A Hardware-Conscious Stateful Stream Compression Framework for IoT Applications (Vision)

Abstract

Data stream compression has attracted vast interest in emerging IoT (Internet of Things) applications. However, adopting stream compression on IoT applications is non-trivial due to the divergent demands, i.e., low energy consumption, high throughput, low latency, high compressibility, and tolerable information loss, which sometimes conflict with each other. This is particularly challenging when adopting stateful stream compression algorithms, which rely on *states*, e.g., a dictionary or model. This paper presents our vision of `CStream`, a hardware-conscious stateful stream compression framework for IoT applications. Through careful hardware-conscious optimizations, `CStream` will minimize energy consumption while striving to satisfy the divergent performance demands for parallelizing complex stateful stream compression algorithms for IoT applications.

CCS Concepts: • **Streams and complex event processing** → **Stateful Stream Compression**; • **Emerging hardware** → **IoT devices**.

Keywords: Stream compression, IoT and Edge Computing, Asymmetric and Heterogeneous Hardware

ACM Reference Format:
. 2023. A Hardware-Conscious Stateful Stream Compression Framework for IoT Applications (Vision). In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Data stream compression, i.e., continuously compressing input data tuples, attracts much attention recently [7, 10], especially due to the rise of IoT applications. Figure 1 demonstrates an IoT use case [2] where stream compression is highly attractive to be adopted. In this application, real-time data streams (e.g., toxic gas, temperature) from massive IoT sensors, deployed in dangerous areas, are continuously

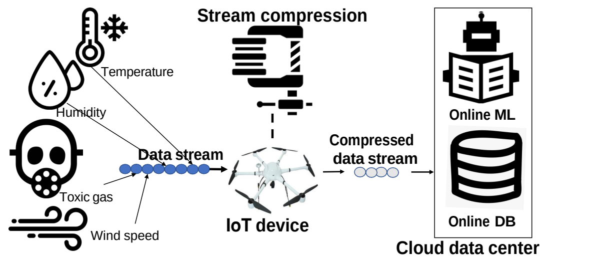


Figure 1. Stream compression may be applied during real-time data gathering by the patrol drone where humans can not enter.

gathered by memory-limited, battery-powered patrol drones (i.e., IoT devices). To reduce transmission overhead, the drone may compress input streams by its equipped multicore processors [2] before passing them to downstream online IoT analytic tasks, such as online aggregation [10], and online machine learning [8] in the cloud.

Parallelizing stream compression on IoT devices, such as the wireless patrol drone in Figure 1, is mandatory to meet the strict high-throughput and low-latency processing requirement. However, it is a non-trivial task as it involves divergent, sometimes conflicting, additional demands including low energy consumption [9], high compression ratio [8], and tolerable information lose [7]. It is particularly challenging when stream compression relies on *states*, i.e., the intermediate value that will be used in subsequent operations during the compressing of data streams (Def 2). Some typical forms of states during stateful stream compression include dictionary [16] and model [26]. Those stateful stream compression algorithms gaining increasing traction nowadays, as they significantly outperform stateless algorithms in terms of compressibility and information loss control [3, 7, 13, 26]. Unfortunately, managing states bring even more difficulties for parallelization [16].

Recent efforts have been devoted to 1) conducting data compression on IoT devices [17] and 2) continuously compressing data streams [7, 10]. For example, a recent IoT-aware database Vergedb [17] proposes to automatically select the compression approach, given the workload, data arrival rates, and resource capacity. However, existing frameworks such as Vergedb are limited to relying on a coarse-grained analysis of performance tradeoffs among data compression algorithms. They fail to fully explore a large software-hardware codesign space and do not intrinsically support state management for complex stateful stream compression algorithms. To the best

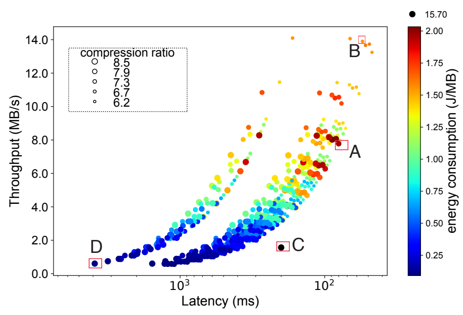


Figure 2. The large feasible solution space of applying piecewise linear approximation (PLA) [26] under relative error bound 0.01 to compress *ECG* [1] data on RK3399 [2] asymmetric multicores as the IoT device.

