CStream: Parallel Data Stream Compression on Multicore Edge Devices

Xianzhi Zeng and 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 $2.8\times$ compression ratio with only marginal information loss, $4.3\times$ throughput, 65% latency reduction and 89% energy consumption reduction, compared to designs lacking such strategic integration.**

***Index Terms*—Asymmetric hardware, edge computing and IoT, stream compression.**

I. Introduction

DATA stream compression, has become a pivotal research problem [1], [2]. Fig. 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 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 Fig. 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 has been well studied in the literature [7], 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 with resource constraints?*

This paper presents `CStream`, a framework for parallelizing stream compression on multicore edge devices. `CStream` explores the design space, balancing compression ratios, speeds, and energy consumption. It employs a software-hardware co-design to improve stream compression, referencing [8], [9]. `CStream` supports various stream compression algorithms, from traditional lossless to advanced lossy methods, including stateful, stateless, and byte-aligned options. It adapts to hardware differences, optimizing for multicore processors, RISC or CISC architectures, and various core counts. This ensures efficient use of hardware resources, enhancing speed while reducing energy use. Furthermore, `CStream` introduces parallelization strategies, some new, covering execution, state management, and scheduling. These strategies allow for effective workload distribution across cores, improving throughput and lowering 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,

Manuscript received 28 June 2023; revised 2 April 2024; accepted 6 April 2024. Date of publication 19 April 2024; date of current version 27 September 2024. This work was supported by the MoE AcRF Tier 2 under Grant MOE-T2EP20122-0010, in part by National Research Foundation, Singapore and Infocomm Media Development Authority under its Future Communications Research and Development Programme under Grant FCP-SUTD-RG-2022-005, and in part by NTU under Grant 023452-00001. Recommended for acceptance by K. Zheng. *(Corresponding author:Shuhao Zhang.)*

Xianzhi Zeng is with the Singapore University of Technology and Design,, Singapore 487372 (e-mail: xianzhi_xianzhi@mymail.sutd.edu.sg).

Shuhao Zhang is with the Nanyang Technological University, Singapore 639798 (e-mail: shuhao.zhang@ntu.edu.sg).

Digital Object Identifier 10.1109/TKDE.2024.3386862

1041-4347 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.

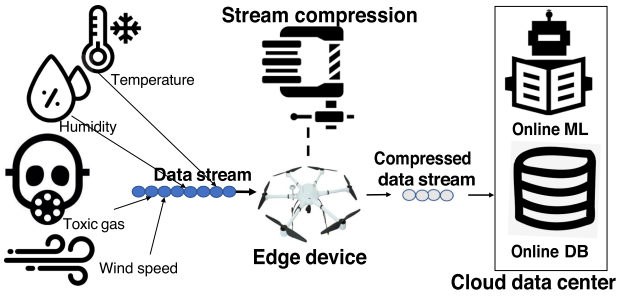


Fig. 1. Real-time data gathering conducted by a patrol drone in environments inaccessible to humans.

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 [10], H2+ [11], and Z8350 [12]). 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 stream compression.

Organization: The paper is structured as follows: Section II provides an overview of stream compression and the challenges of implementing it at the edge. Section III explores the co-design aspects of stream compression, detailing CStream's architecture and implementation. Section IV outlines the methodology for evaluating CStream. Section V presents the results of our experimental evaluation, focusing on CStream's performance. Section VI reviews related work, showing how CStream progresses beyond existing approaches in stream compression for multicore edge devices. Section VII concludes the paper, summarizing CStream's contributions and suggesting future research directions.

II. Background

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

A. 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 [13], time series compression [14], and file compression [15], where all data is ready before conducting the compression.

Not all compression methods are fit for the unique demands of stream data–incremental, infinite, and high-volume. For instance, LZ77/78 algorithms [16], despite their efficiency, fall short in stream compression due to their need for future data and high processing load. We focus on algorithms like LEB128 and ADPCM, which are notable for their low processing needs and versatility across data types, from media streams to sensor data. This selection extends to techniques originating from specialized areas–media streams [17], file systems [18], and sensor communications [14], [19]–proven effective for stream compression. Our study evaluates a curated set of algorithms (Section III-A), including LEB128-NUQ, ADPCM, among others, selected for their ability to compress data streams efficiently, addressing the critical need for high throughput and integrity.

B. Stream Compression at the Edge

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 [6], [20]. The source(s) of these data streams could be singular (e.g., an ECG sensor [21]) or multiple (e.g., distributed game servers [22]). Additionally, the tuple content may be a plain value (e.g., unsigned integer [21]), binary structured (e.g., the $< key, value >$ pair [23]), or textual structured (e.g., in XML format [24]). The arrival pattern and compressibility of the data stream can also greatly vary. For instance, an anti-Covid19 tracking system [25] at a mall may generate data streams more

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.

densely during peak hours. Due to this versatility, efficient stream compression is highly context-dependent.

Performance Demand: Given the *13 V's challenges [8]* 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 [26], 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 [1], [27], [28], but also at the edge. The high-rate data streams at the edge, collected from massive device links [6], [29], 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 [30]. 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.

III. 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 asymmetry-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`.

A. 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 [34]. These algorithms typically target improvements in theoretical compressibility or reductions in compression overhead [35], [36]. In developing `CStream`, we selected a variety of compression algorithms (detailed in Table I) to meet the varied needs. These algorithms are well-suited for edge environments, where computational resources are limited. LEB128 is chosen for its simplicity and efficiency, especially suitable for streaming data due to its byte-aligned and stateless nature. Tcomp32, a variant of LEB128 with a unique encoding format in Algorithm 1's $s1$, provides flexibility with byte-unaligned outputs. Additionally, algorithms like ADPCM and PLA offer lossy compression with minimal data loss, vital for achieving higher compression ratios. Each algorithm contributes distinct qualities, such as statefulness, byte alignment, and fidelity, enabling `CStream` to accommodate a broad range of data types and compression requirements.

TABLE I
Summary of Studied Compression Algorithms

Algorithm Name	Fidelity	State utilization	Byte alignment
LEB128-NUQ [18], [31]	lossy	stateless	aligned
ADPCM [18], [31], [1]	lossy	stateful, value	aligned
UANUQ [31], [32]	lossy	stateless	unaligned
UAADPCM [32], [31], [1]	lossy	stateful, value	unaligned
LEB128 [18]	lossless	stateless	aligned
Delta-LEB128 [18], [1]	lossless	stateful, value	aligned
Tcomp32 [32]	lossless	stateless	unaligned
Tdic32 [33], [32]	lossless	stateful, dictionary	unaligned
RLE [17]	lossless	stateful, value	aligned
PLA [19], [14]	lossy	stateful, model	aligned

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 guarantees exact original data reconstruction. `CStream` compacts data losslessly within Shannon's entropy theoretical limits [36]. One example is the $Tcomp32$ algorithm [32] used in `CStream`'s lossless mode. It employs a lossless stream compression technique that suppresses leading zeros [32], a form of bit-level null suppression [17], 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 reduces fidelity for increased compression ratios. `CStream` discards non-critical data, with compression limits tied to the chosen fidelity level. For example, using 8-bit unaligned non-uniform quantization (UANUQ [31], [32]) yields higher compression but more data loss than a 12-bit approach. `CStream`'s lossy mode balances compression and fidelity, meeting diverse application needs at the edge.

