Parallelizing Stream Compression for IoT Applications on Asymmetric Multicores

Xianzhi Zeng
ISTD Pillar
Singapore University of Technology and Design
Singapore, Singapore
xianzhi_zeng@sutd.edu.sg

Shuhao Zhang
ISTD Pillar
Singapore University of Technology and Design
Singapore, Singapore
shuhao_zhang@sutd.edu.sg

***Abstract*—Data stream compression attracts much attention recently due to the rise of IoT applications. Thanks to the balanced computational power and energy consumption, asymmetric multicores are widely used in IoT devices. This paper introduces *CStream*, a novel framework for parallelizing stream compression on asymmetric multicores to *minimize energy consumption* without violating the user-specified *compressing latency constraint*. Existing works cannot effectively utilize asymmetric multicores for stream compression, primarily due to the non-trivial asymmetric computation and asymmetric communication effects. To this end, *CStream* is developed with the following two novel 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. To validate our proposal, we evaluate *CStream* with five competing mechanisms of parallelizing stream compression algorithms on a recent asymmetric multicore processor. Our extensive experiments based on a benchmark consisting of three algorithms and four datasets show that *CStream* outperforms alternative approaches by up to 53% lower energy consumption without compressing latency constraint violation.**

***Index Terms*—Stream compression, Edge Computing and IoT, Asymmetric Hardware**

I. Introduction

Data stream compression, i.e., continuously compressing data attracts much attention recently [1], [2], [3], [4], [5], [6], [7], due to the rise of IoT applications [8], [9]. Figure 1 demonstrates a smart city use case [10], [11], [12] where stream compression is a highly attractive technique. In this application, real-time data streams (e.g., air qualities, wind speeds) from sensors are continuously gathered by the memory-limited, battery-powered patrol drones (i.e., IoT devices). The drone may continuously compress gathered data streams before uploading to the cloud center to reduce data transmission overhead. However, adopting compression does not guarantee "plug-and-play" performance benefits due to the additional compressing latency and hardware resource constraints such as the battery capacity of IoT devices.

According to a 2018 survey [13], modern ARM machines with asymmetric multicores are typical choices for IoT devices [13]. The key to asymmetric multicores is to couple relatively energy-saving and slower cores (i.e., 'little cores') and relatively more powerful and power-hungry cores (i.e., 'big cores') under the same Instruction Set Architecture (ISA). For instance, an ARM rk3399 processor can be composed with both the in-order A53 'little cores' [14] and the out-of-order A72 [15] 'big cores'. Such a novel asymmetric architecture balances computational power and energy consumption but brings non-trivial asymmetric computation and asymmetric communication effects. The asymmetric computation effect stands for that the computational power of 'big cores' is greater than that of 'little cores', and the asymmetric communication effect stands for that the communication latency from 'big cores' to 'little cores' is larger than the reverse direction.

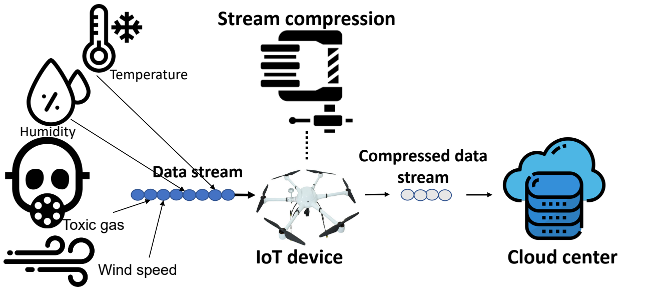


Fig. 1: Real-time data gathering and stream compression at the patrol drone.

In this paper, we propose *CStream*, a novel framework of parallelizing stream compression for IoT applications. Different from file compression or database compression where all data to be compressed are readily presented, stream compression is an incremental procedure of handling continuously arriving data streams. Specifically, a data stream is a list of tuples chronologically arriving at the system, and each tuple needs to be compressed with low latency. When stream compression is conducted in IoT devices, such as in the example in Figure 1, energy consumption is another important factor to be considered. Based on asymmetric multicores, *CStream* parallelizes a stream compression algorithm to compress data streams, such that it *minimizes energy consumption* while satisfying a user-specified *compressing latency constraint*.