of our knowledge, none of them answers the question:

> *How to best parallelize stateful stream compression on IoT devices under energy budget constraints?*

Motivating Evaluation Results. Figure 2 illustrates a large feasible solution space for parallelizing a stateful stream compression algorithm on a modern multicore IoT device, where the state is a piecewise linear approximation (PLA) model [26]. There are three major takeaways. First, various concurrency control mechanisms for accessing the states during stream compression lead to a large design space. For example, the solution *A* configures all threads to maintain a global-shared PLA model, and it achieves a high compression ratio of 8.5. In contrast, solution *B* achieves higher throughput by using a shared-nothing approach [16], where each thread maintains a disjoint PLA model, but the compression ratio drops to 6.2. Second, hardware-conscious optimization is necessary. We highlight a hardware-oblivious solution *C*, which has a simple lock-based concurrency control, OS-based workload scheduling, and random input stream admission. Compared with a hardware-conscious solution such as *A*, *C* wasted more than 6.9× energy consumption while achieving about 1.5× higher latency and 80.5% lower throughput. Third, there is a complex non-linear correlation between energy consumption and other performance demands. As a result, adapting a solution to another to cope with the changes in workloads and performance demands is necessary but challenging. For example, it can achieve about 10× higher throughput and 1/40 lower latency by consuming 8× more energy without losing compressibility comparing solution point *D* (a single-thread implementation) to *A* (a parallel implementation).

In this paper, we describe our vision of a hardware-conscious stateful stream compression framework for IoT applications, namely `CStream`. Differ significantly from prior works, `CStream` aims to hit the sweet spot among the divergent performance demands when parallelizing stateful stream compression on multicore IoT devices. `CStream` will achieve the goal via three key designs, centering around built-in support of state management, hardware-conscious optimizations, and adaptive to dynamic IoT environment:

- First, we introduce a compression procedure compilation module (Section 4.1) that parallelizes a given stream compression procedure (Def 1) into a parallel execution plan as a Directed Acyclic Graph (DAG) [23, 24] with suitable state representations.
- Second, we propose a parallel stateful runtime (Section 4.2) with efficient concurrency control of shared state access and hardware-conscious scheduling mechanisms.
- Third, we design an adaptive control plane (Section 4.3) that manages the input stream admission and runtime adjusting to cope with the dynamicity of IoT environments.

Our early experiments have revealed that `CStream` is able to obtain satisfying solutions such as *A* and *B* as shown in Figure 2 to meet divergent performance demands. Given the encouraging results, we envision that `CStream` can be a key component of the next-generation large-scale IoT deployment [5], especially, since it also aligns with the recent initiative of green computing. Beyond compressing data streams to reduce transmission overhead, it becomes even more beneficial when downstream analytic tasks execute directly over compressed data streams. We leave it as an immediate next step to further enhance `CStream`.

2 Related Work

This section reviews the related work and reveals the limitations that motivate `CStream`.

Stateful Data Compression Algorithms. There is a growing interest in exploring complex stateful compression algorithms for emerging IoT applications. For instance, the dictionary-based states such as LZMA [11] and model-based states like piecewise linear approximation (PLA) [3] are attractive in compressing dynamic vision and achieving error-bounded lossy compression over time series data. Recent work from Li et al [13] also exploits the summarizing-based state during data compression for machine learning. These studies provide valuable insights into utilizing the state in different cases of data compression. `CStream` is broader than their scope, as we exploit hardware-conscious parallelization of stateful stream compression on multicore IoT devices.