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

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: LEB128 Algorithm.

Input: input stream $inData$
Output: output stream $outData$
while *not reach the end of* $inData$ **do**
  /* (s0) */
  $number \leftarrow$ read next 32-bit from $inData$;
  /* (s1) */
  $zeros \leftarrow$ counting the leading zeros in $number$;
  $encoded \leftarrow$ squeeze $zeros$ in $number$ under LEB128 format;
  /* (s2) */
  write $encoded$ to $outData$;
end

Algorithm 2: CStream's Stateful Compression Mode.

Input: input stream $inData$
Output: output stream $outData$
while $inData$ *is not stopped* **do**
  (s0) load the tuple(s) from $inData$;
  (s1) pre-process ;
  (s2) state update ;
  (s3) state-based encoding;
  (s4) output compressed data to $outData$;
end

Algorithm 3: Delta-LEB128 Algorithm.

Input: input stream $inData$
Output: output stream $outData$
state $\leftarrow 0$;
while *not reach the end of* $inData$ **do**
  /* (s0) */
  $number \leftarrow$ read next 32-bit from $inData$;
  /* (s1) */
  lastNumber $\leftarrow$ state ;
  /* (s2) */
  state $\leftarrow$ number ;
  /* (s3) */
  $x \leftarrow$ number-lastNumber ;
  $zeros \leftarrow$ counting the leading zeros in x;
  $encoded \leftarrow$ squeeze $zeros$ in x under LEB128 format;
  /* (s4) */
  write $encoded$ to $outData$;
end

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. It typically involves three steps. (1) load the tuple(s) from $inData$. (2) transform and find the *compressible* parts. (3) output compressed data to $outData$. 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. As a concrete example of stateless compression algorithm, we show the detailed implementation of Android-Dex's $LEB128$ [18] in Algorithm 1.

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 2 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 [1], [37] and run-length encoding (RLE) [17] that are considered "lightweight" for both stream and database compression [1], [17], 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. By ensuring that our dictionary-based state is sized to fit within the L1 cache [33], we can significantly speed up the process of searching and matching dictionary entries with incoming data.
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 [14], [19].

As a concrete example of stateful compression, we add the delta-encoding (i.e., value-based state) to Algorithm 1, and demonstrate the $Delta - LEB128$ in Algorithm 3.

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 [18]. 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 (e.g., 223872 b can be wasted for a batch of 7.4 Megabits).

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.

4) Compression Implementation: Our implementation strategy for CStream is grounded in a step-by-step refinement process, which not only underscores the adaptability of the framework but also demonstrates its potential for integrating a broad spectrum of compression algorithms. This incremental approach, depicted in Fig. 2, allows for a nuanced analysis of each

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.

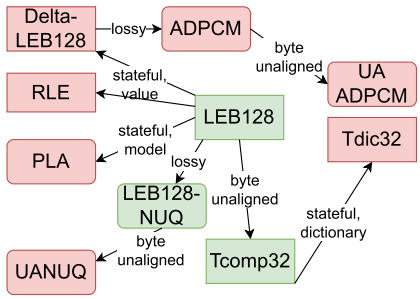


Fig. 2. Incremental relationship among algorithms.

algorithm's impact, fostering a comprehensive understanding of their individual and collective contributions to the framework.

We begin with the implementation of the LEB128 algorithm, mirroring its use in Android-Dex [18]. This foundational stateless and byte-aligned stream compression algorithm sets the stage for subsequent adaptations and enhancements. Building on this, we evolve LEB128 into $Tcomp32$ by simplifying Elias encoding [32], thus accommodating byte-unaligned output. This transformation marks our framework's initial foray into optimizing compression techniques for more diverse data structures. Further diversifying our approach, we integrate lossy compression through $LEB128 - NUQ$, achieved by altering LEB128's encoding to non-uniform quantization [31]. This extension paves the way for the $UANUQ$ variant, enhancing our framework's capability to handle byte-unaligned output efficiently.

In addition to these stateless modifications, we enrich $LEB128$ and $Tcomp32$ with stateful compression features. $Tdic32$ employs a LZ4-like hash table [33], utilizing a dictionary-based state for tracking and compressing data, while Delta-LEB128 calculates a 'delta' state [1], [37], representing the difference between consecutive values. This variation is indicative of our framework's flexibility in accommodating both dictionary and value-based states. Moreover, we explore model-based stateful compression through the integration of PLA [14], [19], which utilizes piecewise linear approximation, further expanding the scope of `CStream`'s capabilities. The framework's adaptability is further exemplified in the implementation of Adaptive Differential Pulse-Code Modulation (ADPCM). Here, we apply lossy non-uniform quantization [31] to the 'delta' state of Delta-LEB128, transitioning from lossless to lossy compression. This method, along with its byte-unaligned variant $UAADPCM$, highlights `CStream`'s capacity to incorporate and optimize a wide array of compression techniques, demonstrating its suitability for diverse IoT edge computing environments.

In essence, the iterative development of these algorithms in our `CStream` showcases our dedication to a flexible framework. It also sets the foundation for future integration of advanced compression methods, addressing the evolving requirements of IoT edge computing.

B. 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 examined hardware are detailed in Table II.

TABLE II
MULTICORE EDGE PROCESSORS STUDIED

Processor Name	Processor Architectures	Byte Length	ISA Pattern	Frequency Range/GHz	Core Number
$H2+$ [11]	SMP	32 bit	RISC	$0.416 \sim 1.2$	4
$RK3399^{AMP}$ [10]	AMP	64 bit	RISC	$0.416 \sim 1.8$	6
$RK3399^{SMP}$ [10]	SMP	64 bit	RISC	$0.416 \sim 1.8$	6
$Z8350$ [12]	SMP	64 bit	CISC	$0.8 \sim 1.92$	4

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 [38] supported by `CStream` employs a 4-core SMP Z8350 [12]. 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 [39], [40].

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) [41]. However, it acknowledges the potential of other forms of asymmetry, such as edge CPU+GPU [42], [43], CPU+DSP [44], and CPU+FPGA [45], as promising areas for future exploration.

2) Instruction Set Architecture: `CStream`'s flexibility shines through its support for a variety of ISAs.

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 [30].

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

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.

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 [46], [47], [48]. 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 [27], [28] 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.

C. 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.

1) 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 [49], [50]. 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* [1], [51]. 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.

2) 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) 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 [40]. This method takes advantage of the symmetry inherent in SMPs, where all cores have equal computational capacity and communication distance.