Our work is linked to the literature on both parallel data compression algorithms [16], [17], [18], [19], [20], [21], [22], [23] and workload scheduling on asymmetric multicores [24], [25], [26], [27], [28], [29], [30], [31], [32], [33]. They provide highly valuable techniques and mechanisms, but none of them is able to answer the question of "*how to achieve energy efficient and low latency stream compression on asymmetric multicores*". Compared to existing works, the superiority of *CStream* is achieved from two novel designs: *fine-grained decomposition* and *asymmetry-aware task scheduling*.

First, *CStream* decomposes the entire stream compression procedure (i.e., running of a stream compression algorithm on a batch of data stream, Definition 1) into fine-grained tasks with different *operational intensity* (i.e., instructions per memory access) [34], [24], [35], [36]. Tasks with higher (resp. lower) operational intensity prefer 'big cores' (resp. 'little cores'). Such an approach better exposes task-core affinity and offers opportunities for better utilization of asymmetric multicores compared to existing mechanisms [35], [33], [32].

Second, *CStream* schedules the decomposed fine-grained tasks on asymmetric multicores according to their exposed task-core affinity. The involved asymmetric communication overheads require one to carefully align the communication pattern among tasks. To this end, we propose a novel cost model to guide the scheduling by considering both task-core affinity and asymmetric communication effects. Specifically, our model accurately predicts both the energy consumption and compressing latency of each decomposed task given a scheduling plan (Definition 2). Based on the model, *CStream* searches for the optimal scheduling plan by enumerating all possible plans with dynamic programming.

For a comprehensive comparison, we have implemented and evaluated the parallelization of three representative stream compression algorithms in *CStream* based on a recent ARM processor rk3399 with asymmetric multicores. The evaluation based on both real-world and synthetic datasets confirm the superiority of *CStream*. In particular, it outperforms the alternative mechanisms [35], [37] by up to 53% more energy consumption reduction without violating the strict compressing latency constraints of 11 $\sim$ 26 microseconds for compressing each byte of data stream under varying workload characteristics. In summary, this paper makes the following contributions.

- We develop *CStream*, a novel framework for parallelizing various stream compression algorithms on asymmetric multicores to minimize energy consumption while ensuring the compressing latency is within a user-defined constraint. All of our code, data, and scripts can be found at https://github.com/intellistream/CStream. The design overview of *CStream* is presented in Section III.
- We propose a fine-grained decomposition mechanism (Section IV) to decompose a stream compression procedure into fine-grained tasks based on the compression behavior (i.e., varying operational intensity) of a stream compression algorithm. This allows *CStream* to better expose task-core affinities on asymmetric multicores.
- We propose an asymmetry-aware task scheduling mechanism (Section V), which schedules the decomposed tasks on asymmetric multicores based on a novel cost model to exploit the exposed task-core affinities while taking asymmetry communication effects into account.
- To the best of our knowledge, this work is also the first comprehensive study to compare various competing mechanisms to parallelize stream compression algorithms on asymmetric multicores using both real-world and synthetic datasets (Section VI and Section VII).

TABLE I: Summary of terminologies

Type	Notation	Description
Workload Specifications	B	The size of batch of data stream to compress
	L_{set}	User specified compressing latency constraint
Device Specifications & Roofline Model	j	A specific AMP core
	C_j	Maximum executable instructions within unit time of core j
	$L_{j',j}^{comm}$	Worst unit communication latency from core j' to j
Model Outputs	L_{est}	Estimated compressing latency
	E_{est}	Estimated total energy consumption
Cost Model Terms	t_i	Task i decomposed from a stream compression job
	p	A possible scheduling plan
	p_{opt}	The optimal scheduling plan
	i_i	Input data size of t_i
	e_i	Energy consumption of t_i
	l_i	Compressing latency of t_i
	l_i^{comp}	Computation latency of t_i
	l_i^{comm}	Communication latency of t_i
	η_i	Instructions per unit time of t_i
	ζ_i	Instructions per unit energy of t_i
	κ_i	Instructions per unit memory access (i.e., operational intensity) of t_i

II. Preliminaries

In this section, we give a preliminary background of stream compression for IoT applications, followed by reviewing the asymmetric multicore architecture. We summarize the terminologies used in our work in Table I.

