CStream: Parallel Data Stream Compression on Multicore Edge Devices

Xianzhi Zeng, Shuhao Zhang

Abstract—In the burgeoning realm of Internet of Things (IoT) applications on edge devices, data stream compression has become increasingly pertinent. The integration of added compression overhead and limited hardware resources on these devices calls for a nuanced software-hardware co-design. This paper introduces `CStream`, a pioneering framework crafted for parallelizing stream compression on multicore edge devices. `CStream` grapples with the distinct challenges of delivering a high compression ratio, high throughput, low latency, and low energy consumption. Notably, `CStream` distinguishes itself by accommodating an array of stream compression algorithms, a variety of hardware architectures and configurations, and an innovative set of parallelization strategies, some of which are proposed herein for the first time. Our evaluation showcases the efficacy of a thoughtful co-design involving a lossy compression algorithm, asymmetric multicore processors, and our novel, hardware-conscious parallelization strategies. This approach achieves a 2.8x compression ratio with only marginal information loss, 4.3x throughput, 65% latency reduction and 89% energy consumption reduction, compared to designs lacking such strategic integration.

1 Introduction

With the rise of Internet of Things (IoT) applications, the need for efficient data processing in edge devices, especially data stream compression, has become a pivotal research problem [1], [2]. Figure 1 illustrates an IoT use case [3] wherein stream compression in multicore edge devices is highly desirable. Real-time data streams (e.g., toxic gas, temperature) from a multitude of IoT sensors in hazardous areas are incessantly gathered by patrol drones, functioning as edge devices, with limited memory and battery power. To reduce transmission overhead, these patrol drones, equipped with multicore processors, act as multicore edge devices that compress the input streams [4] before passing them to downstream online IoT analytic tasks, such as online aggregation [2], and online machine learning [5] in the cloud.

Parallelizing stream compression on multicore edge devices, such as the wireless patrol drones in Figure 1, is mandatory to meet the strict high-throughput processing requirements. However, achieving this in the resource-constrained environment of multicore edge devices is a non-trivial task. It involves striking a delicate balance between often conflicting requirements such as low energy consumption [6], high compression ratio [5], and tolerable information loss [1]. While data stream compression is a well-studied problem, the specific context of multicore edge devices adds a new dimension to it. In particular, none provide a comprehensive answer to our central question: *how can stream compression be optimally implemented on multicore edge devices?*

Parallelizing stream compression on multicore edge devices, such as the wireless patrol drones in Figure 1, is mandatory to meet the strict high-throughput and low-latency processing requirements. However, achieving this

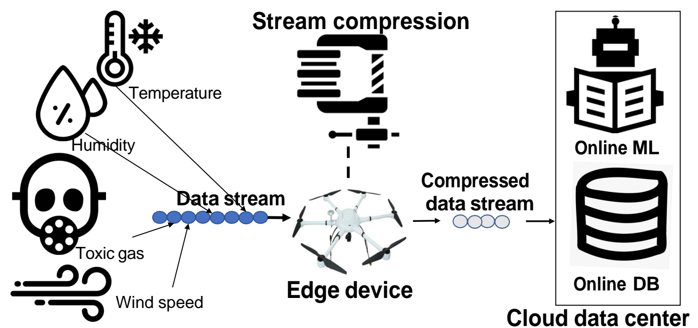


Fig. 1: Application of stream compression in real-time data gathering conducted by a patrol drone in environments inaccessible to humans.

in the resource-constrained environment of multicore edge devices is a non-trivial task. It involves striking a delicate balance between often conflicting requirements such as low energy consumption [6], high compression ratio [5], and tolerable information loss [1]. While data stream compression is a well-studied problem, the specific context of multicore edge devices adds a new dimension to it. In particular, none provide a comprehensive answer to our central question: *how can stream compression be optimally implemented on multicore edge devices?*

This paper introduces `CStream`, an innovative framework specifically engineered for parallelizing stream compression on multicore edge devices. `CStream` systematically navigates the expansive design space, meticulously balancing various factors such as *compression ratios*, *compression speeds*, and *energy consumption*. It offers a flexible, adaptive, and robust solution that pushes the boundaries of stream compression on multicore edge devices, exploiting a versatile software-hardware co-design approach [7], [8].

Firstly, `CStream` supports a wide array of stream

• *Xianzhi Zeng and Shuhao Zhang are with the Singapore University of Technology and Design.*

compression algorithms, each possessing unique strengths and trade-offs. These algorithms span from conventional lossless compression to cutting-edge lossy algorithms, encapsulating both stateful and stateless variations, and those with or without byte alignment.

Secondly, CStream is conscientious of hardware differences, demonstrating adaptability across diverse hardware architectures and configurations. It is capable of optimizing its functionality according to the specificities of various multicore processors, accommodating RISC or CISC architectures, and adjusting to varying word lengths and core numbers. This adaptability allows CStream to optimally harness the hardware resources available on different edge devices, thereby maximizing compression speed while minimizing energy consumption.

Thirdly, CStream introduces a novel set of parallelization strategies, some of which are proposed for the first time. These strategies account for various factors such as execution strategies, state-sharing implementation, and scheduling strategies, offering fine-grained control over the parallelization of stream compression. With these innovative parallelization strategies, CStream is capable of better distributing the compression workload across multiple cores, thereby enhancing throughput and reducing latency.

Our primary contributions are summarized as follows:

- *First*, CStream offers an extensive set of stream compression algorithms, with a particular focus on *lossy stream compression* algorithms. These algorithms strike a balance between high compression ratios (ranging from 2.0 to 8.5) and minimal information loss (less than 5%), enabling CStream to cater to a wide spectrum of application requirements.
- *Second*, CStream is engineered to operate efficiently on asymmetric multicore processors with RISC architecture and 64-bit word length. This results in impressive performance improvements, specifically, up to 59% reduction in processing time and up to 69% reduction in energy consumption when compared to traditional multicore processors.
- *Third*, CStream implements a range of hardware-conscious parallelization strategies. To begin with, it incorporates cache-aware micro-batching of tuples, which significantly improves throughput by up to 11 times. Next, in terms of state management, it opts for private dictionaries for each thread instead of a shared state. This optimization significantly reduces energy consumption and enhances throughput without adversely affecting the compression ratio.
- *Lastly*, CStream adopts asymmetry-aware workload scheduling [4]. This not only reduces energy consumption by about 50% but also optimizes resource utilization by leveraging the unique capabilities of different cores. These strategies, collectively, make CStream an efficient and effective tool for stream compression on heterogeneous multi-core systems.

In demonstrating the efficacy of CStream, we conducted a comprehensive evaluation with five real-world and one synthetic datasets, featuring diverse workload characteristics. Our results confirm CStream's superior performance over traditional approaches, demonstrating its effectiveness in the challenging environment of multicore edge devices (e.g., RK3399 [9], H2+[10], and Z8350[11]). Our observations underscore the value of thoughtful co-design in achieving optimal stream compression on edge devices. We highlight the potential of *lossy stream compression* algorithms, asymmetric multicore processors, and hardware-conscious parallelization strategies. These strategic integrations lead to notable improvements in the compression ratio, throughput, and energy efficiency. With these contributions, we envision CStream to be an indispensable tool for researchers and practitioners aiming to achieve efficient data stream compression for IoT applications.

Organization. The rest of this paper is laid out as follows: Section 2 offers an overview of stream compression and underscores the unique challenges associated with its application at the edge. Section 3 delves into the co-design spaces of stream compression at the edge, underlining the architecture and implementation of CStream. In Section 4, we present the methodology used to evaluate CStream. Section 5 reports on our experimental evaluation of CStream, highlighting its performance across various co-design spaces. Section 6 situates our work within the existing body of literature, emphasizing how CStream advances the current state of the art in stream compression for multicore edge devices. Section 7 concludes the paper, reflecting on the contributions of CStream and offering directions for future research in this domain.

2 Background

In this section, we introduce the basic concepts of stream compression at the edge.

2.1 Overview of Stream Compression

In our definition, a stream tuple, denoted as $x_t = v$, consists of an arrival timestamp t to the processing system (e.g., a compressor), and v, the content to be compressed. We define a list of tuples chronologically arriving at the system as an *input stream*. Stream compression essentially performs the task of *continuously compressing the input stream into an output stream with fewer data footprints*. To describe the relative size between the loaded (i.e., input) and compressed (i.e., output) streaming data, we define *compression ratio* as $compression\ ratio = \frac{loaded\ data\ size}{compressed\ data\ size}$.

Stream compression possesses three major properties stemming from the nature of the continuously arriving data stream. First, the stream is incremental, meaning that compression on the *current tuple* x_τ (i.e., the tuple arriving most recently at time τ) can rely on either 1) itself or 2) the *past tuples* (i.e., those arriving earlier than the current tuple with timestamp $t < \tau$). However, it has no reference to the *future tuples* which haven't arrived yet with timestamp $t > \tau$.

Second, the stream is infinite. Therefore, the proper utilization of cache and memory becomes crucial. Third, the stream is characterized by a large volume and high rate, necessitating high-throughput compression. These distinct properties set stream compression apart from database compression [12], time series compression [13], and file compression [14], where all data is ready before conducting the compression.

2.2 Stream Compression on Edges

The emergence of IoT has led to an explosion in data collection, storage, and processing demand at the edge. Here, we briefly introduce the data properties and performance demand of stream compression on edges.

IoT Data. IoT, being a common application scenario for edge computing, produces diverse data streams at the edge [15], [6]. The source(s) of these data streams could be singular (e.g., an ECG sensor [16]) or multiple (e.g., distributed game servers [17]). Additionally, the tuple content may be a plain value (e.g., unsigned integer [16]), binary structured (e.g., the $< key, value >$ pair [18]), or textual structured (e.g., in XML format [19]). The arrival pattern and compressibility of the data stream can also greatly vary. For instance, an anti-Covid19 tracking system [20] at a mall may generate data streams more densely during peak hours. Due to this versatility, efficient stream compression is highly context-dependent.