Parallelizing Data Compression. Prior work has attempted parallelizing data compression using various hardware architectures such as CPU [7], GPU [16] and FPGA [10]. Especially, recent work such as Tersecades[7] and StreamZip [10] have explored how to accelerate the stream processing operations (i.e., join and aggregation, respectively) by compressing data streams on parallel architectures. They achieve satisfying scaling up on compression, but none of them is designed for multicore IoT

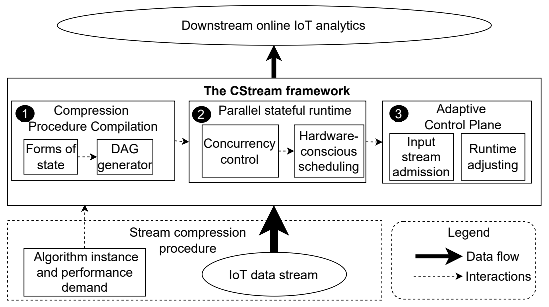

Downstream online IoT analytics
The CStream framework
1 Compression Procedure Compilation
Forms of state
DAG generator
2 Parallel stateful runtime
Concurrency control
Hardware-conscious scheduling
3 Adaptive Control Plane
Input stream admission
Runtime adjusting
Stream compression procedure
Algorithm instance and performance demand
IoT data stream
Legend
Data flow
Interactions

Figure 3. System overview

devices with a constrained energy budget. Further, they lack intrinsic support of state management for complex stateful stream compression algorithms.

Utilizing Novel IoT Hardware. A growing interest has been shown in utilizing asymmetric (e.g., the ARM big. LITTLE processors [14]) and heterogeneous multicore devices (e.g., the coupled CPU-GPU [22]) in IoT, as they can achieve a better trade-off between energy efficiency and other performance demands like throughput compared with symmetric architectures. Recent studies are especially interested in exploring collaborative OS schedulers and energy-efficient machine learning on asymmetric multicore [21] or coupled CPU-GPU architectures [22]. They make valuable contributions to understanding the novel hardware and conducting hardware-conscious workload scheduling, but none investigated the large design space of parallel stateful stream compression on multicore IoT devices.

3 System Overview

In this section, we first outline how CStream works, then we illustrate its key design challenges, followed by a design overview.

3.1 How CStream Works?

As shown in Figure 3, acting as a compression middleware, CStream sits between IoT devices such as sensors and downstream online analytic tasks. To meet various performance demands CStream parallelizes a *stream compression procedure* on multicore IoT devices, defined as follows.

Definition 1 (Stream Compression Procedure). *A stream compression procedure is the process of applying a specific stream compression algorithm to compress a subset of the input data stream under specific demands of energy consumption, throughput, latency, compressibility, and information loss.*

CStream especially targets parallelizing *stateful stream compression procedures* that maintain various kinds of states to help achieve higher compressibility and/or lower information loss, where the *state* can be defined as follows.

Definition 2 (State). *A state in stream compression is the intermediate value or data structure that maintains or approximates historic information of the input data stream to help the algorithm to better determine and apply the compression strategies for subsequent data streams.*

3.2 Design Challenges

CStream needs to address the following three challenges.

The Challenge of Efficient State Management. Achieving effective parallelization over various compression algorithms is difficult [16], especially when the algorithm is *stateful* [24]. In practice, varying kinds of states may be involved in compression, e.g., the last-encountered value [7], the dictionary [16], the sketch structure [13], and the piecewise linear approximation model [3]. Differ significantly from prior work, CStream treats efficient state management as a first-class citizen for better managing those states during stream compression.

The Challenge of Hardware Conscious Optimization. Achieving hardware consciousness parallelization of stream compression is non-trivial [9]. Especially, CStream needs to make wise decisions in a) the selection of concurrency control [24] of state access among multiple parallel instances, and b) the proper workload scheduling on novel IoT devices, which are typically asymmetric [14] and/or heterogeneous [22].

The Challenge of Dynamicity in IoT Environment. Fitting into the dynamics and uncertainty on the fly in a real-world IoT environment [5, 6] introduces another non-trivial design challenge to CStream. In particular, both the admission of infinite, potentially out-of-order data stream and the parallel runtime should be reconfigurable and adaptive to dynamically changing workloads.