A. Data Stream Compression for IoT

A *data stream* is a list of tuples chronologically arriving at the system. Each *tuple* represents an event including timestamp and payloads. *Data stream compression* is an incremental procedure to compress continuously arriving data streams with low latency. It can be conducted solely based on the *current tuple* (i.e., that arrives most recently) or additionally based on the *past tuples* (i.e., those arrive earlier than the current tuple). We classify the former as *stateless stream compression*, which ignores the past tuples, and the latter as *stateful stream compression*, which utilizes a state (e,g., a dictionary [38]) to keep the information of past tuples.

Our work aims to provide framework-level support to optimize stream compression (including stateless and stateful) for IoT applications [9]. In particular, *CStream* parallelizes each stream compression procedure on asymmetric multicores, defined as follows.

Definition 1 (Stream Compression Procedure)**.** *A stream compression procedure is the process of executing a stream* *compression algorithm (stateless or stateful) on a batch of data streams, where the batch size (B) is tunable. In this work, we assume B is pre-determined by applications. To simplify the presentation, we use a pair of* Algorithm-Dataset *to describe a stream compression procedure in the following.*

Note that, the compressibility of a specific compression algorithm is not a concern of this work. Instead, we focus on the following two strict design requirements for adopting stream compression for IoT applications.

Design Requirements of Stream Compression for IoT:

- (R1) Low Latency Stream Compression: Data streams generated from the IoT applications are often real-time constrained, requiring a low latency compressing to meet the quality-of-service (QoS) goal [39], [35].
- (R2) Low Energy Consumption: Stream compression for IoT needs to achieve low energy consumption as the available energy budget for IoT devices is often quite limited compared with that in a data center [8]. For instance, the devices may be solar or battery-powered and far away from a constant power source.

B. Asymmetric Multicore Architecture

The asymmetric multicore architecture is designed to balance computational power and energy consumption [40], [41], [34], and is increasingly deployed for IoT devices [9], especially due to the worldwide rising concern for energy consumption and carbon emissions [42], [43], [44]. Figure 2 depicts a recent 6-core rk3399 processor [45], which couples different types of cores (A53 and A72) on the same chip. Such architecture is also often called “big-Little” architecture.

Compared to conventional symmetric multicores, asymmetric multicores involve two non-trivial asymmetry effects: *1) Asymmetric Computation Effect.* As shown in Figure 2, four A53 cores (i.e., $Core0 \sim Core3$) and two A72 cores (i.e., $Core4 \sim Core5$) are coupled in one chip. Although they share the same ISA (i.e., ARM V8), the A53 cores (‘little cores’) are in-order, relatively battery-saving and slower. In contrast, the A72 cores (‘big cores’) are out-of-order, power-hungry and more powerful; and *2) Asymmetric Communication Effect.* A53 cores and A72 cores are placed in different clusters ($Cluster0 \sim Cluster1$) resulting in different types of cross-core communication patterns [46], [47], i.e., *inter-cluster* and *intra-cluster* communication. The *inter-cluster* communication needs to go through a slow CCI500 interconnection channel, while *intra-cluster* communication involves L2 cache only.

III. Motivation and Design Overview

In this section, we present the motivation for the design of our proposed framework – *CStream* for stream compression on asymmetric multicores, followed by its design overview. The detailed implementation of task decomposition and scheduling with a cost model are presented in Sections IV and V, respectively.

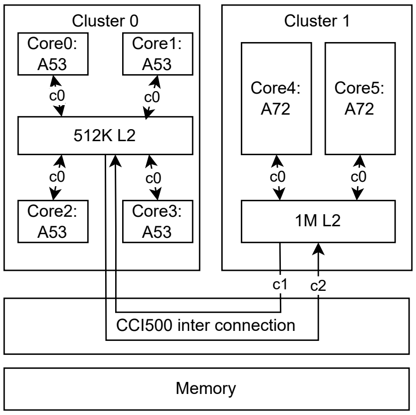


Fig. 2: The 6-core rk3399 processor.