Performance Demand. Given the *13 V's challenges [7]* of IoT, stream compression at the edge needs to meet several critical demands:

- *High compression ratio:* Reducing the data footprint in the output stream is a key reason for using stream compression at the edge. With the data generated at the edge being nearly infinite and growing to ZB level per year recently [21], and considering the limited memory capacity and communication bandwidth [6] on edge devices, a high compression ratio becomes necessary.
- *High throughput:* High throughput is desirable in stream analytics, not only at the data center [22], [23], [1], but also at the edge. The high-rate data streams at the edge, collected from massive device links [24], [6], necessitate the capability to compress as much data as possible within a given unit of time.
- *Low energy consumption:* Energy budget is particularly constrained at the edge, and the devices, often far from a stable and constant power supply, are expected to function as long as possible [25]. Thus, energy consumption should be minimized.
- *Low latency:* In many IoT applications, it's crucial to process data and make decisions in real time. Hence, low-latency stream compression is important to ensure timely response and decision-making at the edge.

Meeting these demands simultaneously is a challenging task. For instance, increasing parallelism and allocating more processor cores can achieve higher throughput, but this approach will also increase energy consumption. Therefore, this study aims to reveal the complex relationships between compression ratio, throughput, energy consumption, and latency of stream compression at the edge. We strive to offer comprehensive guidance for achieving satisfying trade-offs among these factors and also explore potential optimal approaches.

3 CSTREAM: SOFTWARE-HARDWARE CO-DESIGN OF STREAM COMPRESSION

Recognizing the unique challenges in edge computing environments, we introduce `CStream`, a comprehensive software-hardware co-designed stream compression framework. `CStream` stands out by accommodating an array of stream compression algorithms, a variety of hardware architectures and configurations, and an innovative set of parallelization strategies. Furthermore, it presents novel concepts, such as asymmetric-aware scheduling, for the first time. By leveraging the features of an advanced stateful compression algorithm and the inherent characteristics of asymmetric multicore platforms, `CStream` efficiently handles diverse data types while meeting hardware resource constraints and maintaining energy efficiency. This section elaborates on the design and components of `CStream`.

TABLE 1: Summary of studied compression algorithms.

Algorithm Name	Fidelity	State utilization	Byte alignment
LEB128-NUQ [26], [27]	lossy	stateless	aligned
ADPCM [26], [27], [1]	lossy	stateful, value	aligned
UANUQ [27], [28]	lossy	stateless	unaligned
UAADPCM [28], [27], [1]	lossy	stateful, value	unaligned
LEB128 [26]	lossless	stateless	aligned
Delta-LEB128 [26], [1]	lossless	stateful, value	aligned
Tcomp32 [28]	lossless	stateless	unaligned
Tdic32 [29], [28]	lossless	stateful, dictionary	unaligned
RLE [30]	lossless	stateful, value	aligned
PLA [31], [13]	lossy	stateful, model	aligned

3.1 Versatility of Compression Algorithms

The heart of `CStream` lies in its support for a wide variety of compression algorithms. It provides versatility to diverse IoT data types and optimizes compressibility while minimizing information loss. Compression algorithms play a significant role in the historical landscape of data processing, with numerous variations proposed since the 1950s [32]. These algorithms typically target improvements in theoretical compressibility or reductions in compression overhead [33], [34]. In constructing `CStream`, we encompassed ten representative compression algorithms that embody key features such as *fidelity*, *state utilization*, and *alignment* (summarized in Table 1).

3.1.1 Fidelity

One of the fundamental distinctions among compression algorithms is their fidelity: the degree to which the original data can be reconstructed from compressed data.

Lossless Compression. Lossless compression is the most stringent form of fidelity, ensuring that the original data can be perfectly reconstructed from its compressed form. Under lossless compression, `CStream` rearranges data into a more compact representation without altering its underlying meaning or losing any information. This principle is strictly bounded by theoretical limits, such as Shannon's entropy [34], which serves as an upper bound on the effectiveness of lossless compression. One exemplar is the $Tcomp32$ algorithm [28] used in `CStream`'s lossless mode. It employs a lossless stream compression technique that suppresses leading zeros [28], a form of bit-level null suppression [35], and its compressibility limit aligns with Shannon's entropy. In this way, `CStream` preserves the accuracy and fidelity of the original data even under compression.

Lossy Compression. Lossy compression, on the other hand, relaxes the fidelity requirement, accepting some loss of information to achieve higher compression ratios. Under lossy compression, `CStream` selectively discards less significant information from the original data. The upper bound on compression effectiveness is then determined by the level of fidelity the system chooses to maintain. For instance, an unaligned non-uniform quantization (UANUQ [27], [28]) with 8 quantization bits would result in a higher compression ratio, and thus more information loss, than one using 12 quantization bits. `CStream`'s lossy mode is designed to strike a balance between high compression ratios and acceptable information loss, accommodating the wide range of fidelity requirements across different application scenarios at the edge.

3.1.2 State Utilization

State utilization, which determines how a compression algorithm uses historical information, is another critical dimension in the design of `CStream`. We divide it into stateless and stateful compression modes, each with unique advantages and best-use scenarios.

Stateless Compression. In stateless compression, `CStream` operates on each data item or tuple independently without reference to any past tuples. This approach essentially replicates a set of operations for each tuple, requiring no state management. This is particularly beneficial in scenarios where the relationships between consecutive tuples are insignificant or when reducing processing overhead is essential. Algorithm 1 provides an overview of `CStream`'s stateless compression mode.

Algorithm 1: `CStream`'s Stateless Compression Mode

```
  Input: input stream inData
  Output: output stream outData
1 while inData is not stopped do
2 |  (s0) load the tuple(s) from inData ;
3 |  (s1) transform and find the compressible parts;
4 |  (s2) output compressed data to outData ;
5 end
```

As a concrete example of stateless compression algorithm, we show the detailed implementation of Android-Dex's $LEB128$ [26] in Algorithm 2.

Algorithm 2: LEB128 algorithm

```
  Input: input stream inData
  Output: output stream outData
1 while not reach the end of inData do
  |  /* (s0)                                          */
2 |  number ← read next 32-bit from inData ;
  |  /* (s1)                                          */
3 |  zeros ← counting the leading zeros in number;
4 |  encoded ← squeeze zeros in number under LEB128 format;
  |  /* (s2)                                          */
5 |  write encoded to outData ;
6 end
```

Stateful Compression. In contrast, stateful compression in `CStream` uses a maintained state to boost compressibility. While the stateful mode often provides superior compression ratios, it incurs additional overhead due to state maintenance. Algorithm 3 illustrates the five-step procedure used in `CStream`'s stateful compression mode.

During `CStream`'s development, we explored three prevalent types of state usage in stream compression:

1) **Value-based state** is the simplest state type, requiring only updates and records of the "last compressed" value. `CStream` employs this state in scenarios where computational efficiency is of utmost importance. Techniques like delta encoding [36], [1] and run-length encoding (RLE) [30] that are considered "lightweight" for both stream and database compression [1], [30], serve as inspiration for this state type.
2) **Dictionary-based state**, although requiring more memory, can greatly enhance compression ratios. A dictionary-based state maintains a collection of previously encountered "last compressed" values. `CStream` leverages this type of state to optimize compression ratios when memory resources permit. It takes full advantage of the L1 cache [29], making dictionary-based compression particularly effective.
3) **Model-based state** approximates data using a model with several parameters. This state type provides high compression ratios when the data stream closely aligns with a certain model. `CStream` uses this state type for such data streams. The piece-wise linear approximation (PLA) technique has been particularly effective for compressing sensor-generated time series [31], [13].

Algorithm 3: `CStream`'s Stateful Compression Mode

```
  Input: input stream inData
  Output: output stream outData
1 while inData is not stopped do
2 |  (s0) load the tuple(s) from inData ;
3 |  (s1) pre-process ;
4 |  (s2) state update ;
5 |  (s3) state-based encoding;
6 |  (s4) output compressed data to outData ;
7 end
```

As a concrete example of stateful compression, we add the delta-encoding (i.e., value-based state) to Algorithm 2, and demonstrate the $Delta - LEB128$ in Algorithm 4.

Algorithm 4: Delta-LEB128 algorithm

```
   Input: input stream inData
   Output: output stream outData
1  state ← 0 ;
2  while not reach the end of inData do
   |  /* (s0)                                          */
3  |  number ← read next 32-bit from inData ;
   |  /* (s1)                                          */
4  |  lastNumber ← state ;
   |  /* (s2)                                          */
5  |  state ← number ;
   |  /* (s3)                                          */
6  |  x ← number-lastNumber ;
7  |  zeros ← counting the leading zeros in x;
8  |  encoded ← squeeze zeros in x under LEB128 format;
   |  /* (s4)                                          */
9  |  write encoded to outData ;
10 end
```

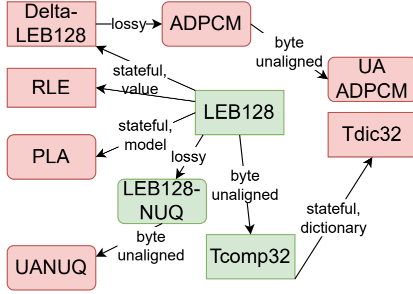


Fig. 2: Incremental relationship among algorithms.

3.1.3 Byte Alignment

The alignment strategy of a compression algorithm, specifically whether it is byte-aligned or not, greatly impacts both compressibility and computational overhead. `CStream` offers both byte-aligned and non-byte-aligned modes to accommodate various computational environments and use cases.

Byte-Aligned Compression. In byte-aligned mode, `CStream` aligns data encoding with byte boundaries, a strategy similar to the $LEB128$ approach [26]. This mode capitalizes on the hardware characteristics of modern processors that manage registers and memories in units of bytes. Despite this, a trade-off exists as the output code length must be an integer multiple of a byte (i.e., 8 bits), potentially leading to some waste of bits.

Non-Byte-Aligned Compression. Non-byte-aligned mode in `CStream` works to mitigate bit-level waste. However, this comes at the expense of additional computational overhead due to the need for more logical and bit-shift operations, such as AND/OR. Because the minimal unit of output code length is one bit instead of one byte, appending and extracting data operations may require extra computational instructions. Despite the increased overhead, this mode can significantly increase the compression ratio in specific scenarios, thereby providing a valuable option in the `CStream` framework.

3.1.4 Compression Implementation

The implementation of our algorithms follows a step-by-step refinement and optimization process, illustrated in Figure 2. This approach allows us to incrementally examine the impact of each algorithmic variation, comparing pairs of sequentially connected algorithms.

Initially, we implement the LEB128 algorithm in the same way as Android-Dex [26], which serves as our foundation for stateless and byte-aligned stream compression. Next, we adapt LEB128 to allow byte-unaligned output by simplifying Elias encoding [28] into $Tcomp32$. This adjustment involves removing leading zeros from each 32-bit tuple and calculating the prefix based on the length of non-zero bits.