3.3 Design Overview

To address the aforementioned challenges, CStream consists of three major components, i.e., a compression procedure compilation, a parallel stateful runtime, and an adaptive control plane, as shown in Figure 3.

CStream takes three consecutive steps to conduct a given procedure. ❶ In the compression procedure compilation, a selected compression algorithm is compiled into a parallel execution plan, which is represented as a DAG with multiple nodes. The compilation exploits both pipelining parallelism and data parallelism. Treating the state management as the first citizen, CStream supports five state-of-art major representations of states during the compilation. ❷ The DAG is then sent to the parallel stateful runtime for execution. There is a notorious problem of *concurrent state access* [6, 12, 24] during stateful stream processing. For CStream, its runtime must decide suitable concurrency control approaches in order to achieve a good balance between compressibility and accessing overhead. Afterward, each running node of the DAG is scheduled with hardware consciousness, by being assigned 1) the right mapping place of hardware and 2) the right portion of the workload according to the procedure demands. ❸ An adaptive control plane is responsible for addressing the dynamicity. Specifically, it regulates the input stream admission in an efficient manner with *micro batcing* and *out-of-order* stream handling [23]. It also chooses the optimal way of adjusting the runtime, in terms of both computational power and software configurations [6]. The adjustments cover the necessary guarantee of state consistency, and they are specially optimized for the long-running online IoT analytics and the dynamic IoT data stream [6].

4 `CStream` Components

In this section, we elaborate on the crucial design of `CStream` in more detail and discuss some of our ongoing works.

4.1 Compression Procedure Compilation

We first introduce how `CStream` compiles the stream compression procedure into a DAG.

4.1.1 Forms of State. `CStream` supports five major forms of compression state which are popularly used in stream compression algorithms as follows. **(1) Stateless compression**, which repeats the compression on the current arrived piece of data without backtracing historical information [20]. **(2) Value-based State**, which is about updating and recording the recent compressed value to improve the compression ratio, such as delta encoding [7] and run length encoding (RLE) [20]. **(3) Dictionary-based State**, which utilizes a dictionary such as hash table [11] to memorize thousands of encountered "last compressed" values, further improving compressibility with higher maintenance overhead. **(4) Model-based State**, which reflects large volumes of data by using approximate mathematical models (e.g., under linear or logistic regression) with only a few parameters [26], achieving ultra-high compression ratio [3] when data is highly fitted to the model. **(5) Summarizing-based State**, which aims to acquire an approximate summary of historic information in several incrementally updated *buckets* or *cells* [8] instead of keeping accurate records, enabling tunable trade-off between compressibility and information loss.

4.1.2 DAG Generator. Based on the state representation, the DAG generator is then involved to exploit both *pipelining parallelism* and *data parallelism* opportunities for compilation inspired by [2, 23]. To achieve pipelining parallelism, `CStream` divides a stream compression procedure into three pipeline steps of *read*, *encode*, and *write*, and further divide *encode* step into three state-related steps [24], i.e., *preprocess*, *state access*, and *postprocess*. For data parallelism, each aforementioned pipeline step can be replicated for providing higher performance and relieving pipeline bottlenecks [23]. However, achieving data parallelism is not always straightforward as concurrent accesses over a compression state may be involved. It is particularly challenging as `CStream` needs to support multiple forms of states (Section 4.1.1). We discuss how `CStream` addresses this issue by concurrency control in the following (Section 4.2.1).

4.2 Parallel Stateful Runtime

We discuss how `CStream` conducts the parallel execution of stream compression DAG and achieves hardware-conscious.