Asymmetry-aware Scheduling: For Asymmetric Multicore Processors (AMPs), which have cores with different computational abilities and communication distances, CStream employs Asymmetric-aware Scheduling [4] with two key designs: 1) fine-grained decomposition, which decomposes a stream compression procedure into multiple fine-grained tasks to better expose the task-core affinities under the asymmetric computation effects; and 2) asymmetry-aware task scheduling, which schedules the decomposed tasks based on a novel cost model to exploit the exposed task-core affinities while considering asymmetric communication effects. This method represents a refined adaptation to general challenges in energy-efficient scheduling [52]. Unlike conventional strategies that depend on black-box energy and metric predictions, this approach integrates an accurate white-box model. As elucidated in [4], it precisely models the computational and memory demands of stream compression tasks in piece-wise linear terms. These are then aggregated into arithmetic formulations to accurately predict final energy usage and latency.

D. Summary of Tuning Identified Trade-Offs.

CStream orchestrates a co-design space featuring a spectrum of tuning mechanisms across four principal domains. Initially, it incorporates adjustable parameters for core utilization, peak frequencies, and dynamic voltage and frequency scaling (DVFS) tactics, rooted in insights from [53]. Furthermore, an innovative internal mechanism allows users to toggle between state sharing and private configurations, a novel application in stream compression informed by prior works on stateful concurrent stream processing [54]. The framework also presents sophisticated options for refining scheduling approaches and calibrating cost models. For user convenience, CStream is preset with optimized configurations from [4], [27], though it remains customizable to accommodate diverse scenarios. While our focus precludes an exhaustive exploration of all tunable facets of stream compression algorithms themselves (e.g., parameter and hyperparameter tuning in some lossy compression [14]), default settings are recommended for lossless compression techniques to maintain efficiency and accuracy, with lossy methods finely adjusted to achieve specified NRMSE objectives with minimal computational demand. The exploration of heuristic

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.

and dynamically adaptive tuning, potentially leveraging online reinforcement learning, is identified as a promising avenue for future research [55].

E. 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 to accurately measure energy consumption across various edge platforms. Technically, it achieves 16-bit resolution for $0\sim15$ V voltage and $0\sim4$ A current at 1K SPS sampling. This energy metering component consists of Texas Instrument's INA226 [56] chip, which acts as a sensor for current and voltage, and the Espressif's ESP32S2 [57] 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 [58] before and after an experimental run.

The integration of the INA226 chip and the ESP32S2 MCU in our energy metering system enables precise measurement of energy usage, ensuring accuracy from the overall system to the specificities of individual operations, and providing insight into the energy profile of the stream compression process. Designed with minimal overhead, the system uses the ESP32S2 MCU for data preprocessing and USB-2.0 for communication, avoiding disruption to stream compression and parallel execution tasks. It offers real-time feedback on energy consumption, facilitating dynamic adjustments in execution strategies, state management, and scheduling to enhance energy efficiency. The system's seamless integration with the broader `CStream` framework allows for synergistic operation with `CStream`'s software strategies and hardware configurations, driving optimizations and adaptations based on current energy data.

IV. 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.

A. Performance Metrics

We focus on a range of metrics, each providing insight into different aspects of its functionality to evaluate `CStream`. As discussed in Section II, we include *compression ratio*, *throughput*, *energy consumption*, and *end-to-end latency*. We report the average of these metrics, calculated over a substantial data volume to avoid fluctuations and ensure 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. Note that, we don't delve into specialized float-point compression or more complicated semantic-aware compression, and we use the raw unsigned 4-byte binary words to investigate $x[i], y[i]$, i.e., each of them varies from $0 \sim 2^{32} - 1$ and $\bar{x}$ is strictly larger than 0. A larger NRMSE generally indicates more information loss in edge applications, such as more system noise in 5 G communication [59], more distortion of images or videos [60], and less accurate readings gathered from IoT sensors [61], [62], [63]. The maximum tolerance of NRMSE depends on specific applications, and an NRMSE within 5% is considered to be acceptable for most applications in [59], [60], [61], [62], [63].

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 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 III-E.

B. Benchmark Workloads

In order to comprehensively evaluate `CStream` across a diverse range of IoT use cases (discussed in Section II-B), 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 III-A2). On the other hand, *stateful compressibility* points to the compressible space hidden in the context, 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 III and are detailed below:

1) *ECG [21]:* 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,

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.

TABLE III
DATASET STUDIED

Dataset Name	Source	Data Structure	Stateless Compressibility	Stateful Compressibility
ECG [21]	single	plain	high	high
Rovio [22]	multiple	binary structured	medium	medium
Sensor [24]	multiple	textual structured	low	high
Stock [23]	multiple	binary structured	low	medium
Stock-key [23]	multiple	plain	low	medium
Micro [64]	single	plain	adjustable	adjustable

TABLE IV
EVALUATED HARDWARE PLATFORMS

Model	Supported processor	Number of cores	Installed memory
Banana pi zero-m2 [66]	$H2+$	4	512MB
Rockpi 4a [65]	$RK3399$	2 big + 4 little	2GB
UP board [38]	$Z8350$	4	1GB
Jetson AGX kit [67]	$Orin$	12	64GB

unstructured reflection of a continuous physical process, it exhibits the highest levels of both stateless and stateful compressibility.

2) *Rovio [22]:* 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 stateless compressibility; second, different tuples may share the same key, which demonstrates stateful compressibility.
3) *Sensor [24]:* 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 stateful compressibility due to the repetition of several fixed XML patterns across different tuples.
4) *Stock [23] and **Stock-Key** [23]:* 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 [64], used to further tune stateless and stateful 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).

C. 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 IV. 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 [65] 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: 1.8 GHz for the larger cores (core4 to core5) and 1.416 GHz for the smaller cores (core0 to core3).

V. 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 III. We delve into five key areas:

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

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

A. 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 3(a). 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 1.5 J/MB, the optimal one is labeled 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

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.

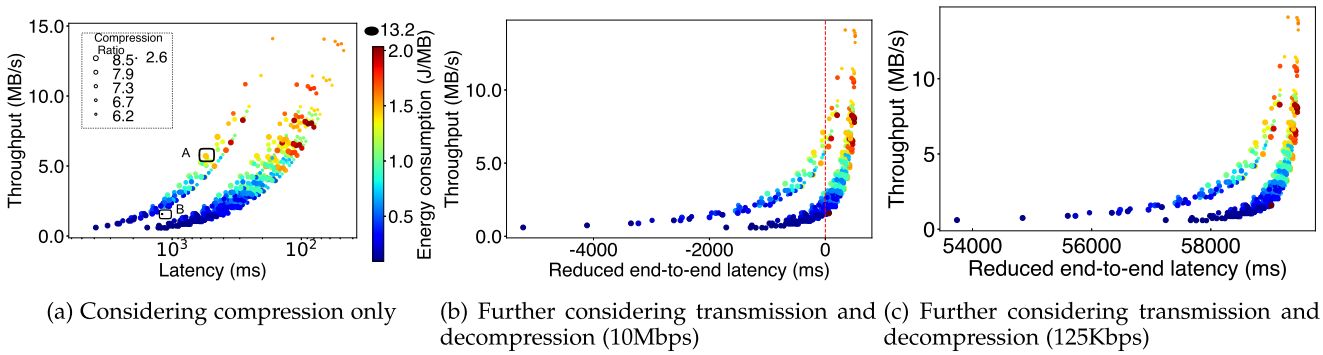


Fig. 3. Solution space of CStream $\leq 5\%$ NRMSE and RK3399 hardware.

workload in an Asymmetric-aware manner. Both cores execute the stream compression under a 8 KB micro-batch.