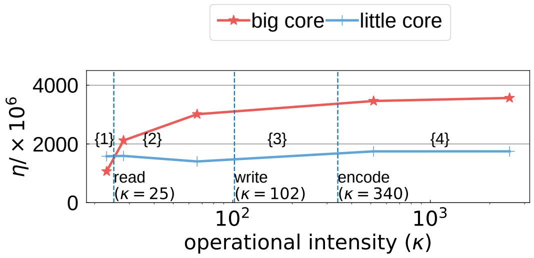


Fig. 3: The four-segment roofline model of asymmetric multicores rk3399. Dashed lines denote the κ of different stream compression steps.

A. Motivations

Parallelizing stream compression on asymmetric multicores can potentially satisfy the aforementioned two design requirements of adopting stream compression for IoT applications. However, the involved asymmetry effects require a careful system design. A poor design can incur both severe compressing latency constraint violation and energy dissipation. It is a particular challenge to support varying stream compression algorithms (e.g., stateless and stateful compression), varying datasets, and varying compressing latency constraints. Our design of *CStream* is motivated by the following observations.

Observation 1: there are varying task-core affinities in different parts of a stream compression procedure. The roofline model [48], [49] reveals that the instructions per unit time (η) or energy (ζ) on hardware grows with the *operational intensity* (i.e., instructions per memory access, denoted as κ) of software, before reaching a maximum value (i.e., the so-called “roof”). By taking the tcomp32 algorithm [50] as an example, we show the roofline model of a ‘big core’ and a ‘little core’ on the rk3399 asymmetric multicores. The roofline profiling is done by adopting a benchmark by Lo et al. [51]. A stream compression procedure is generally composed of three steps ($read$, $encode$, and $write$) with different κ. We mark the κ of each stream compression step as dashed vertical lines in Figure 3. There are two key takeaways. First, the roofline model on asymmetric multicores is more complex than the

TABLE II: Bandwidth and latency of cross-core communication in rk3399

Path	Bandwidth	Latency
intra-cluster $c0$	2.7 GB/s	70.4 ns
inter-cluster $c1$	0.7 GB/s	142.4 ns
inter-cluster $c2$	0.4 GB/s	420.8 ns

original and common assumption of "linear growth" [48]. Specifically, there are four distinct segments marked as $\{1\} \sim \{4\}$ for the rooflines on 'big core' and 'little core', separated by the dotted vertical line. Each segment involves different levels of κ, putting different pressure on the L1 and L2 cache and thus leads to different $\kappa - \eta$ relationships. In particular, η even decreases with increasing κ from 30 to 70 (i.e., the $\{2\}$ segment) on 'little core'. We observe that this is primarily due to the increasing L1-I cache misses. As the 'little core' is an in-order processor, it stalls until instructions become available, severely affecting its performance. Second, we can see that when κ is larger than 25, it is more and more cost-effective (i.e., leads to more performance gain) to run tasks on 'big cores'. Consequently, different steps of the same stream compression procedure should be scheduled independently as they may be better scheduled to different cores due to their different κ as shown in the figure.

Observation 2: there are large differences in communication costs among asymmetric cores. To exploit task-core affinities, decomposed tasks may be scheduled to different cores. Therefore, different core communication paths (i.e., $c0$, $c1$, and $c2$ in Figure 2) may be involved. They have large differences in terms of bandwidth and latency as illustrated in Table II, measured by the STREAM benchmark [52]. The intra-cluster communication has much higher bandwidth and lower latency than inter-cluster communication, due to the slower CCI500 interconnection. More interestingly, inter-cluster communications of different directions (i.e., $c1$ and $c2$) are not involving the same cost. This is because of the additional *synchronization* and *hand-shaking* cycles [53] when sending data from little cores to big cores. Those asymmetric communication effects make the exploiting of task-core affinities a non-trivial quest.