4.2.1 Concurrency Control. We discuss various concurrency control approaches of the state in the following. **1) The Shared-Nothing Approach:** Conducting parallel stream compression without sharing states is a natural choice for maximizing parallelism [16]. However, splitting the state will lose compressibility as a parallel instance is unaware of the historical information about the data stream maintained by others. **2) The Partial-Share Approach:** To alleviate the loss of compressibility caused by the nothing-share approach, the state can be partially shared, e.g., to share the state among some neighbours [18]. Partial sharing is an attractive strategy [15], but questions such as 1) which states to share and 2) how to conduct the partial sharing remain to be answered for parallelizing stateful stream compression. We are currently working on a holistic model to determine the optimal configuration for the partial-sharing of states during stateful stream compression in `CStream`. **3) The Global-Share Approach:** It is also possible to maintain a global shared state while still providing parallelism by taking advantage of state-of-art technologies like concurrent storage [12] or transactional stream processing [24]. However, how to apply these technologies in stateful stream compression is still an open question. In addition, the relatively strong ACID guarantee [24] may not be a necessity for stream compression as long as 1) the compressed data is able to be correctly decompressed (or directly processed) by downstream tasks, and 2) there is only marginal information loss within the specified error-bound.

4.2.2 The Hardware-Conscious Scheduling. Many prior works have revealed the benefits of adopting asymmetric and heterogeneous multicores [14, 22] on IoT devices. In general, they are superior to traditional symmetric hardware in higher performance and higher energy efficiency. However, it is non-trivial to fully take advantage of them in `CStream`, due to the so-called asymmetric computation and asymmetric communication effects [2, 21]. A recent work [2] points out that precise and low-overhead cost modeling is required to guide the hardware-conscious scheduling in stream compression, but it still remains an open challenge as stateful stream compression algorithms, IoT hardware, and dynamic workload should be considered together in an adaptive and cost-effective manner. We are working on more sophisticated controllers that monitor workload statistical information in the data stream [23] to better guide the hardware-conscious scheduling.

4.3 Adaptive Control Plane

To address the dynamicity issue in IoT environments, CStream involves an adaptive control plane of input stream admission and runtime adjusting.

4.3.1 Input Stream Admission. We first discuss how CStream admits input stream including the execution over infinite streaming data and the out-of-order handling in a real-world IoT environment.

Micro-batch Execution. Compressing the data stream pieces as soon as their arrival seems a natural fit for handling infinite streaming inputs. However, such an eager approach is costly. Inspired by [25], CStream adapts the *micro-batching* strategy and conducts stream compression lazily. One subsequent question arises as "how much data should be agglomerated in each micro-batching". Our preliminary results indicate that a suitable batch size correlates to cache size, especially the L1D size. However, conflicts among different performance demands occur in tuning the batch size, e.g., a larger batch size may be beneficial to improve throughput, but also increases the latency.

Out-of-Order Handling. The out-of-order arrival of the data stream is another notorious problem [23], especially for IoT applications [9]. We need to revisit existing approaches [4] for stateful stream compression on IoT devices. Specifically, we are working on two following aspects. First, beyond the latency penalty and arrival pattern [4], we need to investigate how out-of-order arrival can affect the compression ratio and information loss. For instance, the disordered or missing data is harmful to constructing an efficient approximation model in the model-based state. Second, we will explore how to achieve compression-specific optimizations in reducing unnecessary re-ordering and waiting for data from two aspects. (1) 100% correctness of compressed data may not be necessary for some IoT tasks like online machine learning [13], CStream is hence designed to give approximately correct compressed data with the partial disorder or missing, that come with lower latency, higher throughput, and lower energy consumption as benefits. (2) Some output data structures like the lz4-token can potentially indicate the correct decompression order [19] even after an out-of-order compression. CStream will utilize such inherent resistance in achieving out-of-order handling with lower overhead.

4.3.2 Runtime Adjusting. We now introduce how CStream adjusts its runtime under the dynamics in the following.

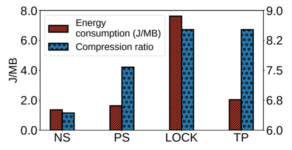
Adjusting Computational Power. CStream provides necessary computational power on demand without wasting energy due to the limited energy budget on IoT devices, and adjusts the computational power as follows: *(1) Frequency regulation* trades off time and energy without requiring other settings to change [6], and it's often conducted under the *dynamic voltage and frequency scaling* (DVFS) technology. However, due to the obliviousness of performance demands by stream compression and coarse-grained guarantee [21] of OS-provided DVFS, we are working on evaluating and adopting some state-of-art DVFS [21] in CStream framework. *(2) Core number regulation* offers a large tuning space of time-energy trading off but leads to high complexity [6]. Specifically, we should merge or split some data parallelism nodes and remap some existing nodes in the DAG to fit into the changed core number[23]. Meanwhile, specific reconfiguration protocols must be used to preserve stream and internal state integrity [6].