By changing the encoding of the LEB128 into non-uniform quantization [27], we can implement the lossy compression $LEB128-NUQ$, which can be further equipped with byte-unaligned output by upgrading into $UANUQ$. Besides the stateless lossy extension, $LEB128$ and $Tcomp32$ can be also enriched with compression states.

TABLE 2: Edge computing processors studied.

Processor Name	Processor Architectures	Byte Length	ISA Pattern	Frequency Range/GHz	Core Number
$H2+$ [10]	SMP	32 bit	RISC	0.416 $\sim$ 1.2	4
$RK3399^{AMP}$ [9]	AMP	64 bit	RISC	0.416 $\sim$ 1.8	6
$RK3399^{SMP}$ [9]	SMP	64 bit	RISC	0.416 $\sim$ 1.8	6
$Z8350$ [11]	SMP	64 bit	CISC	0.8 $\sim$ 1.92	4

$Tdic32$ utilizes a LZ4-like hash table [29] as its state, which tracks duplication across 4096 table entries by default. Each 32-bit tuple is replaced by its shorter hash index if the tuple has appeared previously, as tracked by the table. In contrast, Delta-LEB128 maintains a simple state: it computes the difference between two consecutive values, termed as the "delta" state [1], [36], and applies LEB128's method to output the "delta" state instead of the original value. The well-known commercial algorithm run-length-encoding (RLE) [30] resembles $Delta-LEB128$ in a value-based state, but it can further track the continuous duplication times of the state value. Besides value and dictionary, we can also upgrade $LEB128$ into model-based stateful compression PLA [31], [13], which utilizes piecewise linear approximation as the model.

Finally, we implement a form of lossy compression by modifying the "delta" state encoding in Delta-LEB128, leading to the creation of the Adaptive Differential Pulse-Code Modulation (ADPCM). Specifically, we employ lossy non-uniform quantization [27] in ADPCM to encode the "delta" state, instead of the lossless LEB128 style. The ADPCM can be further equipped with byte-unaligned, $Tcomp32$-like output format as $UAADPCM$.

3.2 Accommodating the Multicore Edge Landscape

This section discusses the fundamental aspects of the multicore edge hardware that influence `CStream`'s implementation, making it conducive to various edge environments. It delves into the choices of architecture and Instruction Set Architecture (ISA), and extends the discussion to the utilization and regulation of hardware resources. The edge processors under examination are detailed in Table 2.

3.2.1 Processor Architecture: Symmetric and Asymmetric

In the context of multicore processors, the architectural design can be broadly bifurcated into symmetric and asymmetric models.

Symmetric Architecture. Symmetric multicore processors (SMPs) are not exclusive to center servers but are equally viable for edge devices. For instance, the UP board platform [37] supported by `CStream` employs a 4-core SMP z8350 [11]. The primary distinction between the SMPs used at the edge and at the center lies in their energy efficiency: each core in an edge SMP is generally more energy-optimized and less powerful than its center counterpart. `CStream` leverages the inherent simplicity and unified nature of the SMP architecture at the edge for straightforward parallel execution, facilitating the transfer of existing optimizations from cloud SMPs [38], [39].

Asymmetric Architecture. Aiming for a balanced trade-off between performance and energy efficiency, asymmetric architectures offer a compelling alternative.

By featuring various types of execution units on a single processor, these architectures open up new avenues for optimization in stream compression. This work emphasizes asymmetric architectures within single ISA, known as asymmetric multicore processors (AMPs) [40]. However, it acknowledges the potential of other forms of asymmetry, such as edge CPU+GPU [41], [42], CPU+DSP [43], and CPU+FPGA [44], as promising areas for future exploration.

3.2.2 Instruction Set Architecture

`CStream`'s flexibility shines through its support for a variety of ISAs, thereby covering the broad scope of edge devices.

32-bit vs. 64-bit. While 32-bit processors maintain a foothold in the edge domain due to their long-standing development ecosystem and lower cost, 64-bit processors are increasingly gaining traction due to their enhanced memory addressing efficiency and speed. `CStream` extends its support to both variants, facilitating a balanced comparison within the realm of stream compression.

CISC vs. RISC. The trade-off between the flexibility of complex (CISC) and the efficiency of reduced (RISC) instruction set architectures is dependent on specific applications. `CStream` explores this dynamic in the context of stream compression. It recognizes the growing popularity of RISC architectures, especially ARM processors, for edge devices [25].

3.2.3 Energy-Efficient Resource Management

Once a specific architecture and ISA have been selected, `CStream` provides the means to finely tune hardware resources, thereby enhancing the balance between processing time and energy consumption.

Frequency Regulation. The processor's clock frequency directly influences its performance and power consumption. By executing more instructions per unit of time, a higher frequency leads to reduced processing time and increased throughput but at the cost of elevated energy consumption. Conversely, lower frequencies are more energy-efficient but result in lower processing rates. `CStream` facilitates exploration of this trade-off by allowing stream compression tasks to be run at varying frequency settings. Moreover, the capability extends to dynamic frequency scaling using Dynamic Voltage and Frequency Scaling (DVFS) technology [45], [46], [47]. Though changes in frequency introduce overheads, `CStream` offers the potential to explore whether the advantages of DVFS outweigh these costs in the context of stream compression.

Core Number Regulation. While cloud-based stream processing frameworks [22], [23] typically utilize all available computational resources in pursuit of maximum performance, edge-based systems need to be more energy-conscious. Consequently, it might be beneficial to turn off or idle certain processor cores during stream compression operations to conserve energy. `CStream` enables investigation into this trade-off between energy consumption and processing throughput by allowing variable core usage, thereby further enhancing its adaptability to diverse edge environments.

3.3 Integrated Software and Hardware Optimization

`CStream` optimizes stream compression by finely tuning software strategies and hardware configurations to create a synergistic interplay that maximizes both efficiency and performance. Central to this integration is `CStream`'s comprehensive suite of parallelization strategies. The nuances of this integration will be unpacked in the subsequent sections.

3.4 Execution Strategies

The choice between Eager and Lazy Execution depends on stream data characteristics and hardware resources.

Eager Execution. Eager execution aligns closely with the inherent nature of streaming data—infinitely incremental [48], [49]. As each tuple arrives, it is compressed instantly. However, while this strategy reflects the streaming model, it can lead to inefficient hardware utilization due to frequent partitioning, synchronization, and potential cache thrashing.

Lazy Execution. To counteract the potential inefficiencies of Eager Execution, `CStream` introduces Lazy Execution. This strategy batches several tuples together before compression, a practice known as *micro-batching* [50], [1]. While this approach reduces communication overhead and promotes better hardware utilization, selecting an appropriate batch size can be challenging, necessitating a careful balance between frequent cache flush and reload operations and not overburdening slower memory storage.

3.4.1 State Management Strategies

Parallelizing stateful stream compression demands careful consideration of state management and sharing. `CStream` provides three key strategies: State Sharing and Private State.

State Sharing. State Sharing allows all threads to share a single state, with locks implemented to prevent write conflicts. While this method can theoretically offer higher compressibility due to a collective record of past tuples, concurrency control overhead may hinder parallelism and impact performance.

Private State. In contrast, a private state lets each thread maintain its own state, thereby eliminating concurrent conflicts. This strategy maximizes parallelism but potentially reduces compressibility, as the isolated states limit a thread's awareness of tuples handled by others.

3.4.2 Scheduling Strategies

For efficient workload distribution across multicore processors, `CStream` utilizes two main scheduling strategies: Simple Uniform Scheduling and Asymmetric-aware Scheduling.

Simple Uniform Scheduling. In Symmetric Multicore Processor (SMP) environments, `CStream` uses a "balanced partition and equal distribution ratio" approach to scheduling [39]. This method takes advantage of the symmetry inherent in SMPs, where all cores have equal computational capacity and communication distance.

Asymmetric-aware Scheduling. For Asymmetric Multicore Processors (AMPs), which have cores with different computational abilities and communication

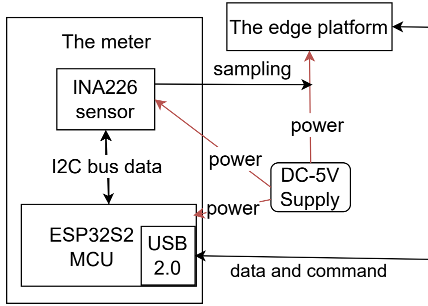


Fig. 3: The design of energy meter.

distances, CStream employs Asymmetric-aware Scheduling [4]. This strategy tailors workload distribution to the computational intensity of different stream compression stages, thereby optimizing hardware utilization.

3.5 Energy Metering

Understanding and accurately measuring energy consumption is of paramount importance in edge computing, where power constraints are often more stringent than in conventional data centers. To support this need, CStream includes a custom-developed energy meter, as illustrated in Figure 3, to accurately measure energy consumption across various edge platforms. The detailed specification is shown in Table 3.

This energy metering component consists of Texas Instrument's INA226 [51] chip, which acts as a sensor for current and voltage, and the Espressif's ESP32S2 [52] micro control unit (MCU) for data pre-processing and USB-2.0 communication with targeted asymmetric multicores. In this configuration, the meter offers precise measurement capabilities while maintaining a low overhead. Additionally, for the UP board platform, it is possible to directly read energy consumption data from X64 msr registers [53] before and after an experimental run.

Accurate Measurement. The integration of the INA226 chip and the ESP32S2 MCU in our energy metering system facilitates high-precision measurement of energy usage. This accuracy extends from the overall system level down to the granularity of individual operations, providing deep insights into the energy profile of the stream compression process.

Low Overhead. Our energy metering system has been designed for minimal overhead. The ESP32S2 MCU handles data pre-processing, and the use of USB-2.0 communication ensures that the monitoring process does not interfere with the primary tasks of stream compression and parallel execution.

Real-time Feedback. The energy metering system offers real-time feedback on energy consumption. This feedback enables dynamic adjustments of execution strategies, state management, and scheduling based on current energy usage, thus supporting the goal of energy-efficient stream compression.

Comprehensive Integration. The energy metering system is tightly integrated with the rest of the CStream framework. This integration allows it to work synergistically with CStream's software strategies and hardware configurations, guiding optimizations and adaptations based on up-to-date energy consumption data.

