit` and `io_getevents`) by attaching a small tracing library using `LD_PRELOAD` and trace the content and metadata of all I/O blocks read and written by the database.

We use a representative workload that captures common operations in cloud deployments to generate the trace. We import the CSV files from the TPC-H data generator for Scale Factor 10, which take about 5.5 GB on disk. For each table in the schema, we run a sequence of five operations: (i) import the file from CSV; (ii) unload the table from main memory; (iii) load the table again; (iv) run `COUNT(*)`; and (v) finally unload again the table. Such a workload emphasizes data loading, a critical operation in replication tasks, e.g., regularly replicating data from an Enterprise Resource Planning (ERP) system into an analytical warehouse. SAP BWH benchmark[5], as well as the TPC-H and TPC-DI benchmarks, report this type of workload as being a crucial element in performance. In the cloud, it captures the constant ETL and loading of data into an analytical system from transactional engines and other sources. Such a workload stresses the I/O subsystem with writes to the write-ahead log, and reads from and writes to data files when tables are loaded into main memory. The same pattern is observed when the read-optimized storage is snapshot to storage after being merged with the newly loaded records from the write-optimized store.

As part of the workload, which includes the startup of the system, SAP HANA reads and writes a total of 7.0 GB and 21.7 GB from and to the storage. Most of these reads are caused by loading the tables in step (iii), while most of the writes are caused by the repeating merges during the import in step (i). At the same time, the system reads and writes a total of 64.0 MiB and 10.1 GB from and to the log, respectively. The reading happens in a single access directly after start-up. Presumably, the system batch-loads the tail of the log and simply determines that the previous shutdown was clean. Most of the data written to the log is caused by the import in step (i).

Figure 6 shows the distribution of the block sizes and Table 2 summarizes the percentage occurrence of relevant block sizes for each page access type. The plot shows that the system accesses most data in blocks larger than the traditional 4 KiB page size. On the data file, most accesses are 64 KiB and some up to 64 MiB. This is not surprising since the snapshots stored in the data file consist of large column vectors and dictionaries that are always accessed in their entirety (namely when flushed after a merge or when loaded back to main memory). The reads from the data file always access blocks whose size is an even power of two and the writes mostly do the same. However, interestingly, the system occasionally also writes blocks with a non-power-of-two size (but still multiples of 4 KiB). The workload only contains a single read from the log (64 MiB at system start-up) while writes occur with a large number of different block sizes. A few of the write block sizes are much more frequent than others. Most notably, about 60 % of all writes are of 118 784 B, precisely 116 KiB (i.e., 29 physical disk pages).

4.2 Evaluation Setup

Software. For the software baseline we use an Intel® Xeon® Gold 6234 Processor 3.3 GHz machine with 8 cores and 16 threads featuring: 512 kB (L1 cache), 8 MB (L2 cache), and 24.75 MB (L3 cache). The level of parallelism set for our compression/decompression and encryption/decryption baselines is consistent with the number of threads SAP HANA allocates for these background tasks, namely 1-2 threads. These threads process the blocks in their entirety.

[5]https://www.sap.com/dmc/exp/2018-benchmark-directory/#/bwh

Table 2: Page type accesses [%].

Block size [B]	4 Ki	16 Ki	64 Ki	116 Ki	256 Ki	1 Mi	4 Mi	16 Mi	64 Mi
data read	42.87	12.01	39.13	-	3.24	1.41	0.64	0.70	0.01
data write	1.08	0.92	45.38	-	32.33	12.28	3.44	1.82	-
log read	-	-	-		-	-	-	-	100
log write	3.46	0.05	0.01	64.64	0.01	0.11	-	-	-

Hardware. Our target platform consists of the Intel Programmable Accelerator Card (PAC) for data centers, Intel FPGA PAC D5005 [31], connected to the CPU via a PCIe Gen3x16 link. The card features a Stratix 10 SX FPGA, two QSFP+ connectors with up to 100 Gbps support, and 32 GB of on-board DDR4-2400 memory, with a peak transfer rate of 19.2 GB/s. We use OpenCL for Intel FPGA SDK (OpenCL RTE version 19.2.0.57) [32] to implement and instantiate the FPGA compute kernels that are interfacing with the on-board DDR4 memory at 64 B cacheline granularity for both read and write operations. The CPU (host processor) allocates memory for the FPGA computing kernels, and a memory management library handles the address translation between CPU main memory and FPGA external memory.

Figure 5 illustrates the OpenCL FPGA framework diagram and differentiates between three memory types in the OpenCL memory model: (i) the device global memory, FPGA external DDR4 memory; (ii) the local memory, FPGA on-chip memory like Block RAM (BRAM); and (iii) the private memory, FPGA on-chip registers.

Global memory represents a major component of the OpenCL compute model, being used to transfer data between the host processor and the FPGA via input/output buffers mapped inside the global memory. The host writes the data to be processed into the input buffer from where the computing kernel reads it, processes it, and writes the obtained results into the output buffer. The results are then transferred back by a memory access generated by the host. This OpenCL computation model is supported by the Board Support Package (BSP) for our Intel FPGA PAC D5005 board, and we use it to evaluate our standalone and pipelined kernels.