In summary, these observations challenge existing schemes for parallelizing stream compression on asymmetric multicores: First, existing mechanisms [35], [33], [32] consider the coarse-grained scheduling and do not expose the fine-grained task-core affinities in the workload. Second, previous studies on utilizing asymmetric multicores [47], [46] surprisingly overlook the different costs of $c1$ and $c2$, which is important to consider when scheduling decomposed tasks from a stream compression procedure that involves heavy inter-task communications.

B. Design Overview of CStream

Figure 4 depicts the overall workflow of *CStream*. First, *CStream* applies fine-grained decomposition of a stream compression procedure driven by the operational intensities of compression steps (e.g., read, encode, and write) to better expose task-core affinities. The decomposition results in several *tasks*, which can run independently and communicate with each other via message passing. For instance, one task may just conduct the encode step and emit the results to downstream tasks. To increase concurrency, a task may be further replicated to multiple replicas (e.g., t_0 and t_1) handling subsets of data streams. Second, the decomposed tasks are scheduled to asymmetric multicores to minimize total energy consumption (E) without violation of user-specified compressing latency constraint (L_{set}), guided by a novel cost model. The cost model estimates both energy consumption (e_i) and compressing latency (l_i) of each task: 1) the e_i is estimated by the operational intensity (κ_i) of each task, and 2) the l_i is the summation of computation latency (l_i^{comp}) and communication latency (l_i^{comm}) of each task t_i. In particular, l_i^{comm} varies depending on where the task and its upstream tasks are scheduled. For instance, in the example scheduling plan shown in Figure 4, t_2 needs to fetch data from t_0 and t_1 via the slow CCI500 interconnection channel.

IV. Fine-grained Decomposition

In this section, we discuss the fine-grained decomposition of a stream compression procedure in detail.

A. Stream Compression Procedure Templates

We generally classify existing stream compression algorithms into *stateless* and *stateful* categories depending on whether they utilize *states*. *CStream* supports the parallelization of both *stateless* and *stateful* algorithms on asymmetric multicores. In the following, we illustrate the code template of both types of stream compression algorithms. We leave the detailed steps of concrete algorithms used as examples in this work in a technical report [54].

Stateless Stream Compression. Algorithm 1 depicts the high-level idea of how stateless stream compression (e.g., *tcomp32* algorithm [50]) works. It involves three steps for every batch of data streams: $s0$, $s1$, and $s2$. First, it $reads$ a batch of tuples in step $s0$ before compressing. This step is mostly about memory copy, so it has low operational intensity. Second, it $encodes$ each tuple in $s1$ by finding its compressible parts. This step typically involves arithmetic and logical operations in searching compressibility and therefore leads to higher operational intensity than $s0$. Third, it $writes$ the compressed data to the output stream in $s2$ according to what $s1$ has encoded. This step involves both integer/float operation and memory access, and typically has a middle level of operational intensity compared with $s0$ and $s1$.

Stateful Stream Compression. Algorithm 2 depicts the high-level idea of how a stateful stream compression algorithm works (e.g., lz4 algorithm [55]). It involves five steps as $s0 \sim s4$. The read (i.e., $s0$) and write (i.e., $s4$) steps are the same as those in a stateless compression algorithm. In contrast, the encode step is now based on state, and can be further partitioned into three steps: $s1 \sim s3$. First, it preprocesses some values before accessing the state (such as an index of a dictionary [55]) in $s1$. Second, it updates the state in-cache (or

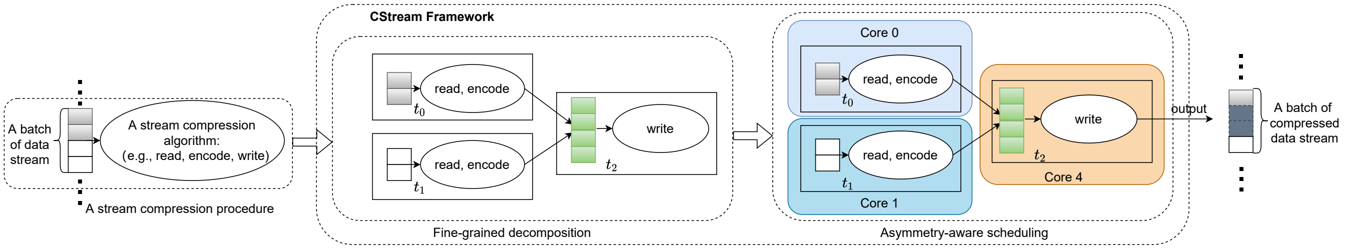

CStream Framework
A batch of data stream
A stream compression algorithm: (e.g., read, encode, write)
A stream compression procedure
read, encode
t_0
read, encode
t_1
write
t_2
Fine-grained decomposition
Core 0
read, encode
t_0
Core 1
read, encode
t_1
write
t_2
Core 4
output
Asymmetry-aware scheduling
A batch of compressed data stream