TABLE 3: Specifications of the energy meter

Sensor chip	INA226
MCU chip	ESP32S2
Interface	USB 2.0 Full
Raw resolution	16 bits
LSB	1.25 mV, 2mA
Measurement range	0-15V, 0-4A
Sample rate	1K SPS

4 Methodology

This section presents our methodological approach to assessing the effectiveness and performance of CStream. It provides a comprehensive explanation of the chosen performance metrics, the benchmark workloads utilized, and the selection and characteristics of the hardware platforms involved in the evaluation.

4.1 Performance Metrics

In our evaluation of CStream's performance, we focus on a range of metrics, each providing insight into different aspects of its functionality. These performance indicators, initially introduced in Section 2, include *compression ratio*, *throughput*, *energy consumption*, and *end-to-end latency*. For each metric, we aim to establish a consistent measurement approach. The average value of these metrics is calculated over a substantial data volume to avoid fluctuations and ensure measurement consistency - specifically, over 932800 bytes of tuples.

The *compression ratio* and *normalized root mean square error (NRMSE)* are calculated by comparing the compressed data against the raw input. The NRMSE, specifically used for lossy compression, is defined as $NRMSE = \frac{1}{\bar{x}} \times \sqrt{\frac{\sum_i^N (x[i]-y[i])^2}{N}}$, where $\bar{x}$ represents the average value of the input data stream, $x[i], y[i]$ denote the individual values of the raw input and the reconstructed data from compression, respectively, and N is the total input data volume.

Throughput and *end-to-end latency* provide insights into the system's operational efficiency. Throughput is measured as $throughput = \frac{N}{processing\ time}$, where the processing time is obtained using OS-specific APIs such as *gettimeofday*. End-to-end latency, an important metric in edge computing scenarios requiring real-time or near-real-time data processing, quantifies the total time for a data element to traverse the compression system from input to output.

Energy consumption evaluation follows a two-step procedure. First, we measure and eliminate the static energy consumption caused by irrelevant hardware or software components, such as the Ethernet chip and back-end tcp/ip threads. Next, we monitor the energy consumption during the running of the stream compression benchmark, without incurring additional overhead or interference. The specifics of energy recording are platform-dependent and are discussed in Section 3.5.

4.2 Benchmark Workloads

In order to comprehensively evaluate CStream across a diverse range of IoT use cases (discussed in Section 2.2), we use five representative IoT datasets. These datasets were chosen to reflect various data sources (single and multiple) and diverse data structures (plain, binary structured, and textual structured). Furthermore, we assess the compressibility of data from two perspectives: *stateless compressibility* and *statefu; compressibility*, utilizing a synthetic dataset to calibrate these properties.

The *stateless compressibility* refers to the compressible space within each individual tuple $x_t = v$, which can be exploited by both stateless and stateful stream compression (refer to Section 3.1.2). On the other hand, *stateful compressibility* points to the compressible space hidden in the context of the data stream, considering the current tuple x_τ and some past tuples $\{x_t | t < \tau\}$ together. This can only be fully exploited by a suitable stateful compression.

Our selected datasets are summarized in Table 4 and are detailed below:

1) **ECG** [16]: The ECG dataset consists of raw ADC recordings from electrocardiogram (ECG) monitoring provided by the MIT-BIH database. Each reading is packaged as a plain 32-bit value for our evaluation. As ECG is a direct, unstructured reflection of a continuous physical process, it exhibits the highest levels of both independent and associated compressibility.
2) **Rovio** [17]: The Rovio dataset continuously monitors user interactions with a specific game to ensure optimal service performance. Each data entry consists of a 64-bit key and 64-bit payload. The Rovio dataset exhibits two compressible traits: first, its payload is constrained to a relatively small dynamic range, indicating independent compressibility; second, different tuples may share the same key, which demonstrates associated compressibility.
3) **Sensor** [19]: The Sensor dataset is comprised of full-text streaming data generated by various automated sensors (e.g., temperature and wind speed sensors). For our evaluation, every 16 ASCII characters in the Sensor dataset forms one 128-bit tuple. The Sensor dataset primarily exhibits associated compressibility due to the repetition of several fixed XML patterns across different tuples.
4) **Stock** [18] and **Stock-Key** [18]: The Stock dataset is a real-world stock exchange dataset packed in a (32-bit key, 32-bit payload) binary format. It exhibits less compressibility than Rovio due to fewer key duplications. Stock-Key is a subset of the Stock dataset, containing only the 32-bit keys.
5) **Micro**: The Micro dataset is a synthetic 32-bit dataset [54], used to further tune independent and associated compressibility. We can adjust its dynamic range to control independent compressibility and its level of duplication to control associated compressibility.

To mitigate the impact of network transmission overhead, all input datasets are preloaded into memory before testing. Each tuple is assigned a timestamp (starting from 0) to reflect its actual arrival time to the system, and tuples are time-ordered. These timestamps, which are stored separately from each tuple and are not subjected to compression, help provide a realistic simulation of data arrival in a real-world scenario. Unless otherwise specified, we generate incremental timestamps evenly to simulate an average arrival speed of 16×10^6 bytes per second (e.g., 10^6 tuples per second for the Rovio dataset, which consists of 128-bit tuples).

TABLE 4: Dataset Studied.

Dataset Name	Source	Data Structure	Stateless Compressibility	Stateful Compressibility
ECG [16]	single	plain	high	high
Rovio [17]	multiple	binary structured	medium	medium
Sensor [19]	multiple	textual structured	low	high
Stock [18]	multiple	binary structured	low	medium
Stock-key [18]	multiple	plain	low	medium
Micro [54]	single	plain	adjustable	adjustable

TABLE 5: Evaluated hardware platforms

Model	Supported processor	Number of cores	Installed memory
Banana pi zero-m2 [56]	$H2+$	4	512MB
Rockpi 4a [55]	$RK3399$	2 big+4 little	2GB
UP board [37]	$Z8350$	4	1GB

4.3 Edge Computing Platforms

To ensure a comprehensive and hardware-variant evaluation, we deploy three distinct edge computing platforms, each featuring unique characteristics as outlined in Table 5. Importantly, all platforms are compatible with the mainline Linux kernel and support the Glibc with C++20 features, thereby enabling a shared codebase.

The Rockpi 4a [55] serves as our default evaluation platform. Unless otherwise specified, we use its processor as $RK3399^{AMP}$ and engage each core to operate at its maximum frequency: 1.8GHz for the larger cores (core4 to core5) and 1.416GHz for the smaller cores (core0 to core3).

5 Evaluation

In this section, we embark on an experimental journey to evaluate the potency of various stream compression schemes, particularly on edge platforms. This evaluation focuses on a strategic software-hardware co-design as explicated in Section 3. We delve into five key areas:

1) An end-to-end case study of CStream's solution space is demonstrated in Section 5.1
2) The selection and performance of different stream compression algorithms, are explored in Section 5.2.
3) The impact and considerations of hardware variants, are investigated in Section 5.3.
4) The effectiveness and scalability of novel parallelization strategies, are examined in Section 5.4.
5) The sensitivity and adaptability to varying workload characteristics, assessed in Section 5.5.

Our exploration culminates in Section 5.6, where we collate our findings. Unless otherwise specified, we adopt the following parallelization strategies: 1) lazy execution

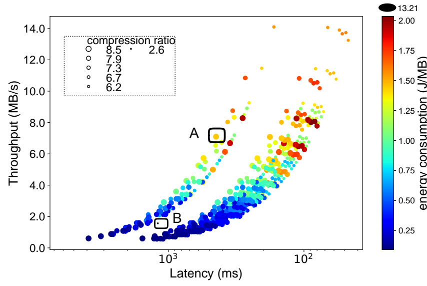


Fig. 4: The solution space of `CStream` under $\geq$ 6.0 compression ratio, $\leq 5\%$ NRMSE, and RK3399 hardware.

with a 400-byte micro batch, 2) a regulated multicore workload distribution ratio to optimize throughput, and 3) the use of a non-shared state for stateful stream compression.

5.1 End to End Case Study

In this case study, we have ECG data to be compressed on RK3399 hardware, the compression ratio is expected to be larger than 6.0, while the NRMSE should be controlled below 5%. `CStream` will hence choose the PLA algorithm, and the whole available solution is provided as colorful points in Fig 4. In general, higher energy consumption is required when higher throughput, higher compression ratio, or lower latency is expected, and the specific optimal solution depends on users' prioritization of performance metrics. For instance, if the user further wants to maximize the compression ratio, and then maximize throughput, while keeping the energy consumption within $1.5J/MB$, the optimal one is labelled as point A. Specifically, it utilizes 1 big core and 1 little core under their highest frequency, lets each of them use a private PLA state, and schedules the workload in an Asymmetric-aware manner. Both cores execute the stream compression under a $8KB$ micro-batch.

We also marked a careless solution point B in Fig 4 as a contrast. This solution conducts a $Tdic32$ compression on 2 big cores and 4 little cores, the $Tdic32$ state is shared by all cores, and OS scheduling is used along with *on-demand* DVFS. All cores use an eager strategy in conducting stream compression. Note that, the optimal solution A achieves $2.8\times$ compression ratio, $4.3\times$ throughput, 65% latency reduction, and 89% energy consumption reduction simultaneously than the careless solution B.

5.2 Algorithm Evaluation

In order to evaluate the performance of `CStream`'s support for diverse stream compression algorithms, we test ten distinct algorithms, as summarized in Table 1, on five real-world IoT datasets, as detailed in Table 4.

5.2.1 Fidelity: Lossless vs. Lossy Compression

Referencing Figures 5a, 5b, 5c, and 5d, the distinction between lossless and lossy compression becomes apparent. Lossy compression, represented by $LEB128-NUQ$, offers a superior compression ratio (between 2.0 and 6.0) across all datasets. Conversely, $LEB128$, a lossless algorithm,

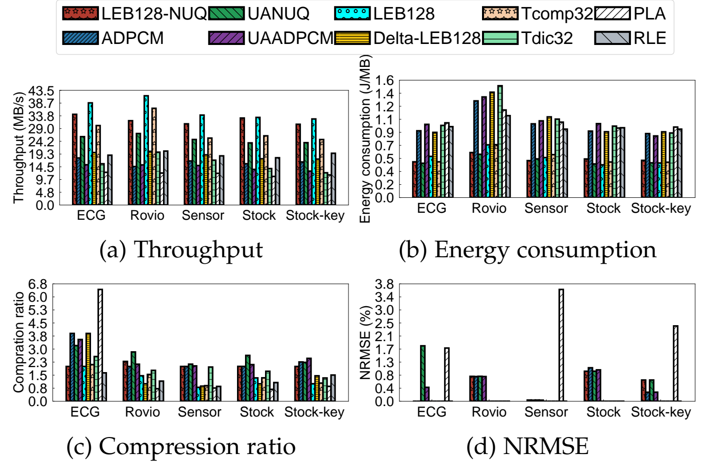


Fig. 5: Comparing ten algorithms on five datasets.

struggles to exceed a ratio of 2.0. Remarkably, this high compression ratio by $LEB128 - NUQ$ introduces only marginal information loss, with a NRMSE below 3.8%.