For our performance evaluation, we assume data to be already in the global memory, and the computing kernel reads it from there at cacheline granularity (64 B). At this granularity, the global memory latency is higher than the computation time, which affects the performance measured for small block sizes (e.g., 4 KiB). To partly mask this overhead, we use prefetching inside the compute kernel. Note that, in the cloud, the CPU needs to move the data after compression and encryption to a NIC, thereby also paying a data transfer overhead. Our design is intended to be deployed on an FPGA with direct network access, so no further data transfer overhead occurs. Thus, we focus on the performance inside the FPGA compared to that of the CPU, since transfer overheads will be either similar in both cases or much less on the FPGA by skipping a transfer to the NIC. Also note that the FPGA board we use has conventional DDR memory. Many FPGA boards are starting to include High Bandwidth Memory (HBM), which would completely

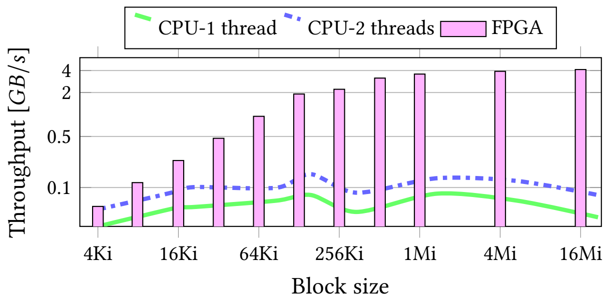


Figure 7: Compression - with 1 and 2 threads on CPU vs. FPGA design. Note the logarithmic scale of the y axis.

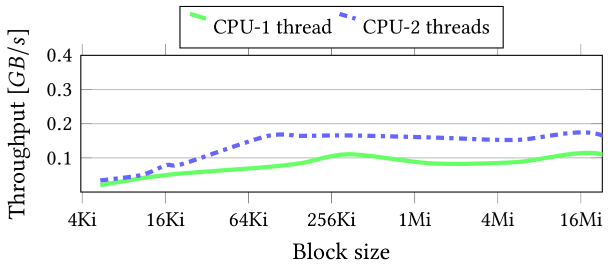


Figure 8: Decompression on CPU.

eliminate this overhead for small data transfers and, in fact, make the overall design faster for all block sizes.

4.3 Compression/Decompression

For the compression and decompression software baselines, we enhance the *rawdeflater* and *rawinflater* methods of the *zlibcomplete* library (version 1.0.5) so that they can independently compress a sequence of different I/O requests (i.e., different blocks are not compressed together). In addition to this, for each block that is processed, we prepend an informative header (64 B) that stores the compressed size of the original block. This header informs the rawinflater method about the exact amount of data it needs to process. Later on, the same header can be used to store information about the type of encryption used after compression. For our evaluation, we use a sliding window of 32 kB with level 1 compression, and we obtain the same range of compression ratios as for the hardware implementation, both using the traces extracted from SAP HANA.

Figure 7 presents the performance comparison of both software baselines and our standalone compression kernel on the FPGA. The algorithms behind LZ77 compression and Huffmann encoding are memory-intensive. First, the search operation of LZ77 compression imposes a variable number of cycles and comparisons to each operand depending on the input data. Second, the Huffman encoding requires handling of the Huffman tree, a data structure that is 1 KiB in our experiments. Hence, for small block sizes, the data overhead is not negligible. Luckily, these two steps can be parallelized and put on a stream dataflow path, minimizing pipeline stalls and global memory accesses.

Compared to the CPU baselines, the FPGA achieves over an order of magnitude speed-up for block sizes larger than 64 KiB. While the

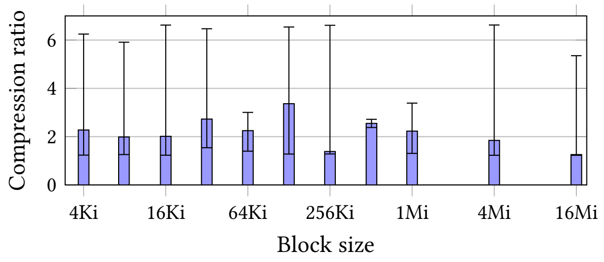


Figure 9: Compression ratios on real-world database traces determined as the ratio of uncompressed size to compressed size; the higher the better.

overhead costs for small block sizes for the FPGA come from both the memory movement overhead and the Huffman tree; on the CPU, the bottleneck of the cache size cannot be avoided. Therefore, the throughput gain obtained on the FPGA by increasing the block size is not seen on the CPU, where the fastest implementation saturates at less than 0.2 GB/s.

Altogether, Figure 7 and Figure 8 show that both compression and decompression are computationally expensive when carried out on the CPU, as similarly identified by several works in literature [1, 63, 65] and in industry [50, 59, 62].

The compression ratio of one block is computed as the ratio between the input data size (uncompressed, or light-weight compressed as it is for SAP HANA) and its compressed size (e.g., a block of 4 KiB yields a compression ratio of 2.4 for a 1.67 KiB compressed file). In Figure 9, we present the compression ratio range that is obtained for different block sizes when compressing the traces in Section 4.1. Even if the compression ratio range varies broadly from less than 1 to more than 6, its average is around 2. If the compression ratio is less than 1, it means that the compressed block size is larger than the original file size. We have rarely observed such cases.

It is important to notice that, for SAP HANA, the input is usually already compressed using light-weight compression, as described in Section 2.1. This means that the benefits obtained from compression can be even larger for a system which does not perform any light-weight compression on the raw data.

4.4 Encryption/Decryption

As a baseline, we build a library on top of Intel AES intrinsic instruction set for the three AES-256 modes (ECB, CTR, CBC) [28]. Each block cipher mode receives for encryption/decryption the same block sizes as the ones traced in SAP HANA (Section 4.1).

Our results yield performance for a peak clock frequency of 3.3 GHz consistent with the results reported by Intel [28] for single threaded implementations: 1.76 cycles/byte for ECB encryption/decryption; 1.88 cycles/byte for CTR encryption/decryption; 1.78 cycles/byte for CBC decryption, and 5.65 cycles/byte for CBC encryption. Both AES-256 software operations, encryption (Figure 10) and decryption (Figure 11) are compute-bound, with the individual performance of each mode scaling proportionally with the doubling of allocated threads.