Adjusting Software Configurations. The streaming workloads in the IoT environment vary significantly in both arrival patterns and statistical properties over time [5, 9]. Static profiling and fixed strategies are insufficient, we are investigating how to achieve the adaptive configuration of CStream (e.g., workload scheduling, input stream admission, etc). Moving beyond a recent feedback-based control [2], CStream will enable (1) adaptive input stream admissions such as configuring the optimal batch size [6] and generating the optimal watermark [4] on the fly and (2) utilizing more powerful approaches such as proactive-model [23] and machine-learning [6].

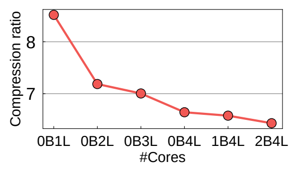
5 Preliminary Evaluation

CStream supports all aforementioned concurrency control approaches (Section 4.2.1) for state management in stream compression. Specifically, we follow a prior work [16] to implement the shared nothing approach (*NS*), enable each thread to have state read access on its two nearest neighbors to implement the partial-share approach (*PS*), and implement global-share approach by either lock-based implementation (*LOCK*) and transactional-based [24] implementation (*TP*). We use RK3399 asymmetric multicores as the IoT device [2] for PLA compression [26] on an ECG dataset [1]. We use "B" to denote big cores and "L" to denote little cores on RK3399, and enable all cores by default, i.e., "2B4L". Unless otherwise specified we let each core work under its highest frequency, i.e., 1.416GHz for B and 1.8GHz for L. We use a 2K byte as batch size for micro batching and use an asymmetry-aware scheduling strategy.

Figure 4 shows the comparison of different concurrency control implementations. There are three major observations. First, although the *NS* approach leads to minimal energy consumption, its compressibility is the lowest (i.e., up to 24% less compression ratio compared with *LOCK* or *TP*).

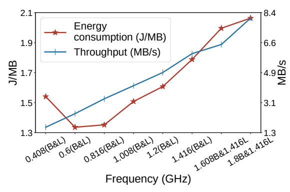


(a) Compression ratio and energy consumption.

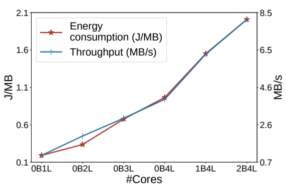


(b) Compression ratio drops while scaling up with *NS*.

Figure 4. Preliminary evaluation of various concurrency control implementations in `CStream`.



(a) Impacts of different frequency settings.



(b) Impacts of different core number settings.

Figure 5. Preliminary results about adjusting the runtime.

Even worse, the compression ratio drops with an increasing number of cores, as shown in Figure 4(b). Second, *LOCK* brings much higher energy consumption, i.e., 4.6× more energy consumption than *NS*, although it can guarantee the highest compression ratio. Compared with *LOCK*, the novel *TP* implementation avoids the centralized lock contentions while still allowing the global sharing of states. Third, there is an interesting space for trading off compressibility and overhead between *PS* and *TP* (i.e., 7.7 ∼ 8.5 compression ratio, 1.6 ∼ 2.0 J/MB energy consumption), and we plan to further investigate it in the future.

Figure 5 further shows that `CStream` can adjust runtime by setting different core frequencies and numbers of cores. Noting that the relationship between energy consumption and frequency can be non-monotonic. We observed that this is because the energy consumption decreases or increases with increasing frequency in the little core or big core, respectively. These results highlight that a more careful design is required in achieving varying demands in dynamic workloads in `CStream`.

6 Conclusion

With the increasing deployment of massive IoT devices, continuously reducing data footprints between data sources and downstream IoT analytics becomes crucial. Subsequently, there is growing interest in adopting stream compression for IoT, especially for stateful stream compression which has great potential and emerging applications. However, such adoption needs to carefully take multiple performance demands into account when realizing stream compression algorithms with complex state management on energy budget-constrained IoT devices. We presented our vision of `CStream` to hopefully bridge the literature gap.