5.2.2 State Utilization: Stateless vs. Stateful Compression

On comparing $Tcomp32$ (stateless) with $Tdic32$ (stateful, dictionary-based), we observe a trade-off between compression cost and effectiveness. While $Tcomp32$ offers more modest compression ratios, it presents a less complex, lower-cost compression process. On the other hand, $Tdic32$, despite its higher processing cost due to dictionary-based state management, excels in handling text data streams with high associated but low independent compressibility, like the Sensor dataset.

5.2.3 State Implementation: Value, Dictionary, and Model

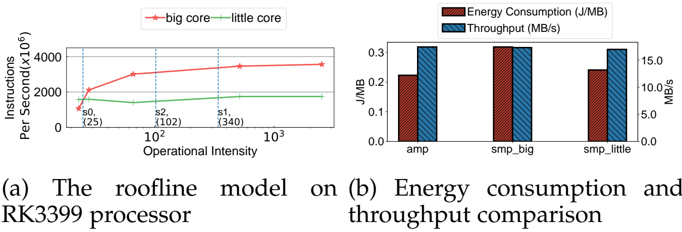
Further comparisons among $ADPCM$ (stateful, value-based), $Tdic32$ (stateful, dictionary-based), and PLA (stateful, model-based) illustrate how different state implementations can impact compression performance. While $ADPCM$ consistently leads in throughput and energy consumption, $Tdic32$ and PLA shine in handling structured data or data from multiple sources, offering higher compression ratios.

5.2.4 Byte Alignment Variations

When comparing byte-aligned $LEB128$ and byte-unaligned $Tcomp32$, we note a trade-off between compression ratio and computational cost. $Tcomp32$ achieves higher compression ratios but at the expense of added computational overhead, given its bit-by-bit encoding process.

5.3 Hardware Evaluation

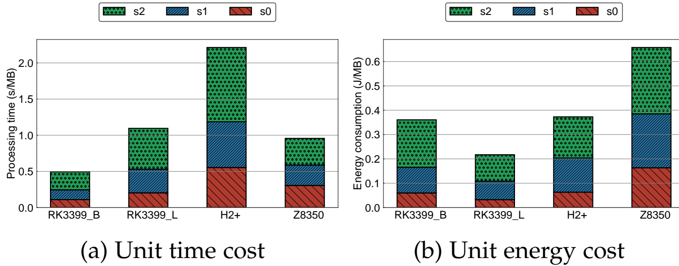
In this section, we highlight how our novel framework, `CStream`, explores various hardware design spaces, including architecture selection, ISA selection, frequency regulation, and core count adjustment. These are critical aspects in ensuring optimal energy consumption and throughput. We use the $Tcomp32$ algorithm and Rovio dataset as our primary test cases for this evaluation.



(a) The roofline model on RK3399 processor

(b) Energy consumption and throughput comparison

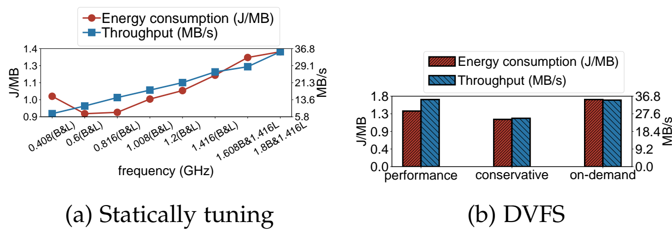
Fig. 6: Impacts of processor architectures for $Tcomp32$.



(a) Unit time cost

(b) Unit energy cost

Fig. 7: Impacts of ISA for $Tcomp32$.



(a) Statically tuning

(b) DVFS

Fig. 8: Impacts of frequency regulation for $Tcomp32$.

5.3.1 Architecture Selection

CStream allows us to study the impact of different multicore configurations. We compare symmetric and asymmetric multicores under the same total computational power (i.e., maximum instructions per second). The roofline model benchmark [57], [58] shows that one 1.416GHz A72 big core in the RK3399 processor has about twice the computational power of one A53 little core at the same frequency (Figure6a). Therefore, we compare different architecture configurations using the following settings:

- **Asymmetric multicore processor** (amp): Using 1 A72 big core and 2 A53 little cores at 1.416GHz.
- **Symmetric multicore processor with only big cores** (smp_big): Using 2 A72 big cores at 1.416GHz.
- **Symmetric multicore processor with only little cores** (smp_little): Using 4 A53 little cores at 1.416GHz.

Each architectural case is tuned to its maximum throughput, after which we compare their energy consumption along with throughput (Figure 6b). The asymmetric multicore configuration (amp) outperforms the symmetric configurations (smp_big and smp_little), achieving both the lowest energy consumption and the highest throughput.

Furthermore, we observe that different stream compression steps (i.e., $s0 \sim s2$ in Algorithm 1) involve varying *operational intensities*[59], [60], [61], [62] (i.e., instructions per memory access), as shown by the dashed lines in Figure6a. This difference in operational intensities causes either over-provision or under-provision when conducting stream compression on symmetric multicores, thereby increasing energy consumption.

The $s0$ step leads to less performance gain when run on a big core than $s1$ and $s2$, as it primarily involves memory manipulation and makes out-of-order big cores over-provisioned. Therefore, much energy is wasted in the smp_big configuration. Conversely, $s1$ and $s2$ are more worthwhile running on big cores as they offer enough computation density for big cores to support. Their high computation density also makes the little cores under-provisioned, resulting in energy dissipation in the smp_little configuration. Through CStream, we efficiently manage this architectural selection and intricacies, ensuring optimal energy usage and performance.

5.3.2 ISA Selection

CStream also enables us to investigate the effects of different Instruction Set Architectures (ISA) on stream compression performance. In this evaluation, we consider the RK3399, H2+, and Z8350 hardware platforms, as introduced in Section 4.3.

Specifically, we compare the time and energy cost per core for each step of $Tcomp32$ under different ISAs (Figure 7a and 7b). To account for their different working frequencies, we mathematically align their frequency to 1 GHz. The results for RK3399 are split into $RK3399_B$ and $RK3399_L$, representing its big and little cores, respectively.

Our evaluation highlights two key findings. First, the RISC ISA (used in RK3399) demonstrates superior performance compared to the CISC ISA (used in Z8350). Both RK3399_B and RK3399_L cores have significantly lower unit energy costs than the Z8350, and the RK3399_B can even reduce the processing time of each step by 50% compared to the Z8350. This is because RK3399's RISC-based execution hardware can directly execute the instructions used for stream compression, avoiding the extra overhead of micro-coding them. As a result, both unit latency and unit energy cost are reduced.

Second, traditional 32-bit processors (like the H2+) perform poorly in terms of both performance and energy efficiency. This is due to the limitations of a shorter register length. For example, manipulating a 33-bit intermediate result of $Tcomp32$ can be done with a single instruction on a single 64-bit register but requires two or more operations on 32-bit registers. This inefficiency at the instruction and register levels leads to significantly increased latency and is detrimental to energy efficiency.

From our analysis, we conclude that stream compression tasks should ideally be conducted on 64-bit RISC edge processors. These insights, derived from CStream's evaluation capabilities, underscore the importance of choosing the right ISA for efficient stream compression on edge devices.

5.3.3 Frequency Regulation

CStream provides flexibility in frequency regulation, allowing for both static frequency setting and dynamic voltage and frequency scaling (DVFS). In our evaluation, we

explore these approaches' impact on throughput and energy consumption.

Statically Tuning the clock frequency. We first adjust the frequency of the big cores (denoted as "B") and the little cores (denoted as "L") on the RK3399 processor. We observe how frequency changes influence the throughput and energy consumption, as shown in Figure 8a. As expected, the relationship between frequency and throughput is nearly linear: higher frequency enables cores to execute more operations per unit time, and therefore more tuples are compressed.

Energy consumption, however, doesn't follow a simple monotonic relationship with frequency. It is the product of power and time, both of which respond differently to frequency changes. For example, a frequency of 0.408 GHz leads to lower power according to existing work [45], [46], but also involves more processing time, which counteracts the power reduction. Thus, it's less energy-efficient than the 0.6 GHz frequency. When the frequency surpasses 0.816GHz, the energy consumption increases with frequency because the rise in power is more significant than the time reduction.

DVFS. We also employ the DVFS approach [45], [46], [47] to dynamically adjust the frequency. We use different DVFS strategies and present the results in Figure 8b. The "performance" strategy, which fixes each core at its highest frequency without dynamic reconfiguration (1.8GHz for big cores and 1.416GHz for little cores), serves as a reference. The "conservative" and "on-demand" strategies attempt to reduce energy consumption by dynamically reconfiguring frequency. The primary difference is that the "conservative" strategy changes frequency less frequently than "on-demand".

Our results indicate that the "conservative" strategy can further reduce energy consumption by 15% compared to the default "performance strategy", albeit at the cost of a 38% increase in latency. This strategy offers a coarser-grained balance of energy efficiency and latency constraints, since the overhead of dynamic frequency regulation can introduce latency variability. In contrast, the "on-demand" strategy doesn't provide any benefits; it actually increases both latency and energy consumption due to the high overhead of frequent frequency switches.

Through this exploration, `CStream` highlights the critical role of frequency regulation in achieving energy-efficient stream compression, guiding us towards more efficient hardware configurations and settings.

5.3.4 Tuning the Number of Cores

Finally, we demonstrate the ability of `CStream` to effectively manage the number of cores for stream compression tasks. The results are shown in Figure 9. By enabling different numbers of big and little cores, we observe a trade-off between energy consumption and throughput. This demonstrates `CStream`'s flexible core management strategy, striking a balance between energy efficiency and throughput.

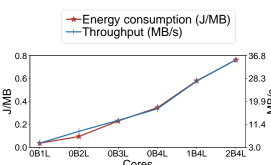


Fig. 9: Scalability

5.4 Parallelization Strategies

We now explore various parallelization strategies provided by `CStream`, including different execution strategies, micro-batch size, state sharing, and scheduling strategies. Our exploration, illustrated with the $Tcomp32$ algorithm and the Rovio dataset, illustrates the importance of each strategy and its impacts on energy consumption and throughput.