The difference in the software performance of the three modes comes from their compute complexities. ECB mode is the simplest

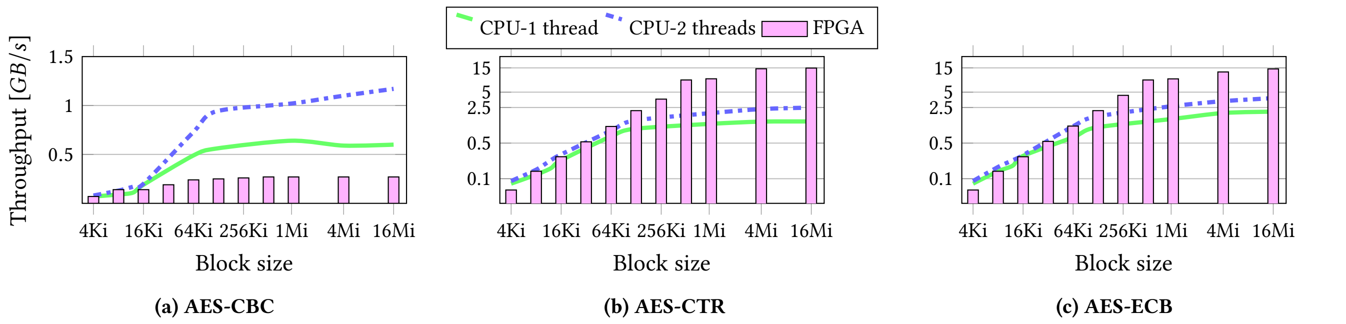


Figure 10: AES Encryption - with 1 and 2 threads on CPU vs. FPGA design. Note the logarithmic scale of the y axis for (b) & (c).

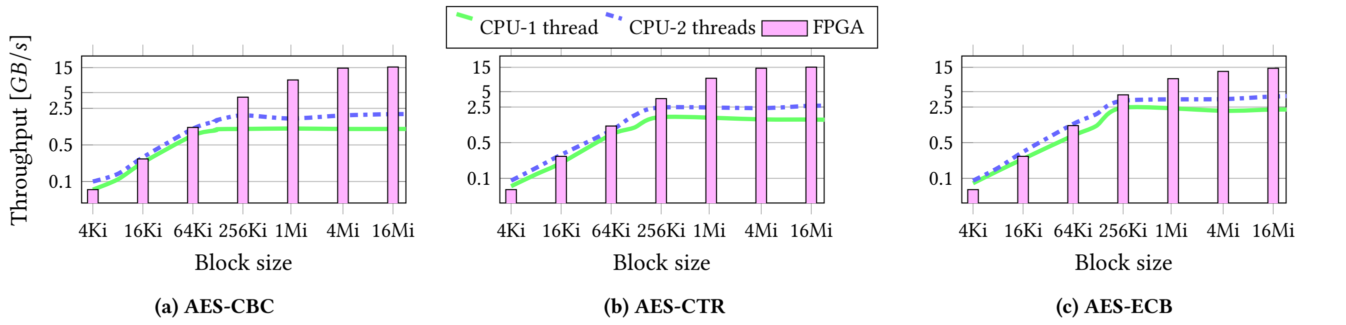


Figure 11: AES Decryption - with 1 and 2 threads on CPU vs. FPGA design. Note the logarithmic scale of the y axis.

of them with only 14 transformation rounds on the datapath, resulting in the best software performance of a maximum of 4 GB/s for both encryption and decryption. CTR mode adds-in complexity by requiring a 128-bit XOR operation on the datapath after the transformation rounds are performed. Its software performance gets penalized, reaching a maximum throughput of 2.5 GB/s. Figure 11a shows that the software performance of decryption in CBC mode is similar to the one in CTR mode due to the required 128-bit XOR operation. Encryption in CBC mode requires each 128-bit input data word to be XOR-ed with the previously obtained 128-bit output data word (the first one is XOR-ed with an initialization vector), before entering the transformation rounds. The cost of the data dependency adds to the cost of the XOR operation, leading to a maximum throughput of 1.2 GB/s.

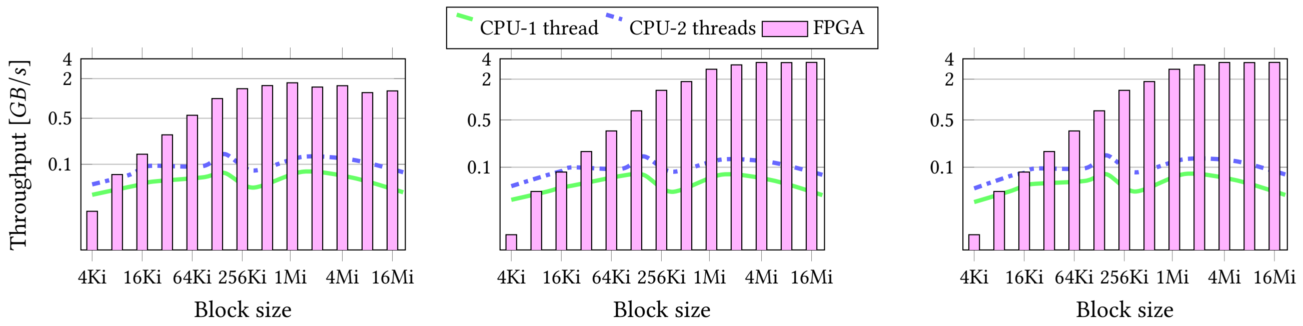
In Figure 10a, we show the limitations of the MHz operational clock range of the FPGA. Even if for the FPGA the XOR operation comes at no performance cost, the data dependency translated into the sequential nature of the CBC mode implementation limits the FPGA CBC encryption throughput performance to 0.27 GB/s. At block size granularity, CBC encryption cannot take advantage of the parallelization potential of the FPGA, whereas the CTR and ECB modes benefit from it, reaching a maximum throughput performance of 15 GB/s. By exploiting the spatial parallelism available on the FPGA, CTR and ECB encryption modes exceed by up to seven times their corresponding CPU performance.

Irrespective of the mode used for encryption, the transfer latency from the global memory is visible for small block sizes (up to 16 KiB). Figure 10 (a-c) shows that this overhead has a larger impact on performance than the algorithm's complexity itself, making all three FPGA encryption modes perform similarly when compared to their corresponding CPU implementations.