We also marked a careless solution point B in Fig. 3(a) 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.

To better understand the total data transfer time, which includes data compression, transmission, and decompression, we explore the effect on CStream's solution space when considering data transmission and decompression across varying network bandwidths. An RK3399 unit is used as the receiver and decompressor, operating in a single-threaded manner for simplicity. Future studies could see improvements by using more advanced processors and implementing parallel decompression. We examine two bandwidth scenarios: 10 Mbps and 125 Kbps, depicted in Fig. 3(b) and (c), respectively. The 10 Mbps setting mimics a situation with substantial bandwidth, similar to an 802.11n wireless channel, while 125 Kbps corresponds to a bandwidth-limited environment, typical of long-distance communications technologies like LoRa.

In scenarios with restricted bandwidth, such as 125 Kbps, stream compression significantly reduces the overall data transfer latency, with reductions of up to about 1 minute. However, in environments with higher bandwidth, like 10 Mbps, the application of stream compression requires careful consideration. A red dashed line in Fig. 3(b) highlights the point at which compression may begin to increase rather than decrease end-to-end latency. Techniques positioned to the left of this line could add processing time that surpasses the time saved from reduced data transmission, leading to an overall latency increase.

B. 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 I, on five real-world IoT datasets, as detailed in Table III.

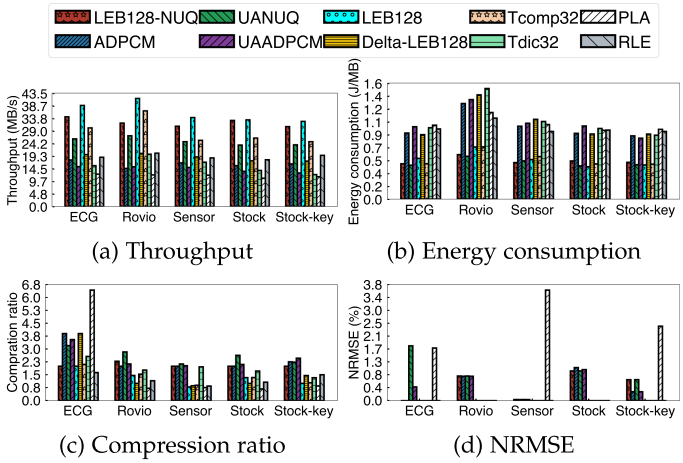


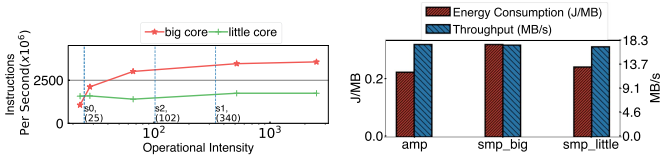
Fig. 4. Comparing ten algorithms on five datasets.

1) Fidelity. Lossless vs. Lossy Compression: Referencing Fig. 4(a), (b), (c), and (d), 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, 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%.

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.

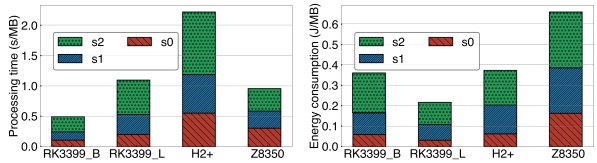
3) State Implementation. Value, Dictionary, and Model: Further comparisons among $ADPCM$ (stateful, value-based), $Tdic32$ (stateful, dictionary-based), and PLA (stateful, model-based) illustrate how different state implementations can impact compression performance. While $ADPCM$ consistently leads in throughput and energy consumption, $Tdic32$ and PLA shine

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.



(a) The roofline model on RK3399 processor (b) Energy consumption and throughput comparison

Fig. 5. Impacts of processor architectures for $Tcomp32$.



(a) Unit time cost (b) Unit energy cost

Fig. 6. Impacts of ISA for $Tcomp32$.

in handling structured data or data from multiple sources, offering higher compression ratios.

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 increases computational overhead due to the bit-by-bit encoding.

C. 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.

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 [68], [69] shows that one 1.416 GHz A72 big core in the RK3399 processor has about twice the computational power of one A53 little core at the same frequency (Fig. 5(a)). 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 1.416 GHz.
- *Symmetric multicore processor with only big cores* (smp_big): Using 2 A72 big cores at 1.416 GHz.
- *Symmetric multicore processor with only little cores* (smp_little): Using 4 A53 little cores at 1.416 GHz.

Each architectural case is tuned to its maximum throughput, after which we compare their energy consumption along with throughput (Fig. 5(b)). 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 ??) involve varying *operational intensities* [70], [71], [72], [73] (i.e., instructions per memory access), as shown by the dashed lines in Fig. 5(a). 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.

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 IV-C.

Specifically, we compare the time and energy cost per core for each step of $Tcomp32$ under different ISAs (Fig. 6(a) and (b)). 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.

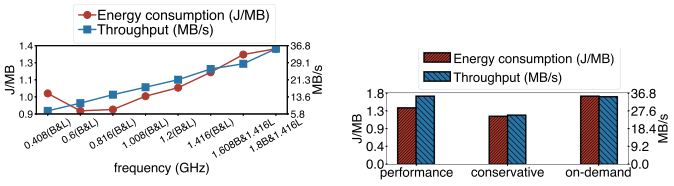
Our evaluation highlights two key findings. First, the RISC ISA (used in RK3399) demonstrates superior performance compared to the CISC ISA (used in Z8350). Both RK3399_B and RK3399_L cores have significantly lower unit energy costs than the Z8350, and the RK3399_B can even reduce the processing time 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.

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.

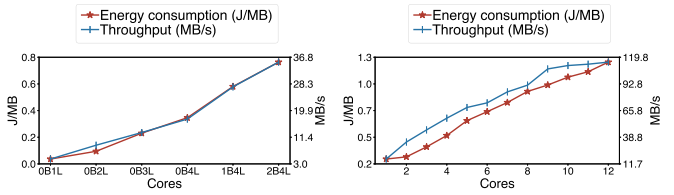
Statically Tuning the clock frequency: We first adjust the frequency of the big cores ('B') and the little cores ('L') on

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.



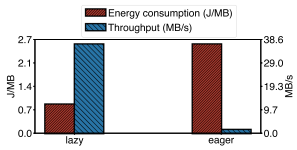
(a) Statically tuning (b) DVFS

Fig. 7. Impacts of frequency regulation for $Tcomp32$.



(a) On 6-core asymmetric multicores $RK3399$. (b) On 12-core symmetric multicores $Orin$.

Fig. 8. Scalability evaluation.