5.4.1 Execution Strategy: Eager vs Lazy

A key component of our `CStream` framework is the ability to vary the execution strategy. Specifically, we explore the differences between eager and lazy execution strategies using the $Tcomp32$ algorithm and the Rovio dataset.

Under eager execution, each incoming tuple is compressed as soon as it arrives. On the contrary, under lazy execution (implemented as the default setting in `CStream`), compression is conducted only after the accumulation of a 400-byte micro-batch. These strategies do not affect the compression ratio but significantly influence throughput and energy consumption.

Figure 10a clearly illustrates the superiority of the lazy strategy, leading to higher throughput and lower energy consumption. The eager approach incurs a high energy cost because it must immediately distribute each incoming tuple to the running cores, thus reducing parallelism. In contrast, the lazy strategy enhances parallelism by accumulating workloads before distribution.

To better understand the cost difference between the two execution strategies, we break down the processing time into 'blocked' and 'running' phases, as presented in Table 10b. 'Blocked' time is spent resolving concurrent read/write conflicts and mitigating cache trashing, while 'running' time is devoted to the actual compression process. The eager execution strategy incurs significantly more blocked time, leading to high overhead and reduced energy efficiency.

In our study of execution strategies, we further investigate the effects of varying batch sizes during lazy execution. Continuing with the $Tcomp32$ algorithm and Rovio dataset, we vary the batch size from hundreds of bytes to millions of bytes. The impacts of batch size on energy consumption and throughput are illustrated in Figure 11a.

The results show that both minimum energy consumption and maximum throughput occur simultaneously at an optimal, moderate batch size. Any deviation from this size—either by increasing or decreasing the batch—leads to reduced throughput and increased energy consumption. This finding underscores the importance of optimizing batch size for efficient stream compression on edge devices.

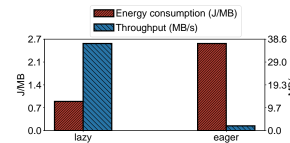
Significantly, we draw attention to a correlation between the optimal batch size and the total L1D cache size of all cores, indicated by the red dashed line in Figure 11a. The closeness of these values suggests that efficient micro-batching should be cognizant of the L1D cache size. Hence, `CStream`'s strategy for lazy execution, which includes an L1D-cache-aware approach to micro-batching, can offer substantial advantages in terms of energy efficiency and throughput.

As we continue to evaluate the implications of the batch size in our lazy execution strategy, we now turn our attention to the role of latency. Using the $Tcomp32$ algorithm and the Rovio dataset, we consider average latency under different batch sizes. These results are presented in Figure 11b.

Interestingly, we observe a curve where latency first decreases and then increases as the batch size changes. This inverted U-shape pattern implies a trade-off in determining the optimal batch size. It is crucial to balance the competing requirements of lower latency and improved throughput and energy efficiency.

Furthermore, the increase in latency appears to correlate strongly with the total L1D cache size of all cores, highlighted by the red dashed line in Figure 11b. Upon exceeding this cache size, latency sees a noticeable increase, which can be attributed to the higher likelihood of cache misses as the batch size outstrips the available cache.

These observations reinforce the importance of an L1D-cache-aware approach to setting the batch size in `CStream`'s lazy execution strategy. Balancing the demands of latency, throughput, and energy efficiency requires a sophisticated approach to micro-batching — a challenge that `CStream` effectively navigates with its adaptive and hardware-conscious design.

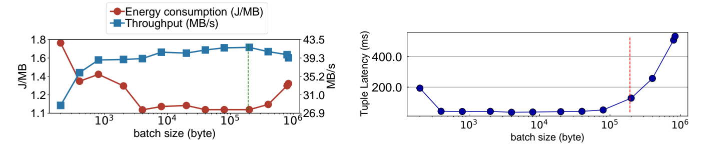


(a) Energy and throughput consumption

Execution way	Blocked time	Running time
Eager	0.53	0.03
Lazy	0.006	0.03

(b) Processing time breakdown

Fig. 10: Impacts of execution strategy for $Tcomp32$.



(a) Energy and throughput consumption

(b) Average latency; The red dashed line is the total L1D size of all cores

Fig. 11: Impacts of batch sizes for $Tcomp32$.

5.4.2 State Management Strategy: Shared vs Private

One of the critical considerations within the realm of stateful stream compression in `CStream` is the choice between sharing the state across threads or maintaining private states for each thread. We utilized the $Tdic32$ algorithm and the Rovio dataset to investigate the trade-offs associated with each state management strategy.

In the shared state scenario, each working thread shares a common concurrent dictionary. In contrast, with private state management, each thread maintains its private dictionary. These two strategies are elaborated further in Section 3.1.2. Figure 12a compares the resulting energy consumption and throughput for each scenario.

Interestingly, state sharing incurs a significantly higher energy consumption and concurrently reduces throughput. However, the shared state strategy only marginally improves the compression ratio, moving from 1.78 without state sharing to 1.81 with state sharing. In most cases, pursuing a mere 3% improvement in compression ratio is not worthwhile, especially considering the overhead involved.

To delve deeper into the source of the increased overhead with shared state management, we broke down the processing time for each step, as defined in Algorithm 3, for the two implementation versions. This breakdown is presented in Figure 12b.

The analysis revealed that the bulk of the added cost is accrued during the state updating step (i.e., $s2$), where frequent locking significantly reduces parallelism. The cores, while waiting for locks, are mainly engaged in memory access, which is less energy-demanding than arithmetic calculations. This explains why the rise in energy consumption is not as pronounced as the drop in throughput with shared state management.

This in-depth analysis underscores the preference for a private state management strategy in `CStream`'s stateful stream compression, enhancing throughput and energy efficiency without significantly compromising the compression ratio.

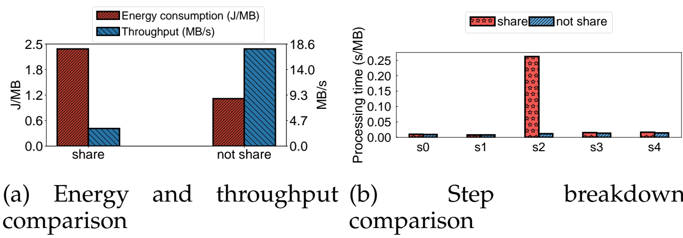
5.4.3 Varying Scheduling Strategy

In order to determine the optimal scheduling strategy, we explore both symmetric and asymmetric approaches using the $Tcomp32$ algorithm and the Rovio dataset. As the scheduling strategy doesn't influence the compression ratio, we focus our evaluation on energy consumption and throughput, as presented in Figure 13a.

Interestingly, symmetric scheduling results in both a reduction in throughput (26.2%) and an increase in energy consumption (13.4%) compared with asymmetric scheduling. The underlying reason is that symmetric scheduling fails to take into account the differences in hardware, as we have previously illustrated in Figure 6a. This oversight leads to a wastage of the superior computational power of big cores.

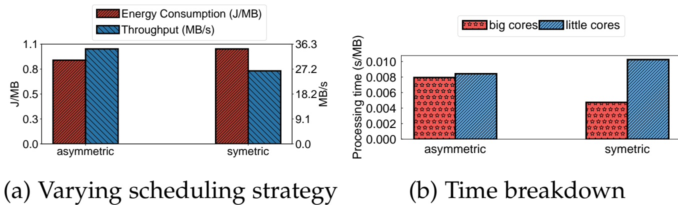
For instance, Figure 13b shows the processing time for the $s1$ step of the $Tcomp32$ algorithm for both big and little cores. It becomes evident that under symmetric scheduling, big cores spend a considerable amount of time waiting for little cores. This inefficient waiting period is the primary contributor to the sub-optimal energy efficiency and performance under symmetric scheduling.

To circumvent this issue, we strongly recommend asymmetric scheduling. This approach respects the unique computational capabilities of different cores, efficiently allocating tasks in a manner that optimizes performance and energy usage. As a result, it achieves higher throughput and lower energy consumption than symmetric scheduling, demonstrating its superiority in managing heterogeneous multi-core systems for stream compression tasks.



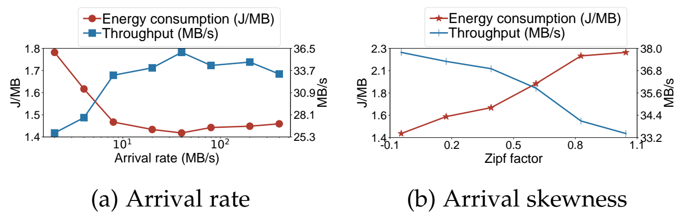
(a) Energy and throughput comparison (b) Step breakdown comparison

Fig. 12: Impacts of state management strategy for $Tdic32$.



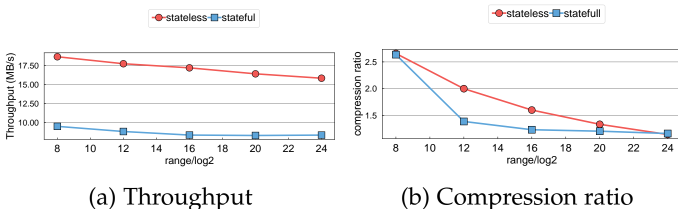
(a) Varying scheduling strategy (b) Time breakdown

Fig. 13: Impacts of scheduling strategy for $Tcomp32$.



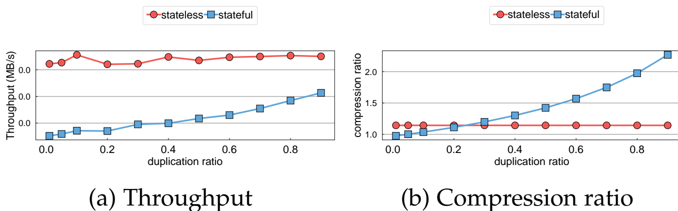
(a) Arrival rate (b) Arrival skewness

Fig. 14: Impacts of arrival pattern for $Tcomp32$.



(a) Throughput (b) Compression ratio

Fig. 15: Impacts of dynamic range for stateless ($Tcomp32$) and stateful stream compression ($Tdic32$).



(a) Throughput (b) Compression ratio

Fig. 16: Impacts of tuple duplication for stateless ($Tcomp32$) and stateful stream compression ($Tdic32$).