Figure 11 (a-c) illustrates the parallelization potential of the FPGA since decryption is parallelizable for all three modes. While the CPU performance saturates at around 2.5 GB/s for the CBC and CTR modes and at around 4.5 GB/s for the ECB mode, the FPGA implementation can reach up to 15 GB/s, irrespective of the mode employed for decryption. The transfer latency from the global memory for small block sizes (up to 16 KiB) impacts decryption throughput performance as it did for encryption. For small block sizes, the CPU and the FPGA have comparable performance.

4.5 Compression and Encryption

There are several points to pay attention when porting both compression and encryption modules together into a single pipeline. Notably, as observed in Figure 7 and Figure 10, the modules have a very different maximum throughput. The choice of which AES-256 block cipher mode to use plays a big role in determining the overall throughput performance. Compared to the 4 GB/s saturation throughput delivered by the compression module, the CBC mode (0.27 GB/s) would impose back-pressure and limit the overall performance, whereas both CTR and ECB modes (15 GB/s) would turn compression into the bottleneck. The block diagram of the full pipeline consisting of compression and encryption working together has been firstly introduced in Figure 4. In Figure 12 we analyze its throughput performance. The pipeline with encryption in CBC mode reaches a maximum throughput of 1.72 GB/s,



(a) Compression + AES-CBC **(b) Compression + AES-CTR** **(c) Compression + AES-ECB**

Figure 12: Full pipeline - with 1 and 2 threads on CPU vs. FPGA design. Note the logarithmic scale of the y axis.

while for encryption in CTR or ECB mode, we achieve a comparable throughput as imposed by the compression module alone, namely 4 GB/s.

For the software baseline, we extend compression with encryption functionality for the three encryption block cipher modes. Each compressed block is encrypted using Intel's AES intrinsic instruction set and placed at a contiguous memory location from where it can be read. The performance of the software pipeline is limited by compression, and even if its performance is better or marginally comparable to the FPGA (for block sizes of 4 KiB and up to 16 KiB, respectively), the full pipeline is accelerated on the FPGA for block sizes bigger than 16 KiB in all three cases, as shown in Figure 12.

When considering the performance of the combined design, it is important to understand the effects of compressing the data before encrypting it. The compression module reads (i.e., consumes) an amount X of data and produces an amount Y of data, where $X \gg Y$. The write throughput, i.e., the rate at which the compression module produces data, is therefore, by definition, equal or smaller than the read throughput. Additionally, the better the compression ratio, the smaller the write throughput. In other words, while our goal is to maximize the read throughput and also aim a high compression efficiency, the write throughput (as a measure, not as a capacity) decreases. While this observation does not play a role when evaluating the compression module in isolation, it becomes a key factor when the compression module is combined with the encryption one. Notably, the maximum read throughput observed for AES-256 in CBC mode (Figure 10a) does not match the maximum read throughput when the module is placed *after* compression (Figure 12a). Since AES-256 CBC encryption works on compressed data, the overall throughput of the combined design is higher than that of the AES-256 module in CBC mode alone.

4.6 Batch Processing

For our use case, an analytical workload using asynchronous I/O, it is natural to consider the possibility of batching I/O requests to process larger data sizes. Most database engines do this already in one way or another. When using an accelerator, batching has the advantage of reducing the number of calls (which add overhead) to the FPGA, in addition to the advantages related to better network and storage utilization. In the experiments so far, we see a large performance gap depending on the amount of data processed in each request (notably, 0.27 GB/s and 15 GB/s in Figure 12). Therefore, the question is whether we can improve performance for smaller data sizes by batching them into a larger transfer sizes where the accelerator has a clear advantage. In order to find the answer, we perform an experiment where small blocks of a given size are combined into larger batches. We repeat the experiment for different sizes of the small blocks and for different batch sizes (64 KiB, 128 KiB and 256 KiB). As a baseline for this experiment, we compare the results of batching to the ones obtained for a single block call (Figure 12). Since the performance of combined compression and encryption module for the three AES modes is very similar for block sizes up to 256 KiB, we report the numbers of CBC mode only.

Figure 13 plots the throughput for each one of these experiments, grouped by batch sizes (i.e., a 64 KiB batch is composed of 16 blocks of 4 KiB, or 8 blocks of 8 KiB, etc.). The two key takeaways from the figure are that (i) batching does not improve performance significantly, and (ii) the reason is the way compression works. While batching improves the throughput to a certain extent, it is not a significant improvement and it actually requires to batch a lot of requests to get some gains. The explanation for this behavior is that, for each block, the compression module has to create the dictionary, search the matching bits, and erase or modify the data. In the case

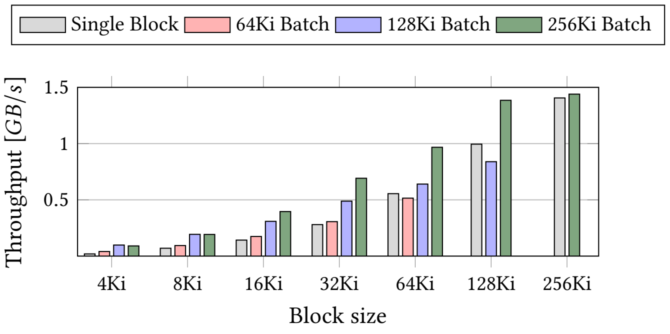


Figure 13: Full pipeline throughput, Compression + AES-CBC, processing multiple blocks from the same request. Batch sizes of 64 KiB, 128 KiB and 256 KiB for blocks from 4 KiB to 256 KiB.

of batch processing of small blocks, this process has to be repeated several times since each block could come from different tables and cannot be compressed together. As a result, the compression overhead dominates even when batching. In some extreme cases of batching, e.g., when batching a very small block with a large block, the throughput for the small block might improve but we do not consider such situations common enough to enforce more sophisticated batching beyond what the database engine does by itself.