(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. 9. Impacts of execution strategy for $Tcomp32$.

the RK3399 processor. We observe how frequency changes influence the throughput and energy consumption, as shown in Fig. 7(a). 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 [46], [47], 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 0.816 GHz, 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 [46], [47], [48] to dynamically adjust the frequency. We use different DVFS strategies and present the results in Fig. 7(b). The "performance" strategy, which fixes each core at its highest frequency without dynamic reconfiguration (1.8 GHz for big cores and 1.416 GHz 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.

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 Fig 8. By enabling different numbers of big and little cores in Fig. 8(a), 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. Besides, similar scalability remains when we shift the focus to generally higher computational power as demonstrated in Fig. 8(b).

D. Parallelization Strategies

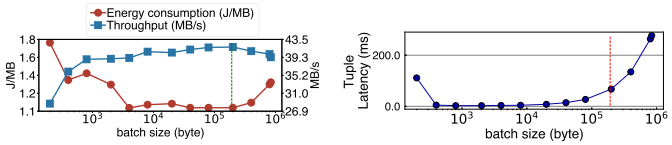
We now explore various parallelization strategies provided by `CStream`, including different execution strategies, micro-batch size, state sharing, and scheduling strategies. Our exploration, illustrated with the $Tcomp32$ algorithm and the Rovio dataset, illustrates the importance of each strategy and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

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, tuples are compressed immediately upon arrival, whereas lazy execution, the default in `CStream`, waits until a 400-byte micro-batch forms. These approaches don't alter the compression ratio but have a notable impact on throughput and energy consumption. Lazy execution is shown to be superior, improving throughput and reducing energy use, as depicted in Fig. 9(a). Eager execution, with its need for instant processing of each tuple, diminishes parallelism and hikes energy costs. The lazy method boosts parallelism by batching tasks before processing them. Analyzing the processing time split between 'blocked' and 'running' phases in Table 9b offers insight into the cost differences. 'Blocked' time, for resolving conflicts and reducing cache issues, and 'running' time, for actual compression, indicate that eager execution leads to more 'blocked' time. This results in higher overheads and lower energy efficiency compared to lazy execution.

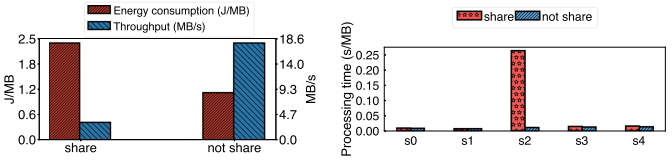
In our exploration of execution strategies within `CStream`, we delved into how varying batch sizes during lazy execution impact performance, leveraging the $Tcomp32$ algorithm and

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.



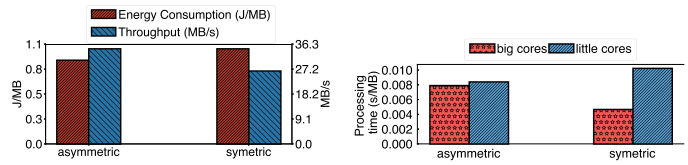
(a) Energy and throughput consumption (b) Average latency; The red dashed line is the total L1D size of all cores

Fig. 10. Impacts of batch sizes for $Tcomp32$.



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

Fig. 11. Impacts of state management strategy for $Tdic32$.



(a) Varying scheduling strategy (b) Time breakdown

Fig. 12. Impacts of scheduling strategy for $Tcomp32$.

the Rovio dataset. Our investigation ranged from batch sizes of hundreds to millions of bytes, with the effects on energy consumption and throughput depicted in Fig. 10(a). The study revealed an optimal batch size that minimizes energy consumption while maximizing throughput. Deviating from this optimal size, in either direction, incurs higher energy costs and diminishes throughput. This finding stresses the critical role of batch size optimization for efficient stream compression on edge devices. A pivotal insight from our analysis is the relationship between the optimal batch size and the combined L1D cache size of the processor cores, illustrated by a red dashed line in Fig. 10(a). This correlation suggests that micro-batching strategies should be informed by the hardware's L1D cache capacity to enhance both energy efficiency and throughput. Further investigation into latency implications across different batch sizes presented an inverted U-shaped pattern, as shown in Fig. 10(b). This pattern highlights a delicate balance between reducing latency and optimizing throughput and energy efficiency. The study underscores that exceeding the L1D cache size significantly increases latency, likely due to increased cache misses, as indicated by the red dashed line in Fig. 10(b). These findings advocate for a micro-batching approach in `CStream`'s lazy execution strategy that is aware of the L1D cache size. Such an approach not only helps in optimizing latency but also ensures improved throughput and energy efficiency. By adapting to the hardware's capabilities, `CStream` navigates the challenges of stream compression on edge devices, striking a balance between latency, throughput, and energy consumption with its innovative, cache-conscious design.

2) State Management Strategy. Shared vs Private: In `CStream`'s stateful stream compression, deciding between shared or private state management is crucial. We analyzed this using the $Tdic32$ algorithm and the Rovio dataset, exploring the implications of each approach. Shared state means all threads access a common dictionary, while private state assigns each thread its own dictionary, as detailed in Section III-A2. Fig. 11(a) shows the impact on energy consumption and throughput for both methods. Sharing states led to higher energy use and lower throughput. Yet, it slightly improved the compression ratio, from 1.78 in private to 1.81 in shared state scenarios. Given the minimal compression gain versus the significant overhead, the benefits of a mere 3% improvement in compression ratio often do not justify the costs involved.

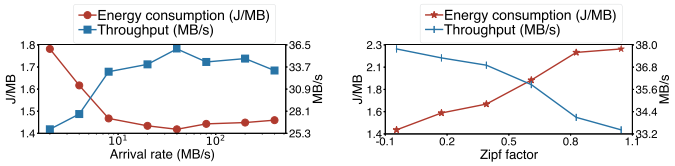
To understand the increased overhead from shared state management, we analyzed the processing time for each step in Algorithm 2 across two versions. This analysis, shown in Fig. 11(b), indicates the majority of extra cost comes from the state updating step ($s2$). Here, frequent locking hampers parallelism, with cores spending time on memory access–a less energy-intensive task than computations. This accounts for the slight increase in energy use compared to the more noticeable throughput reduction under shared state management. Our findings highlight the effectiveness of private state management in `CStream` for stateful stream compression, improving throughput and energy efficiency without greatly affecting the compression ratio.

3) Varying Scheduling Strategy: To identify the best scheduling strategy, we compared symmetric and asymmetric methods using the $Tcomp32$ algorithm and the Rovio dataset. Our focus was on energy consumption and throughput as shown in Fig. 12(a). Symmetric scheduling led to decreased throughput (by 26.2%) and increased energy consumption (by 13.4%) compared to asymmetric scheduling. The key issue with symmetric scheduling is its failure to account for hardware differences, as previously shown in Fig. 5(a). This results in underutilization of the computational capabilities of more powerful cores. For example, Fig. 12(b) illustrates the processing times for big and little cores during the $s1$ step of the $Tcomp32$ algorithm. Big cores are seen waiting for little cores under symmetric scheduling, causing inefficiencies that reduce energy efficiency and performance. Thus, asymmetric scheduling is recommended. It leverages the distinct computational strengths of different cores, assigning tasks in a way that boosts performance and reduces energy use. This approach outperforms symmetric scheduling in throughput and energy efficiency, proving its effectiveness in stream compression tasks on heterogeneous multi-core systems.