5.5 Workload Sensitivity Study

In this subsection, we illustrate the robustness of our framework, CStream, by conducting a comprehensive sensitivity analysis based on varying IoT data workloads. Our goal is to assess how CStream responds under diverse arrival patterns and differing levels of data compressibility. Our evaluation utilizes the $Tcomp32$ algorithm to compress the Rovio dataset while adjusting the arrival pattern of tuples, and a synthetic dataset, Micro, for evaluating the impact of data compressibility.

5.5.1 Arrival Pattern

Arrival patterns of tuples can significantly affect the performance of stream compression algorithms. By manipulating the order of timestamps in the Rovio dataset, we are able to modify the tuple arrival characteristics while keeping other parameters constant.

Impacts of arrival rate. The impact of varying arrival rates is depicted in Figure 14a. Here, the arrival rate of tuples varies from 500 to 10^6 per second. When the arrival rate is low, the hardware is underutilized, resulting in increased processing latency. This underutilization is due to more cycles of periodical DDR4 refreshing [63], leading to increased overhead. Conversely, when the arrival rate is high, processing latency increases due to resource conflicts, such as cache misses.

Impacts of arrival skewness. Figure 14b shows the impact of varying levels of arrival skewness on the latency of $Tcomp32$. Arrival skewness is varied by adjusting the Zipf factor from 0 to 1. Increased skewness leads to higher latencies as both under-utilization and overloading cases become more prevalent.

5.5.2 Data Compressibility

We further assess the impact of data compressibility on the performance of both stateless and stateful compression algorithms using the Micro dataset. The stateless $Tcomp32$ and stateful $Tdic32$ compression algorithms are chosen for this analysis.

Impacts of stateless compressibility. Stateless compressibility can be exploited by analyzing a single piece of input data without referring to a state. Figure 15a and 15b show the throughput and compression ratio respectively for varying levels of stateless compressibility. For the stateless $Tcomp32$, as the dynamic range increases, it becomes more costly and less compressible. However, $Tdic32$ exhibits a "cliff effect" at around 2^{12}, corresponding to its dictionary entries. Before this threshold, $Tdic32$ achieves higher compressibility and lower latency, after which the compressibility becomes nearly constant.

Impacts of stateful compressibility. Stateful compressibility, on the other hand, can be detected by referencing the history of the compression process. Figure 16a and 16b show the throughput and compression ratio respectively for varying levels of stateful compressibility. As the duplication ratio increases, the stateful $Tdic32$ experiences higher throughput and a higher compression ratio due to less frequent state updates and fewer bits of compressed data output. In comparison, the stateless $Tcomp32$ is largely unaffected by changes in stateful compressibility.

5.6 Summary

We have three key finding as summarized below according to our evaluation results.

Firstly, there is no single stream compression algorithm that can always outperform others. On the one hand, the

applicable stream compression algorithms have different and complex impacts on efficiency and cost of stream compression, and there are trade-offs during the selection. For instance, we can use more costly stateful stream compression to improve compression ratio, or the more lightweight stateful stream compression to achieve lower latency under lower compression ratio. On the other hand, the workload with different structure pattern and stateless/stateful compressibility can make the optimal selection different. Moreover, the different arrival pattern of even the same workload can affect the compression cost a lot. Despite the complex relationship, we observe the lossy stream compression can always achieve highest compression ratio for all real-world scenarios with marginal information loss.

Secondly, the parallelization of stream compression requires a careful design to reduce cost. Specifically, we should consider both hardware utilization and communication overhead when determine the granularity of partition and communication. Moreover, a cache-aware agglomeration is essential to reduce cost and increase parallelism, and there would be $11x$ more penalty without using it.

Thirdly, it's highly recommend to conduct stream compression on asymmetric 64-bit RISC edge processor, as such new hardware tendency can deliver up to 59% latency reduction and up to 69% energy consumption reduction in stream compression compared with legacy hardware. Besides, the frequency and cores regulation allow us to further tread-off time and energy in conducting stream compression.

Considering the findings above, we accordingly suggest a stream compression system for IoT analytic should be able to 1) adapt to the IoT workload and dynamically choose the optimal compression approach; 2) wisely determine the parallelization granularity and use cache-aware agglomeration; and 3) be specially aware to the asymmetric 64-bit RISC edge processor, and wisely regulates its frequency and cores on the fly under different user demands.

6 Related Work

Existing work in data compression, parallel data compression, and the use of asymmetric architecture at the edge, has laid a strong foundation for our study. However, the focus of these works does not entirely address the unique challenges and requirements of edge computing in the context of IoT. Our work, CStream, harnesses the insights of these foundational studies [12], [64], [65], [35], [66], [67], [15], [68], [69], [13] while extending and adapting them to address the unique needs of stream compression at the edge.

Experimental Study of Data Compression Algorithms. Historically, data compression studies [12], [64], [65], [35], [66] have investigated various methods for reducing database sizes or for specific applications such as IoT [67], [15], [68], [69], [13]. These studies provide valuable insights, but their effectiveness in guiding efficient stream compression at the edge is limited. The primary reason is that these works do not specifically consider the continuous arrival of data streams, a characteristic that significantly affects the dynamics of compression at the edge. Furthermore, they do not fully cover all possible data cases and often overlook energy impacts, which are a major concern in edge computing. CStream differs by considering the continuous data stream arrival, comprehensively covering different data cases, and prioritizing energy consumption as a key performance metric.

Parallel Data Compression. While numerous studies [70], [71], [72], [73], [74], [75], [76], [77], [78], [79], [80], [38], [1], [81], [82], [83], [24], [84], [85] have explored the parallelization of data compression algorithms, their focus has mostly been on compressing static data across various hardware platforms. However, these works do not primarily concern energy efficiency and stream compression at the edge. They also tend to focus on heavy-weight stateful compression methods. In contrast, CStream addresses these gaps by focusing on both stateless and stateful compression algorithms' pros and cons and emphasizes the incremental nature of data streams in edge computing scenarios.

Utilizing Asymmetric Architecture at the Edge. Asymmetric architecture [40], [60], [86], [62], [47], [87], [88], [61], [89], [90] has proven beneficial for delivering high performance with energy efficiency, a critical requirement for edge computing. While valuable contributions have been made in understanding and managing asymmetry in workload scheduling [61], [91], [89], none of these works have specifically explored stream compression tasks. CStream is designed to fill this gap, exploiting the fine-grained behavior and complex selection of stream compression algorithms and the workload's impact on asymmetric architecture utilization.

7 Conclusion

In this paper, we introduced `CStream`, an innovative framework tailored for optimizing stream compression on IoT edge devices. Through the strategic integration of various stream compression algorithms and asymmetric multicore processors, `CStream` capably navigates the challenges of achieving high compression ratios, increasing throughput, minimizing latency, and reducing energy consumption. Our comprehensive evaluations highlight the exceptional performance of `CStream`. It delivers a 2.8x compression ratio with minimal information loss, a 4.3x increase in throughput, and a substantial reduction in both latency (65%) and energy consumption (89%) compared to traditional designs. `CStream`, with its adaptable nature and co-design strategy, sets a new benchmark in IoT edge computing. It not only addresses the immediate demands of stream compression but also opens the way for future advancements, proving that a well-integrated software-hardware design approach can yield remarkable results.

References

[1] G. Pekhimenko, C. Guo, M. Jeon, P. Huang, and L. Zhou, "Tersecades: Efficient data compression in stream processing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 307–320. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/pekhimenko