5 DISCUSSION

Based on the results in Section 4, we can now put into perspective the potential of using an accelerator on the data path of SAP HANA. We focus on the intrinsic overheads of the explored methods since some of the hardware limitations will disappear over time (e.g., FPGAs with HBMs and with larger capacities for parallel modules).

5.1 General Discussion

The results indicate that, for larger data transfers, offloading compression and encryption to an FPGA instead of performing them on the CPU has significant advantages. On the one hand, the process is significantly faster on the FPGA, which makes the option of both compressing and encrypting data on the I/O path for cloud deployments feasible. This has many advantages: data is secured while at rest and in transit, the encryption keys remain in control of the database engine and not of the cloud provider (i.e., it is the database the one compressing and encrypting, not the storage layer), the space needed on storage is reduced and also makes the network, a scarce resource in the cloud, more efficient. On the other hand, doing so frees up valuable CPU cycles that can be either used for other purposes or not used at all, with the engine requiring less virtual CPUs to run on the cloud for a similar performance.

The results also show that while there are gains when compressing and encrypting, the potential gains when encrypting alone would be much bigger. Encryption for large block sizes operates at several GB/s, a rate that matches or surpasses the available network bandwidth in conventional cloud deployments and, thus, making the encryption free in terms of I/O latency. From a business perspective, several commercial strategies could be envisaged. First, for users prioritizing storage price over performance, the engine could compress and encrypt to reduce storage cost. Second, for high-end users who prioritize performance, encryption can be provided without compression, thus maximizing throughput and freeing up CPU resources for query processing. Third, the engine can accommodate different security levels. Finally, with compression placed before encryption, the security gap between different AES encryption block cipher modes is minimized since compression aims to eliminate redundancies within a given data block.

Introducing an accelerator like the one proposed on the I/O path also opens up several interesting possibilities for cloud database engines. Databases are full of design decisions driven by the assumption of a slow I/O path and the overhead associated with compressing and encrypting the data. Such design decisions would change with an accelerator. In fact, cloud native databases already use block sizes that are typically larger than those of conventional databases [17]. The accelerator would benefit from these larger transfer sizes and motivate such change even further.

5.2 Discussion on I/O Transfer Sizes

The experimental results indicate that offloading compression and encryption to an accelerator becomes attractive for block sizes larger than 64 KiB. Compression and encryption are relevant when writing data or the log to storage. As the traces show, this size range corresponds to more than 95 % of the I/O requests observed in the traces (observe the jump in the CDF in Figure 6 at 64 KiB for data and at slightly larger sizes for the log). The same observation can be made for decryption: there are many requests of sizes large enough for the accelerator to be a better choice overall. From this analysis we conclude that the disadvantage of processing small block sizes on the FPGA is mitigated by the small frequency of such requests. Even if for some small sizes the accelerator is slightly slower, for the entire workload it brings a clear advantage. This compromise across a workload is very common in database engines and the data-write path of SAP HANA can afford such a compromise, as is probably the case for most analytical databases.

Note that the analysis of the traces also seem to indicate that, if the data is going to be compressed before being written to cloud storage, it is better to do this on the FPGA than on the CPU, even if compression creates a bottleneck with respect to encryption on the FPGA. Compression on the FPGA wins for block sizes larger than 32 KiB, a range representing over 96 % of all write requests.

6 CONCLUSION

In this paper we have explored the implementation of compression and encryption on the I/O path of a relational, analytics engine (SAP HANA). These two operations play a key role in cloud deployments: in the case of compression, to reduce the cost of storage as well as to maximize the available network bandwidth; in the case of encryption, to protect the data both while at rest and while it is being transmitted over the network. Our results show that offloading these operations to an FPGA accelerator offer significant advantages both in terms of performance as well as freeing up valuable CPU resources in cloud settings as long as there is enough data being transferred. As we demonstrate with SAP HANA as an example, in analytical engines the data transfer sizes are large enough for the accelerator to offer a clear advantage. Conversely, the approach we have explored does not seem to be suitable for transactional engines, as the amount of data involved in every transfer tends to be too small and the synchronous I/O precludes batching. Future work includes exploring other forms of compression and decompression that provide a better trade-off performance vs. compression ratio, integration of the proposed approach with a SmartNIC to perform these operations on the fly as data is sent to storage, and considering data formats other than relational as they are increasingly being used in modern engines and pose challenges different from those of relational data.

ACKNOWLEDGMENTS

We thank Intel for loaning the PAC D5005 FPGA board used for this work. We also deeply thank Suleyman Demirsoy and Christian Färber (Intel) for the technical support and Thorsten Glebe (SAP) for the technical input. The work of Fabio Maschi and Monica Chiosa has been funded in part through a donation from SAP.

REFERENCES

[1] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. 2014. Gzip on a chip: high performance lossless data compression on FPGAs using OpenCL. In *Proceedings of the International Workshop on OpenCL, IWOCL 2013 & 2014, May 13-14, 2013, Georgia Tech, Atlanta, GA, USA / Bristol, UK, May 12-13, 2014*, Simon McIntosh-Smith and Ben Bergen (Eds.). ACM, 4:1–4:9. https://doi.org/10.1145/2664666.2664670

[2] Mohammed Abouzour, Günes Aluç, Ivan T. Bowman, Xi Deng, Nandan Marathe, Sagar Ranadive, Muhammed Sharique, and John C. Smirnios. 2021. Bringing Cloud-Native Storage to SAP IQ. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2410–2422. https://doi.org/10.1145/3448016.3457563

[3] Nabihah Ahmad and S. M. Rezaul Hasan. 2021. A new ASIC implementation of an advanced encryption standard (AES) crypto-hardware accelerator. *Microelectron. J.* 117 (2021), 105255. https://doi.org/10.1016/j.mejo.2021.105255