E. Workload Sensitivity Study

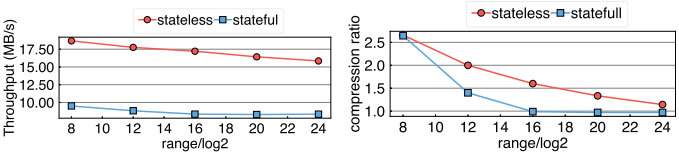
In this subsection, we illustrate the robustness of our framework, `CStream`, by conducting a comprehensive sensitivity analysis based on varying IoT data workloads. Our goal is to assess how `CStream` responds under diverse arrival patterns and differing levels of data compressibility. Our evaluation utilizes

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.



(a) Arrival rate (b) Arrival skewness

Fig. 13. Impacts of arrival pattern for $Tcomp32$.



(a) Throughput (b) Compression ratio

Fig. 14. Impacts of dynamic range for stateless ($Tcomp32$) and stateful stream compression ($Tdic32$).

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.

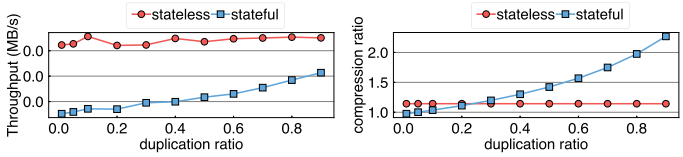
1) Arrival Pattern: Arrival patterns of tuples can significantly affect the performance of stream compression algorithms. By manipulating the order of timestamps in the Rovio dataset, we are able to modify the tuple arrival characteristics while keeping other parameters constant.

The impact of varying arrival rates is depicted in Fig. 13(a). 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 [74], leading to increased overhead. Conversely, when the arrival rate is high, processing latency increases due to resource conflicts, such as cache misses.

Fig. 13(b) 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.

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. Stateless compressibility can be exploited by analyzing a single piece of input data without referring to a state. Fig. 14(a) and (b) 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.

Stateful compressibility, on the other hand, can be detected by referencing the history of the compression process. Fig. 15(a)



(a) Throughput (b) Compression ratio

Fig. 15. Impacts of tuple duplication for stateless ($Tcomp32$) and stateful stream compression ($Tdic32$).

and (b) 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.

F. Summary

Our comprehensive investigation and evaluations have elucidated a number of key insights that underline the efficacy of our novel `CStream` framework, and are summarized in Table V. First, algorithm diversity in stream compression reveals no single algorithm reigns supreme; the choice hinges on application needs and workload nature. Lossy compression often delivers superior compression ratios with minimal integrity loss. `CStream`'s adaptability facilitates the integration of diverse algorithms to meet specific IoT application demands. Second, efficient parallelization is crucial. `CStream`'s strategic principles ensure optimal parallelization granularity and implement a cache-aware strategy, avoiding potential penalties up to 11x. Third, leveraging asymmetric 64-bit RISC edge processors offers notable latency and energy efficiency benefits. `CStream`'s design aligns with these processors, optimizing resource use through frequency and core regulation. In summary, effective IoT stream compression systems must adapt to workloads, select optimal compression methods, fine-tune parallelization, and efficiently manage processor resources. `CStream` excels in these aspects, redefining stream compression standards for IoT analytics.

VI. Related Work

Our work, `CStream`, draws on and extends the significant contributions made by previous studies in the domains of data compression, parallel data compression, and asymmetric architecture utilization at the edge [13], [14], [17], [20], [75], [76], [77], [78], [79], [80]. While these works lay a robust groundwork, they do not entirely cater to the distinct demands of stream compression in the edge computing context intrinsic to IoT. Our framework, `CStream`, builds upon these fundamental studies and refines them to fulfill the distinct requirements of edge-based stream compression.

Data Compression Algorithm Studies: Previous research [13], [14], [17], [20], [75], [76], [77], [78], [79], [80] on data compression, focused on database size reduction and

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.

TABLE V
COMPARATIVE SUMMARY OF TRADITIONAL METHODS AND CStream FRAMEWORK

Aspect	Traditional Methods	CStream Framework	Improvement
Compression Ratio	Lower, varies by algorithm	Up to 2.8x higher	Significantly higher compression with minimal information loss
Throughput	Constrained by hardware & algorithm	Up to 4.3x increase	Markedly higher processing speed
Latency	Higher due to inefficient processing	Reduced by 65%	Substantial decrease in data processing time
Energy Consumption	Higher, especially on symmetric processors	Reduced by 89%	Drastically lower, especially on asymmetric multicore processors
Hardware Utilization	Sub-optimal on asymmetric processors	Optimal use of asymmetric processors	Enhanced efficiency and performance
Parallelization Efficiency	Limited by static approaches	Dynamic, cache-aware strategies	Improved throughput and reduced costs

application-specific strategies, often overlooks the nuances of stream compression at the edge, such as continuous data arrival and energy efficiency, vital in edge computing. Recent work on Piecewise Linear Approximation (PLA) for edge computing marks a significant advance, addressing PLA's application to numerical timestamped streams and its traditional challenges like data inflation and latency. Yet, it still misses out on energy considerations crucial for edge scenarios [7]. CStream aims to bridge these gaps by concentrating on the perpetual stream of data, encompassing various data use-cases, and prioritizing energy efficiency. By integrating these aspects, CStream advances the domain of stream compression in IoT edge computing, offering a solution that is both comprehensive and tailored to edge-specific requirements.

Exploiting Asymmetric Architecture at the Edge: Asymmetric architectures have been demonstrated to be highly effective in delivering high-performance computation with energy efficiency, a crucial demand in edge computing [41], [48], [53], [71], [72], [73], [81], [82], [83], [84]. While significant advancements have been made in comprehending and managing asymmetry in workload scheduling [53], [72], [85], no studies have specifically explored stream compression tasks. CStream fills this research gap by leveraging the fine-grained behavior of stream compression algorithms and the potential impact of workloads on asymmetric architecture utilization.

VII. CONCLUSION

This paper presents CStream, a framework designed to boost stream compression on multicore edge devices. By integrating a suite of compression algorithms and exploiting the strengths of both symmetric and asymmetric multicore architectures, CStream tackles the hurdles inherent in IoT edge computing. It seeks to achieve unparalleled compression ratios, elevated throughput, reduced latency, and minimized energy consumption. Our assessments confirm CStream's exceptional efficiency: compared to an intuitive solution with 1) random choice of stream compression algorithms [14], [32], [33], 2) lock-based parallelization [86] and 3) coarse-grained, compression-oblivious hardware resource management [53], it realizes a compression ratio of $2.8\times$ with negligible information loss, enhances throughput by $4.3\times$, and substantially diminishes latency (by 65%) and energy use (by 89%). These findings underscore the benefits of a cohesive approach to software and hardware design. Looking ahead, we aim to refine CStream's algorithm selection mechanism through the integration of cutting-edge machine learning techniques and to evaluate the potential of bespoke hardware accelerators in further elevating CStream's performance and energy efficiency. Moreover, future work will consider the application of CStream in enhancing system development, particularly in facilitating more effective cloud-edge interaction.