[2] P. R. Geethakumari *et al.*, "Streamzip: Compressed sliding-windows for stream aggregation," in *ICFPT*. IEEE, 2021.
[3] X. Zeng and S. Zhang, "A hardware-conscious stateful stream compression framework for iot applications (vision)," 2023.
[4] ——, "Parallelizing stream compression for iot applications on asymmetric multicores," in *2023 IEEE 39rd International Conference on Data Engineering (ICDE)*, 2023.
[5] L. et al., "Camel: Managing data for efficient stream learning," in *SIGMOD 2022*, 2022.
[6] S. Zeuch, A. Chaudhary, B. D. Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Breß, J. Traub, and V. Markl, "The nebulastream platform for data and application management in the internet of things," in *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf
[7] M. Bansal, I. Chana, and S. Clarke, "A survey on iot big data: Current status, 13 v's challenges, and future directions," vol. 53, no. 6, 2020. [Online]. Available: https://doi.org/10.1145/3419634
[8] (2021) Arm solutions for iot, https://www.arm.com/solutions/iot. Last Accessed: 2021-05-10.
[9] (2021) Rockchip wiki rk3399, http://opensource.rock-chips.com/wiki_RK3399. Last Accessed: 2021-05-10.
[10] (2013) Allwinner soc family, https://linux-sunxi.org/Allwinner_SoC_Family. Last Accessed: 2022-05-10.
[11] (2021) Intel atom® x5-z8350 processor, https://ark.intel.com/content/www/us/en/ark/products/93361/intel-atom-x5z8350-processor-2m-cache-up-to-1-92-ghz.html. Last Accessed: 2022-05-10.
[12] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 671–682.
[13] D. Blalock, S. Madden, and J. Guttag, "Sprintz: Time series compression for the internet of things," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, no. 3, pp. 1–23, 2018.
[14] (2017) zlib home page, http://www.zlib.net/. Last Accessed: 2021-06-29.
[15] T. Lu, W. Xia, X. Zou, and Q. Xia, "Adaptively compressing {IoT} data on the resource-constrained edge," in *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.
[16] (2005) Mit-bih database distribution,http://ecg.mit.edu/. Last Accessed: 2021-06-29.
[17] (2019) Creator of the angry birds game, www.rovio.com. Last Accessed: 2021-05-10.
[18] (2018) Shanghai stock exchange, http://english.sse.com.cn/. Last Accessed: 2021-11-12.
[19] (2021) Beach weather stations - automated sensors , https://catalog.data.gov/dataset/beach-weather-stations-automated-sensors/resource/3b820f68-4dca-4ea7-8141-f37d9237734d. Last Accessed: 2021-11-12.
[20] (2021) Tracetogether, safer together, https://www.tracetogether.gov.sg/. Last Accessed: 2021-11-07.
[21] D. R.-J. G.-J. Rydning *et al.*, "The digitization of the world from edge to core," *Framingham: International Data Corporation*, vol. 16, 2018.
[22] S. Zhang, J. He, A. C. Zhou, and B. He, "Briskstream: Scaling data stream processing on shared-memory multicore architectures," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 705–722.
[23] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 4, pp. 1–39, 2009.
[24] A. Ukil, S. Bandyopadhyay, and A. Pal, "Iot data compression: Sensor-agnostic approach," in *2015 data compression conference*. IEEE, 2015, pp. 303–312.
[25] (2020) Eclipse iot working group. iot developer survey 2018. [Online]. Available: https://https://blogs.eclipse.org/post/benjamin-cab%C3%A9/key-trends-iotdeveloper-survey-2018,2018.
[26] (2020) Dalvik executable format in android, https://source.android.com/devices/tech/dalvik/dex-format.html. Last Accessed: 2022-05-29.
[27] C. Baskin, N. Liss, E. Schwartz, E. Zheltonozhskii, R. Giryes, A. M. Bronstein, and A. Mendelson, "Uniq: Uniform noise injection for non-uniform quantization of neural networks," *ACM Transactions on Computer Systems (TOCS)*, vol. 37, no. 1-4, pp. 1–15, 2021.
[28] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 194–203, 1975.
[29] (2021) lz4 source code, https://github.com/lz4/lz4/. Last Accessed: 2021-07-25.
[30] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An experimental study of bitmap compression vs. inverted list compression," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 993–1008.
[31] Y. Zhou, Z. Vagena, and J. Haustad, "Dissemination of models over time-varying data," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 864–875, 2011.
[32] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, 1952.
[33] A. Gupta *et al.*, "Modern lossless compression techniques: Review, comparison and analysis," in *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. IEEE, 2017.
[34] A. Moffat, "Huffman coding," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–35, 2019.
[35] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An experimental study of bitmap compression vs. inverted list compression," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 993–1008.
[36] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 377–388.
[37] (2021) Up board series, https://up-board.org/up/specifications/. Last Accessed: 2021-05-10.
[38] Y. Dua, V. Kumar, and R. S. Singh, "Parallel lossless hsi compression based on rls filter," *Journal of Parallel and Distributed Computing*, vol. 150, pp. 60–68, 2021.
[39] V. Pankratius, A. Jannesari, and W. F. Tichy, "Parallelizing bzip2: A case study in multicore software engineering," *IEEE software*, vol. 26, no. 6, pp. 70–77, 2009.
[40] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–38, 2016.
[41] W. Zhang, Z. He, L. Liu, Z. Jia, Y. Liu, M. Gruteser, D. Raychaudhuri, and Y. Zhang, "Elf: accelerate high-resolution mobile deep vision with content-aware parallel offloading," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, 2021, pp. 201–214.
[42] Z. Pan, F. Zhang, Y. Zhou, J. Zhai, X. Shen, O. Mutlu, and X. Du, "Exploring data analytics without decompression on embedded gpu systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 7, pp. 1553–1568, 2021.
[43] (2022) Dra829j-q1 asymmetric processor, https://www.ti.com/product/DRA829J-Q1. Last Accessed: 2022-05-10.
[44] S. Yang, R. A. Shafik, G. V. Merrett, E. Stott, J. M. Levine, J. Davis, and B. M. Al-Hashimi, "Adaptive energy minimization of embedded heterogeneous systems using regression-based learning," in *2015 25th international workshop on power and timing modeling, optimization and simulation (PATMOS)*. IEEE, 2015, pp. 103–110.
[45] W. Wolff and B. Porter, "Performance optimization on big.little architectures: A memory-latency aware approach," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 51–61. [Online]. Available: https://doi.org/10.1145/3372799.3394370
[46] H. Ribic and Y. D. Liu, "Energy-efficient work-stealing language runtimes," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, p. 513–528, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2654822.2541971
[47] T. Somu Muthukaruppan, A. Pathania, and T. Mitra, "Price theory based power management for heterogeneous multi-cores," ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 161–176. [Online]. Available: https://doi.org/10.1145/2541940.2541974

[48] S. Yamagiwa, E. Hayakawa, and K. Marumo, "Adaptive entropy coding method for stream-based lossless data compression," ser. CF '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 265–268. [Online]. Available: https://doi.org/10.1145/3387902.3394037

[49] S. Yamagiwa, R. Morita, and K. Marumo, "Bank select method for reducing symbol search operations on stream-based lossless data compression," in *2019 Data Compression Conference (DCC)*. IEEE, 2019, pp. 611–611.

[50] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[51] (2021) Ina226, https://www.ti.com/product/INA226. Last Accessed: 2021-11-12.

[52] (2021) esp32s2, https://www.espressif.com/en/products/socs/esp32-s2. Last Accessed: 2021-11-12.

[53] (2016) Intel® 64 and ia-32 architectures software developer's manual, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf. Last Accessed: 2021-03-12.

[54] S. Zhang, Y. Mao, J. He, P. M. Grulich, S. Zeuch, B. He, R. T. Ma, and V. Markl, "Parallelizing intra-window join on multicores: An experimental study," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2089–2101.

[55] (2021) Rock pi 4 wiki, https://wiki.radxa.com/Rockpi4. Last Accessed: 2021-05-10.

[56] (2021) Getting started with p2-zero, https://wiki.banana-pi.org/Getting_Started_with_P2-Zero. Last Accessed: 2021-05-10.

[57] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[58] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, "Roofline model toolkit: A practical tool for architectural and program analysis," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014, pp. 129–148.

[59] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 506–517.

[60] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 2013, pp. 1–10.

[61] T. Yu, R. Zhong, V. Janjic, P. Petoumenos, J. Zhai, H. Leather, and J. Thomson, "Collaborative heterogeneity-aware os scheduler for asymmetric multicore processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1224–1237, 2020.

[62] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 213–224.

[63] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez, "Understanding and mitigating refresh overheads in high-density ddr4 dram systems," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 48–59, 2013.

[64] M. A. Roth and S. J. Van Horn, "Database compression," *ACM sigmod record*, vol. 22, no. 3, pp. 31–39, 1993.

[65] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte, "The implementation and performance of compressed databases," *ACM Sigmod Record*, vol. 29, no. 3, pp. 55–67, 2000.

[66] P. Damme, A. Ungethüm, J. Hildebrandt, D. Habich, and W. Lehner, "From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 44, no. 3, pp. 1–46, 2019.

[67] K. Iqbal, N. Khan, and M. G. Martini, "Performance comparison of lossless compression strategies for dynamic vision sensor data," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 4427–4431.

[68] M. Bharde, S. Bhattacharya, D. D. Shree *et al.*, "{Store-Edge}{RippleStream}: Versatile infrastructure for {IoT} data transfer," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[69] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm, "A time-series compression technique and its application to the smart grid," *The VLDB Journal*, vol. 24, no. 2, pp. 193–218, 2015.

[70] M. Milward, J. L. Nunez, and D. Mulvaney, "Design and implementation of a lossless parallel high-speed data compression system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 481–490, 2004.

[71] K. Sano, K. Katahira, and S. Yamamoto, "Segment-parallel predictor for fpga-based hardware compressor and decompressor of floating-point data streams to enhance memory i/o bandwidth," in *2010 Data Compression Conference*. IEEE, 2010, pp. 416–425.

[72] J. Tian, S. Di, C. Zhang, X. Liang, S. Jin, D. Cheng, D. Tao, and F. Cappello, "Wavesz: A hardware-algorithm co-design of efficient lossy compression for scientific data," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 74–88.

[73] M. Bark, S. Ubik, and P. Kubalik, "Lz4 compression algorithm on fpga," in *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. IEEE, 2015, pp. 179–182.

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

[75] A. Ozsoy and M. Swany, "Culzss: Lzss lossless data compression on cuda," in *2011 IEEE International Conference on Cluster Computing*. IEEE, 2011, pp. 403–411.

[76] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher, "Mpc: a massively parallel compression algorithm for scientific data," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 381–389.

[77] Y. Huang, Y. Li, Z. Zhang, and R. W. Liu, "Gpu-accelerated compression and visualization of large-scale vessel trajectories in maritime iot industries," *IEEE Internet of Things Journal*, vol. 7, no. 11, pp. 10 794–10 812, 2020.

[78] M. Burtscher and P. Ratanaworabhan, "pfpc: A parallel compressor for floating-point data," in *2009 Data Compression Conference*. IEEE, 2009, pp. 43–52.

[79] J. Shun and F. Zhao, "Practical parallel lempel-ziv factorization," in *2013 Data Compression Conference*. IEEE, 2013, pp. 123–132.

[80] F. Knorr, P. Thoman, and T. Fahringer, "ndzip: A high-throughput parallel lossless compressor for scientific data," in *2021 Data Compression Conference (DCC)*. IEEE, 2021, pp. 103–112.

[81] T. Ma, M. Hempel, D. Peng, and H. Sharif, "A survey of energy-efficient compression and communication techniques for multimedia in resource constrained systems," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 3, pp. 963–972, 2012.

[82] D. Zordan, B. Martinez, I. Vilajosana, and M. Rossi, "On the performance of lossy compression schemes for energy constrained sensor networking," *ACM Transactions on Sensor Networks (TOSN)*, vol. 11, no. 1, pp. 1–34, 2014.

[83] C. J. Deepu, C.-H. Heng, and Y. Lian, "A hybrid data compression scheme for power reduction in wireless sensors for iot," *IEEE transactions on biomedical circuits and systems*, vol. 11, no. 2, pp. 245–254, 2016.

[84] D.-U. Lee, H. Kim, M. Rahimi, D. Estrin, and J. D. Villasenor, "Energy-efficient image compression for resource-constrained platforms," *IEEE Transactions on Image Processing*, vol. 18, no. 9, pp. 2100–2113, 2009.

[85] H. Zhang, X. Chen, N. Xiao, and F. Liu, "Architecting energy-efficient stt-ram based register file on gpgpus via delta compression," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[86] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann, "Caloree: Learning control for predictable latency and low energy," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 184–198, 2018.

[87] B. Salami, H. Noori, and M. Naghibzadeh, "Fairness-aware energy efficient scheduling on heterogeneous multi-core processors," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 72–82, 2020.

[88] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, "Exploiting heterogeneity for tail latency and energy efficiency," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 625–638.

[89] M. Wang, S. Ding, T. Cao, Y. Liu, and F. Xu, "Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus,"

in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, 2021, pp. 215–228.

[90] Q. Zeng, Y. Du, K. Huang, and K. K. Leung, "Energy-efficient resource management for federated edge learning with cpu-gpu heterogeneous computing," *IEEE Transactions on Wireless Communications*, vol. 20, no. 12, pp. 7947–7962, 2021.

[91] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 13–24.