[4] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. 2017. SAP HANA Adoption of Non-Volatile Memory. *Proc. VLDB Endow.* 10, 12 (2017), 1754–1765. https://doi.org/10.14778/3137765.3137780

[5] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant Time Recovery in Azure SQL Database. *Proc. VLDB Endow.* 12, 12 (2019), 2143–2154. https://doi.org/10.14778/3352063.3352131

[6] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure database-as-a-service with Cipherbase. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 1033–1036. https://doi.org/10.1145/2463676.2467797

[7] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security with Cipherbase. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper33.pdf

[8] Kubilay Atasu, Luca Breveglieri, and Marco Macchetti. 2004. Efficient AES implementations for ARM based platforms. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus, March 14-17, 2004*, Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock (Eds.). ACM, 841–845. https://doi.org/10.1145/967900.968073

[9] AWSCloud. 2020. *AQUA (Advanced Query Accelerator) for Amazon Redshift*. Amazon Web Services. Retrieved June 3, 2022 from https://aws.amazon.com/redshift/features/aqua/

[10] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54. https://doi.org/10.1145/3015146

[11] Michaela Blott, Kimon Karras, Ling Liu, Kees A. Vissers, Jeremia Bär, and Zsolt István. 2013. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *5th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'13, San Jose, CA, USA, June 25-26, 2013*, Dilma Da Silva and George Porter (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/blott

[12] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 11 (2020), 2649–2661. http://www.vldb.org/pvldb/vol13/p2649-boncz.pdf

[13] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2010. TPC-E vs. TPC-C: characterizing the new TPC-E benchmark via an I/O comparison study. *SIGMOD Rec.* 39, 3 (2010), 5–10. https://doi.org/10.1145/1942776.1942778

[14] Shuang Chen, Wei Hu, and Zhenhao Li. 2019. High Performance Data Encryption with AES Implementation on FPGA. In *5th IEEE International Conference on Big Data Security on Cloud, IEEE International Conference on High Performance and Smart Computing, and IEEE International Conference on Intelligent Data and Security, BigDataSecurity/HPSC/IDS 2019, Washington, DC, USA, May 27-29, 2019*. IEEE, 149–153. https://doi.org/10.1109/BigDataSecurity-HPSC-IDS.2019.00036

[15] Pawel Chodowiec and Kris Gaj. 2003. Very Compact FPGA Implementation of the AES Algorithm. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings (Lecture Notes in Computer Science)*, Colin D. Walter, Çetin Kaya Koç, and Christof Paar (Eds.), Vol. 2779. Springer, 319–333. https://doi.org/10.1007/978-3-540-45238-6_26

[16] Joan Daemen and Vincent Rijmen. 1999. *AES proposal: Rijndael.*

[17] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 215–226. https://doi.org/10.1145/2882903.2903741

[18] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. 2021. Transformations of High-Level Synthesis Codes for High-Performance Computing. *IEEE Trans. Parallel Distributed Syst.* 32, 5 (2021), 1014–1029. https://doi.org/10.1109/TPDS.2020.3039409

[19] Peter Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. , 17 pages. https://doi.org/10.17487/RFC1951

[20] Mohamed A. Abd El-Ghany, Aly E. Salama, and Ahmed H. Khalil. 2007. Design and Implementation of FPGA-based Systolic Array for LZ Data Compression. In *International Symposium on Circuits and Systems (ISCAS 2007), 27-20 May 2007, New Orleans, Louisiana, USA*. IEEE, 3691–3695. https://doi.org/10.1109/ISCAS.2007.378644

[21] Jian Fang, Jianyu Chen, Jinho Lee, Zaid Al-Ars, and H. Peter Hofstee. 2020. An Efficient High-Throughput LZ77-Based Decompressor in Reconfigurable Logic. *J. Signal Process. Syst.* 92, 9 (2020), 931–947. https://doi.org/10.1007/s11265-020-01547-w

[22] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33. http://sites.computer.org/debull/A12mar/hana.pdf

[23] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, Sujata Banerjee and Srinivasan Seshan (Eds.). USENIX Association, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[24] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. 2015. A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs. In *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*. IEEE Computer Society, 52–59. https://doi.org/10.1109/FCCM.2015.46

[25] Panu Hämäläinen, Timo Alho, Marko Hännikäinen, and Timo D. Hämäläinen. 2006. Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core. In *Ninth Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD 2006), 30 August - 1 September 2006, Dubrovnik, Croatia*. IEEE Computer Society, 577–583. https://doi.org/10.1109/DSD.2006.40

[26] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 651–665. https://doi.org/10.1145/3299869.3314041

[27] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[28] Intel. 2010. Intel Advanced Encryption Standard (AES) New Instructions Set. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[29] Intel. 2020. Intel® Stratix® 10 Logic Array Blocks and Adaptive Logic Modules User Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-lab.pdf

[30] Intel. 2021. FPGA SDK for OpenCL Pro Edition, Programming Guide 21.3. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[31] Intel. 2021. *Intel FPGA Programmable Acceleration Card D5005*. Intel. Retrieved June 3, 2022 from https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/intel-fpga-pac-d5005/specifications.html

[32] Intel. 2021. Intel FPGA SDK for OpenCL Pro Edition: Getting Started Guide. https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807309901.html

[33] Intel. 2021. Intel® FPGA SDK for OpenCL™ Pro Edition: Programming Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[34] Intel. 2021. Intel® QuickAssist Technology. https://01.org/sites/default/files/downloads//330684-011-intel-qat-api-programmers-guide.pdf

[35] Intel. 2021. Memory Optimization for OpenCL on Intel FPGAs. https://www.intel.com/content/www/us/en/programmable/support/training/course/oopnclmemopt.html

[36] Zsolt István. 2020. Let's add transactions to FPGA-based key-value stores!. In *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, Danica Porobic and Thomas Neumann (Eds.). ACM, 13:1–13:3. https://doi.org/10.1145/3399666.3399909