REFERENCES

[1] G. Pekhimenko, C. Guo, M. Jeon, P. Huang, and L. Zhou, "Tersecades: Efficient data compression in stream processing," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA: USENIX Association, 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 *Proc. Int. Conf. Field-Programmable Technol.*, 2021, pp. 1–9.

[3] X. Zeng and S. Zhang, "A hardware-conscious stateful stream compression framework for IoT applications (vision)," in *Proc. ACM Int. Conf. Distrib. Event-Based Syst.*, 2023, pp. 7–12.

[4] X. Zeng and S. Zhang, "Parallelizing stream compression for IoT applications on asymmetric multicores," in *Proc. IEEE 39rd Int. Conf. Data Eng.*, 2023, pp. 950–964.

[5] Y. Li et al., "Camel: Managing data for efficient stream learning," in *Proc. Int. Conf. Manage. Data*, 2022, pp. 1271–1285.

[6] S. Zeuch et al., "The nebulastream platform for data and application management in the internet of things," in *Proc. 10th Conf. Innov. Data Syst. Res.*, Amsterdam, The Netherlands, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

[7] R. Duvignau, V. Gulisano, M. Papatriantafilou, and V. Savic, "Streaming piecewise linear approximation for efficient data management in edge computing," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, New York, NY, USA: Association for Computing Machinery, 2019, pp. 593–596, doi: 10.1145/3297280.3297552.

[8] M. Bansal, I. Chana, and S. Clarke, "A survey on IoT big data: Current status, 13 v's challenges, and future directions," *ACM Comput. Surv.*, vol. 53, no. 6, 2020, doi: 10.1145/3419634.

[9] Arm solutions for IoT, 2021, Accessed: May 10, 2021. [Online]. Available: https://www.arm.com/solutions/iot

[10] Rockchip wiki rk3399, 2021, Accessed: May 10, 2021. [Online]. Available: http://opensource.rock-chips.com/wiki_RK3399

[11] Allwinner soc family, 2013, Accessed: May 10, 2022. [Online]. Available: https://linux-sunxi.org/Allwinner_SoC_Family

[12] Intel atom x5-z8350 processor, 2021, Accessed: May 10, 2022. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/93361/intel-atom-x5z8350-processor-2m-cache-up-to-1-92-ghz.html

[13] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2006, pp. 671–682.

[14] D. Blalock, S. Madden, and J. Guttag, "Sprintz: Time series compression for the Internet of Things," *Proc. ACM InterAct. Mobile Wearable Ubiquitous Technol.*, vol. 2, no. 3, pp. 1–23, 2018.

[15] zlib home page, 2017, Accessed: Jun. 29, 2021, [Online]. Available: http://www.zlib.net/

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.

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

[17] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An experimental study of bitmap compression versus. inverted list compression," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 993–1008.

[18] Dalvik executable format in android, 2020, Accessed: May 29, 2022, [Online]. Available: https://source.android.com/devices/tech/dalvik/dex-format.html

[19] Y. Zhou, Z. Vagena, and J. Haustad, "Dissemination of models over time-varying data," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 864–875, 2011.

[20] T. Lu, W. Xia, X. Zou, and Q. Xia, "Adaptively compressing {IoT} data on the resource-constrained edge," in *Proc. 3rd USENIX Workshop Hot Top. Edge Comput.*, 2020.

[21] Mit-bih database distribution, 2005, Accessed: Jun. 29, 2021, [Online]. Available: http://ecg.mit.edu/

[22] Creator of the angry birds game, 2019, Accessed: May 10, 2021, [Online]. Available: www.rovio.com

[23] Shanghai stock exchange, 2018, Accessed: Nov. 12, 2021, [Online]. Available: http://english.sse.com.cn/

[24] Beach weather stations - automated sensors, 2021, Accessed: Nov. 12, 2021, [Online]. Available: https://catalog.data.gov/dataset/beach-weather-stations-automated-sensors/resource/3b820f68-4dca-4ea7-8141-f37d9237734d

[25] Tracetogether, safer together, 2021, Accessed: Nov. 7, 2021, [Online]. Available: https://www.tracetogether.gov.sg/

[26] D. Reinsel et al., "The digitization of the world from edge to core," *Framingham: Int. Data Corporation*, vol. 16, pp. 1–28, 2018.

[27] S. Zhang, J. He, A. C. Zhou, and B. He, "Briskstream: Scaling data stream processing on shared-memory multicore architectures," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 705–722.

[28] B. He et al., "Relational query coprocessing on graphics processors," *ACM Trans. Database Syst.*, vol. 34, no. 4, pp. 1–39, 2009.

[29] A. Ukil, S. Bandyopadhyay, and A. Pal, "IoT data compression: Sensor-agnostic approach," in *Proc. Data Compression Conf.*, 2015, pp. 303–312.

[30] (2020) Eclipse IoT working group. IoT developer survey 2018. 2018. [Online]. Available: https://https://blogs.eclipse.org/post/benjamin-cab%C3%A9/key-trends-iotdeveloper-survey-2018

[31] C. Baskin et al., "Uniq: Uniform noise injection for non-uniform quantization of neural networks," *ACM Trans. Comput. Syst.*, vol. 37, no. 1/4, pp. 1–15, 2021.

[32] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Trans. Inf. Theory*, vol. 21, no. 2, pp. 194–203, Mar. 1975.

[33] lz4 source code, 2021, Accessed: Jul. 25, 2021. [Online]. Available: https://github.com/lz4/lz4/

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

[35] A. Gupta et al., "Modern lossless compression techniques: Review, comparison and analysis," in *Proc. 2nd Int. Conf. Elect. Comput. Commun. Technol.*, 2017, pp. 1–8.

[36] A. Moffat, "Huffman coding," *ACM Comput. Surv.*, vol. 52, no. 4, pp. 1–35, 2019.

[37] 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 *Proc. 21st Int. Conf. Parallel Architectures Compilation Techn.*, 2012, pp. 377–388.

[38] Up board series, 2021, Accessed: May 10, 2021. [Online]. Available: https://up-board.org/up/specifications/

[39] Y. Dua, V. Kumar, and R. S. Singh, "Parallel lossless HSI compression based on RLS filter," *J. Parallel Distrib. Comput.*, vol. 150, pp. 60–68, 2021.

[40] V. Pankratius, A. Jannesari, and W. F. Tichy, "Parallelizing bzip2: A case study in multicore software engineering," *IEEE Softw.*, vol. 26, no. 6, pp. 70–77, Nov./Dec. 2009.

[41] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *ACM Comput. Surv.s*, vol. 48, no. 3, pp. 1–38, 2016.