References

[1] 2005. *MIT-BIH Database Distribution,http://ecg.mit.edu/*. Last Accessed: 2021-06-29.

[2] Anonymous Authors. 2022. Parallelizing Stream Compression for IoT Applications on Asymmetric Multicores (Technical Report).

[3] Davis Blalock et al. 2018. Sprintz: Time series compression for the internet of things. *In ACM IMWUT* 2, 3 (2018), 1–23.

[4] Awad et al. 2019. Adaptive Watermarks: A Concept Drift-based Approach for Predicting Event-Time Progress in Data Streams.. In *EDBT*. 622–625.

[5] Bansal et al. 2020. A Survey on IoT Big Data: Current Status, 13 V's Challenges, and Future Directions. 53, 6 (2020).

[6] Cardellini et al. 2022. Runtime Adaptation of Data Stream Processing Systems: The State of the Art. *CSUR* 54, 11s (2022), 1–36.

[7] Gennady Pekhimenko et al. 2018. TerseCades: Efficient Data Compression in Stream Processing. In *USENIX ATC 18*. Boston, MA.

[8] Li et al. 2022. Camel: Managing Data for Efficient Stream Learning. In *SIGMOD 2022*. 1271–1285.

[9] Steffen Zeuch et al. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *CIDR 2020*.

[10] Prajith Ramakrishnan Geethakumari et al. 2021. Streamzip: Compressed sliding-windows for stream aggregation. In *ICFPT*. IEEE.

[11] Khurram Iqbal et al. 2020. Performance comparison of lossless compression strategies for dynamic vision sensor data. In *ICASSP*. IEEE, 4427–4431.

[12] Søren Kejser Jensen et al. 2018. Modelardb: Modular model-based time series management with spark and cassandra. *VLDB* 11, 11 (2018).

[13] Yiming Li et al. 2022. Camel: Managing Data for Efficient Stream Learning. In *SIGMOD*. 1271–1285.

[14] Sparsh Mittal. 2016. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 1–38.

[15] Muhammad Anis Uddin Nasir et al. 2015. The power of both choices: Practical load balancing for distributed stream processing engines. In *ICDE*. IEEE, 137–148.

[16] Adnan Ozsoy et al. 2011. CULZSS: LZSS lossless data compression on CUDA. In *ICCC*. IEEE, 403–411.

[17] John Paparrizos et al. 2021. VergeDB: A Database for IoT Analytics on Edge Devices.. In *CIDR*.

[18] Julian Shun et al. 2013. Practical parallel lempel-ziv factorization. In *2013 Data Compression Conference*. IEEE, 123–132.

[19] Evangelia Sitaridi et al. 2016. Massively-parallel lossless data decompression. In *ICPP*. IEEE, 242–247.

[20] Jianguo Wang et al. 2017. An experimental study of bitmap compression vs. inverted list compression. In *SIGMOD*. 993–1008.

[21] Manni Wang et al. 2021. AsyMo: scalable and efficient deep-learning inference on asymmetric mobile CPUs. In *MobiCom*. 215–228.

[22] Qunsong Zeng et al. 2021. Energy-efficient resource management for federated edge learning with CPU-GPU heterogeneous computing. *IEEE TWC* 20, 12 (2021), 7947–7962.

[23] Shuhao Zhang et al. 2019. Briskstream: Scaling data stream processing on shared-memory multicore architectures. In *SIGMOD*. 705–722.

[24] Shuhao Zhang et al. 2020. Towards Concurrent Stateful Stream Processing on Multicore Processors. In *ICDE*. 1537–1548.

[25] Shuhao Zhang et al. 2021. Parallelizing Intra-Window Join on Multicores: An Experimental Study. In *SIGMOD*. 2089–2101.

[26] Yongluan Zhou et al. 2011. Dissemination of models over time-varying data. *In VLDB* 4, 11 (2011), 864–875.