[37] Zsolt István, Kaan Kara, and David Sidler. 2020. FPGA-Accelerated Analytics: From Single Nodes to Clusters. *Found. Trends Databases* 9, 2 (2020), 101–208. https://doi.org/10.1561/1900000072

[38] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent Distributed Storage. *Proc. VLDB Endow.* 10, 11 (2017), 1202–1213. https://doi.org/10.14778/3137628.3137632

[39] Kimmo U. Järvinen, Matti Tommiska, and Jorma Skyttä. 2003. A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 2003, Monterey, CA, USA, February 23-25, 2003*, Steve Trimberger and Russell Tessier (Eds.). ACM, 207–215. https://doi.org/10.1145/611817.611848

[40] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: A High-Performance Database System Leveraging In-Storage Computing. *Proc. VLDB Endow.* 9, 12 (2016), 924–935. https://doi.org/10.14778/2994509.2994512

[41] Kaan Kara, Ken Eguro, Ce Zhang, and Gustavo Alonso. 2018. ColumnML: Column-Store Machine Learning with On-The-Fly Data Transformation. *Proc. VLDB Endow.* 12, 4 (2018), 348–361. https://doi.org/10.14778/3297753.3297756

[42] Youngil Kim, Seungdo Choi, Joonyong Jeong, and Yong Ho Song. 2019. Data dependency reduction for high-performance FPGA implementation of DEFLATE compression algorithm. *J. Syst. Archit.* 98 (2019), 41–52. https://doi.org/10.1016/j.sysarc.2019.06.005

[43] Robert Lasch, Süleyman Sirri Demirsoy, Norman May, Veeraraghavan Ramamurthy, Christian Färber, and Kai-Uwe Sattler. 2020. Accelerating re-pair compression using FPGAs. In *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, Danica Porobic and Thomas Neumann (Eds.). ACM, 8:1–8:8. https://doi.org/10.1145/3399666.3399931

[44] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. 2020. High-Throughput FPGA-Based Hardware Accelerators for Deflate Compression and Decompression Using High-Level Synthesis. *IEEE Access* 8 (2020), 62207–62217. https://doi.org/10.1109/ACCESS.2020.2984191

[45] Donghun Lee, Andrew Chang, Minseon Ahn, Jongmin Gim, Jungmin Kim, Jaemin Jung, Kang-Woo Choi, Vincent Pham, Oliver Rebholz, Krishna Malladi, and Yang-Seok Ki. 2020. Optimizing Data Movement with Near-Memory Acceleration of In-memory DBMS. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 371–374. https://doi.org/10.5441/002/edbt.2020.35

[46] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. 2010. Speeding Up Queries in Column Stores - A Case for Compression. In *Data Warehousing and Knowledge Discovery, 12th International Conference, DAWAK 2010, Bilbao, Spain, August/September 2010. Proceedings (Lecture Notes in Computer Science)*, Torben Bach Pedersen, Mukesh K. Mohania, and A Min Tjoa (Eds.), Vol. 6263. Springer, 117–129. https://doi.org/10.1007/978-3-642-15105-7_10

[47] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 447–461. https://doi.org/10.1145/3341301.3359628

[48] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 137–152. https://doi.org/10.1145/3132747.3132756

[49] Máire McLoone and John V. McCanny. 2001. High Performance Single-Chip FPGA Rijndael Algorithm Implementations. In *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings (Lecture Notes in Computer Science)*, Çetin Kaya Koç, David Naccache, and Christof Paar (Eds.), Vol. 2162. Springer, 65–76. https://doi.org/10.1007/3-540-44709-1_7

[50] Microsoft. 2021. *Azure, SQL Server Data Compression*. Microsoft. https://docs.microsoft.com/en-us/sql/relational-databases/data-compression/data-compression?view=sql-server-ver15

[51] Microsoft. 2021. Transparent Data Encryption (TDE). https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption?view=sql-server-ver15

[52] Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, Norman May, and Akash Kumar. 2021. Resource-Efficient Database Query Processing on FPGAs. In *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, Danica Porobic and Spyros Blanas (Eds.). ACM, 4:1–4:8. https://doi.org/10.1145/3465998.3466006

[53] MongoDB. 2019. *Compression, Key Rotation and Configuration*. MongoDB. https://www.mongodb.com/blog/post/coming-in-42-compression-key-rotation-and-configuration

[54] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy (Eds.). OpenProceedings.org, 283–294. https://doi.org/10.5441/002/edbt.2014.27

[55] MySQL. 2020. Encryption and Compression Functionse. https://dev.mysql.com/doc/refman/8.0/en/encryption-functions.html#function_aes-encrypt

[56] MySQL. 2020. *How Compression Works for InnoDB Tables*. MySQL. https://dev.mysql.com/doc/refman/8.0/en/innodb-compression-internals.html

[57] MySQL. 2021. MySQL 8.0 Reference Manual :: 15.11.2 File Space Management. https://dev.mysql.com/doc/refman/8.0/en/innodb-file-space.html

[58] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[59] Oracle. 2018. *Advanced Compression Proof-of-Concept Insights*. Oracle. https://www.oracle.com/a/tech/docs/advanced-compression-poc-insights.pdf

[60] Oracle. 2019. Encryption and Redaction with Oracle Advanced Security. https://www.oracle.com/a/tech/docs/dbsec/aso/advanced-security-wp-19c.pdf

[61] Oracle. 2021. General Considerations of Using Transparent Data Encryption. https://docs.oracle.com/en/database/oracle/oracle-database/18/asoag/general-considerations-of-using-transparent-data-encryption.html#GUID-ED481093-51B0-42F2-BADB-E5E55889AD47