[42] W. Zhang et al., "ELF: Accelerate high-resolution mobile deep vision with content-aware parallel offloading," in *Proc. 27th Annu. Int. Conf. Mobile Comput. Netw.*, 2021, pp. 201–214.

[43] Z. Pan et al., "Exploring data analytics without decompression on embedded gpu systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 7, pp. 1553–1568, Jul. 2022.

[44] Dra829j-q1 asymmetric processor, 2022, Accessed: May 10, 2022, [Online]. Available: https://www.ti.com/product/DRA829J-Q1

[45] S. Yang et al., "Adaptive energy minimization of embedded heterogeneous systems using regression-based learning," in *Proc. 25th Int. Workshop Power Timing Model. Optim. Simul.*, 2015, pp. 103–110.

[46] W. Wolff and B. Porter, "Performance optimization on big.little architectures: A memory-latency aware approach," in *Proc. 21st ACM SIGPLAN/SIGBED Conf. Lang.s Compilers Tools Embedded Syst.*, New York, NY, USA: Association for Computing Machinery, 2020, pp. 51–61. [Online]. Available: https://doi.org/10.1145/3372799.3394370

[47] H. Ribic and Y. D. Liu, "Energy-efficient work-stealing language runtimes," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 513–528, Feb. 2014, doi: 10.1145/2654822.2541971.

[48] T. Somu Muthukaruppan, A. Pathania, and T. Mitra, in *Price Theory Based Power Manage. for Heterogeneous Multi-Cores*, New York, NY, USA: Association for Computing Machinery, 2014, pp. 161–176, doi: 10.1145/2541940.2541974.

[49] S. Yamagiwa, E. Hayakawa, and K. Marumo, *Adaptive Entropy Coding Method for Stream-Based Lossless Data Compression*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 265–268, doi: 10.1145/3387902.3394037.

[50] S. Yamagiwa, R. Morita, and K. Marumo, "Bank select method for reducing symbol search operations on stream-based lossless data compression," in *Proc. Data Compression Conf.*, 2019, pp. 611–611.

[51] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Workshop Hot Top. Cloud Comput.*, 2010.

[52] S. Albers and H. Fujiwara, "Energy-efficient algorithms for flow time minimization," *ACM Trans. Algorithms*, vol. 3, no. 4, pp. 49–65, 2007.

[53] M. Wang, S. Ding, T. Cao, Y. Liu, and F. Xu, "Asymo: Scalable and efficient deep-learning inference on asymmetric mobile cpus," in *Proc. 27th Annu. Int. Conf. Mobile Comput. Netw.*, 2021, pp. 215–228.

[54] S. Zhang, Y. Wu, F. Zhang, and B. He, "Towards concurrent stateful stream processing on multicore processors," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1537–1548.

[55] C.-Y. Wei, Y.-T. Hong, and C.-J. Lu, "Online reinforcement learning in stochastic games," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017.

[56] INA226, 2021, Accessed: Nov. 12, 2021. [Online]. Available: https://www.ti.com/product/INA226

[57] ESD32S2, 2021, Accessed: Nov. 12, 2021. [Online]. Available: https://www.espressif.com/en/products/socs/esp32-s2

[58] Intel 64 and ia-32 architectures software developer's manual, 2016, Accessed: Mar. 12, 2021, [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf

[59] H. Han, Y. Li, and X. Guo, "User identity-aided pilot access scheme for massive mimo-idma systemxf," *IEEE Trans. Veh. Technol.*, vol. 68, no. 6, pp. 6197–6201, Jun. 2019.

[60] Y. Tai et al., "Towards highly accurate and stable face alignment for high-resolution videos," in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 8893–8900.

[61] S. Zauli-Sajani, S. Marchesi, C. Pironi, C. Barbieri, V. Poluzzi, and A. Colacci, "Assessment of air quality sensor system performance after relocation," *Atmospheric Pollut. Res.*, vol. 12, no. 2, pp. 282–291, 2021.

[62] D. A. Jacobs and D. P. Ferris, "Estimation of ground reaction forces and ankle moment with multiple, low-cost sensors," *J. NEur.–2–> Rehabil.*, vol. 12, no. 1, pp. 1–12, 2015.

[63] E. Shahabpoor and A. Pavic, "Estimation of vertical walking ground reaction force in real-life environments using single imu sensor," *J. Biomech.*, vol. 79, pp. 181–190, 2018.

[64] S. Zhang et al., "Parallelizing intra-window join on multicores: An experimental study," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 2089–2101.

[65] ROCK pi 4 wiki, 2021, Accessed: May 10, 2021, [Online]. Available: https://wiki.radxa.com/Rockpi4

[66] Getting started with p2-zero, 2021, Accessed: May 10, 2021, [Online]. Available: https://wiki.banana-pi.org/Getting_Started_with_P2-Zero

[67] Getting started with jetson agx orin developer kit, 2023, Accessed: Feb. 5, 2024, [Online]. Available: https://developer.nvidia.com/embedded/learn/get-started-jetson-agx-orin-devkit

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

[69] Y. J. Lo et al., "Roofline model toolkit: A practical tool for architectural and program analysis," in *Proc. Int. Workshop Perform. Model. Benchmarking Simul. High Perform. Comput. Syst.*, 2014, pp. 129–148.

[70] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *Proc. 32nd Int. Symp. on Comput. Architecture*, 2005, pp. 506–517.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.

[71] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Proc. Int. Conf. Compilers Architecture Synth. Embedded Syst.*, 2013, pp. 1–10.

[72] T. Yu et al., "Collaborative heterogeneity-aware OS scheduler for asymmetric multicore processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1224–1237, May 2021.

[73] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *Proc. 39th Annu. Int. Symp. Comput. Architecture*, 2012, pp. 213–224.

[74] 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 Comput. Architecture News*, vol. 41, no. 3, pp. 48–59, 2013.

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

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

[77] 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 Trans. Database Syst.*, vol. 44, no. 3, pp. 1–46, 2019.

[78] K. Iqbal, N. Khan, and M. G. Martini, "Performance comparison of lossless compression strategies for dynamic vision sensor data," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2020, pp. 4427–4431.

[79] M. Bharde et al., "{Store-Edge }{ RippleStream}: Versatile infrastructure for {IoT} data transfer,," in *Proc. USENIX Workshop Hot Top. Edge Comput.*, 2018.

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

[81] 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.

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

[83] 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 *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2017, pp. 625–638.

[84] Q. Zeng, Y. Du, K. Huang, and K. K. Leung, "Energy-efficient resource management for federated edge learning with CPU-GPU heterogeneous computing," *IEEE Trans. Wireless Commun.*, vol. 20, no. 12, pp. 7947–7962, Dec. 2021.

[85] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Architecture*, 2013, pp. 13–24.

[86] D. Wang, E. A. Rundensteiner, and R. T. Ellison III, "Active complex event processing over event streams," *Proc. VLDB Endowment*, vol. 4, no. 10, pp. 634–645, 2011.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2026 at 02:22:02 UTC from IEEE Xplore. Restrictions apply.