[62] Jian Ouyang, Hong Luo, Zilong Wang, Jiazi Tian, Chenghui Liu, and Kehua Sheng. 2010. FPGA implementation of GZIP compression and decompression for IDC services. In *Proceedings of the International Conference on Field-Programmable Technology, FPT 2010, 8-10 December 2010, Tsinghua University, Beijing, China*, Jinian Bian, Qiang Zhou, Peter Athanas, Yajun Ha, and Kang Zhao (Eds.). IEEE, 265–268. https://doi.org/10.1109/FPT.2010.5681489

[63] Jian Ouyang, Hong Luo, Zilong Wang, Jiazi Tian, Chenghui Liu, and Kehua Sheng. 2010. FPGA implementation of GZIP compression and decompression for IDC services. In *Proceedings of the International Conference on Field-Programmable Technology, FPT 2010, 8-10 December 2010, Tsinghua University, Beijing, China*, Jinian Bian, Qiang Zhou, Peter Athanas, Yajun Ha, and Kang Zhao (Eds.). IEEE, 265–268. https://doi.org/10.1109/FPT.2010.5681489

[64] Marcus Paradies, Christian Lemke, Hasso Plattner, Wolfgang Lehner, Kai-Uwe Sattler, Alexander Zeier, and Jens Krüger. 2010. How to juggle columns: an entropy-based approach for table compression. In *Fourteenth International Database Engineering and Applications Symposium (IDEAS 2010), August 16-18, 2010, Montreal, Quebec, Canada (ACM International Conference Proceeding Series)*, Bipin C. Desai and Jorge Bernardino (Eds.). ACM, 205–215. https://doi.org/10.1145/1866480.1866510

[65] Ovidiu Plugariu, Alexandru Dumitru Gegiu, and Lucian Petrica. 2017. FPGA systolic array GZIP compressor. In *2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. 1–6. https://doi.org/10.1109/ECAI.2017.8166387

[66] PostgresSQL. 2021. *PostgreSQL 14.0 Documentation*. PostgresSQL. https://www.postgresql.org/docs/14/backup-dump.html

[67] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the Latency Gap between NVM and DRAM for Latency-bound Operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*, Thomas Neumann and Ken Salem (Eds.). ACM, 13:1–13:8. https://doi.org/10.1145/3329785.3329917

[68] Burkhard Ringlein, François Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, Marc Reichenbach, and Dietmar Fey. 2021. A Case for Function-as-a-Service with Disaggregated FPGAs. In *14th IEEE International Conference on Cloud Computing, CLOUD 2021, Chicago, IL, USA, September 5-10, 2021*, Claudio Agostino Ardagna, Carl K. Chang, Ernesto Daminai, Rajiv Ranjan, Zhongjie Wang, Robert Ward, Jia Zhang, and Wensheng Zhang (Eds.). IEEE, 333–344. https://doi.org/10.1109/CLOUD53861.2021.00047

[69] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. 2019. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *29th International Conference on Field Programmable Logic and Applications, FPL 2019, Barcelona, Spain, September 8-12, 2019*, Ioannis Sourdis, Christos-Savvas Bouganis, Carlos Álvarez, Leonel Antonio Toledo Díaz, Pedro Valero-Lara, and Xavier Martorell (Eds.). IEEE, 286–292. https://doi.org/10.1109/FPL.2019.00053

[70] SAP. 2013. Optimizing your SAP NetWeaver Cloud application with Gzip compression. https://blogs.sap.com/2013/03/04/optimizing-your-sap-netweaver-cloud-application-with-gzip-compression

[71] Snowflake. 2022. *Continuous Data Loads and File Sizing*. Snowflake. Retrieved June 3, 2022 from https://docs.snowflake.com/en/user-guide/data-load-considerations-prepare.html#continuous-data-loads-i-e-snowpipe-and-file-sizing

[72] Bharat Sukhwani, Bülent Abali, Bernard Brezzo, and Sameh W. Asaad. 2011. High-Throughput, Lossless Data Compresion on FPGAs. In *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2011, Salt Lake City, Utah, USA, 1-3 May 2011*, Paul Chow and Michael J. Wirthlin (Eds.). IEEE Computer Society, 113–116. https://doi.org/10.1109/FCCM.2011.56

[73] Jens Teubner and Louis Woods. 2013. *Data Processing on FPGAs*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00514ED1V01Y201306DTM035

[74] Tobias Vinçon, Arthur Bernhardt, Ilia Petrov, Lukas Weber, and Andreas Koch. 2020. nKV: near-data processing with KV-stores on native computational storage. In *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, Danica Porobic and Thomas Neumann (Eds.). ACM, 10:1–10:11. https://doi.org/10.1145/3399666.3399934

[75] Shawn Wang. 2019. *The difference in five modes in the AES encryption algorithm*. https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/

[76] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading. *Proc. VLDB Endow.* 7, 11 (2014), 963–974. https://doi.org/10.14778/2732967.2732972

[77] Xilinx. 2020. *Advanced Encryption Standard (AES) Engine v1.1*. Xilinx. https://www.xilinx.com/content/dam/xilinx/support/documentation/ip_documentation/aes/v1_1/pg383-aes.pdf

[78] Yue Zha and Jing Li. 2021. When application-specific ISA meets FPGAs: a multi-layer virtualization framework for heterogeneous cloud FPGAs. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 123–134. https://doi.org/10.1145/3445814.3446699

[79] Ning Zheng, Xubin Chen, Jiangpeng Li, Qi Wu, Yang Liu, Yong Peng, Fei Sun, Hao Zhong, and Tong Zhang. 2020. Re-think Data Management Software Design Upon the Arrival of Storage Hardware with Built-in Transparent Compression. In *12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*, Anirudh Badam and Vijay Chidambaram (Eds.). USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/zheng

[80] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

[81] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. 2014. Compression and SSDs: Where and How?. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads, INFLOW '14, Broomfield, CO, USA, October 5, 2014*, Kaoutar El Maghraoui and Gokul B. Kandiraju (Eds.). USENIX Association. https://www.usenix.org/conference/inflow14/workshop-program/presentation/zuck