Fig. 4: The workflow of *CStream*. Using a stateless stream compression procedure as an example.

```
Algorithm 1: Stateless stream compression
  Input: input stream $inData$
  Output: output stream $outData$
  while $inData$ is not stopped do
      (s0) read the tuples from $inData$ ;
      (s1) encode by finding the compressible parts;
      (s2) write compressed data to $outData$ ;
  end
```

```
Algorithm 2: Stateful stream compression
  Input: input stream $inData$
  Output: output stream $outData$
  while $inData$ is not stopped do
      (s0) read the tuples from $inData$ ;
      (s1) preprocess ;
      (s2) state update ;
      (s3) state-based encoding;
      (s4) write compressed data to $outData$ ;
  end
```

in-memory) in $s2$ with 1) the current tuple and 2) $s1$-produced value. Third, it finally achieves encoding by state reference in $s3$. Due to the state write or read, $s2$ and $s3$ lead to lower operational intensity than $s1$.

B. Parallelizing Stream Compression Procedures

CStream explores both pipelining and data parallelism for parallelizing a stream compression procedure as follows.

Exploring Pipelining Parallelism. First, *CStream* achieves pipelining parallelism by executing the aforementioned stream compression steps (Section IV-A) in a pipeline fashion. Specifically, each step in Algorithm 1 and 2 can run concurrently as an independent task. In this way, their varying operational intensity is exposed for further exploitation. When communication latency (l_i^{comm}) of a task t_i is greater than the computation latency (l_i^{comp}) of this task or that ($l_{i'}^{comp}$) of its upstream task (t_i'), we fuse the tasks t_i and $t_{i'}$ to reduce communication overheads. For example, read ($s0$) and encode ($s1$) are fused, while write ($s2$) runs separately as shown in Figure 4.

Exploring Data Parallelism. Second, when compressing latency constraints L_{set} is unable to meet through pipelining solely, we can replicate tasks to further explore data parallelism. We follow topologically sorted iterative scaling optimization [56] to replicate the bottleneck task. For each iteration, the bottleneck task t_i with the highest compressing latency l_i is replicated. Task replication iteration ends when compressing latency constraints can be met or hardware resources are saturated. We discuss the estimation of l_i to identify bottleneck tasks under varying scheduling plans in the next section.

The replication of all steps of both types of algorithms is straightforward, except $s2$ of Algorithm 2, as it requires the manipulation of states. We let each thread maintain its own state to avoid concurrent access conflicts during stream processing [57]. Compared to sharing states with locks, this approach leads to a slight reduction in compressibility (0.03 lower *compression ratio* [1], [58]), but significantly lower (51%) energy consumption and lower (82%) compressing latency. More details can be found in our report [54].

V. Asymmetry-aware Scheduling

In this section, we introduce how *CStream* decides the optimal scheduling plan with the guidance of a novel cost model. We first give formal definitions of the scheduling problem, then illustrate the design of the cost model, followed by the procedures of model-guided task scheduling.

A. Problem Formulation

The goal of our scheduling is to minimize energy consumption while ensuring the user-specified compressing latency constraint in parallelizing stream compression on asymmetric multicores. Specifically, we define a scheduling plan as follows.

Definition 2 (Scheduling Plan)**.** *A scheduling plan is for mapping each decomposed task t_i to a specific core of asymmetric multicores. Given n tasks decomposed (i.e., $\{t_0, .., t_i, ..., t_{n-1}\}$), there is an n-element array $p = \{j_0, ..., j_i, ..., j_{n-1}\}$ to describe a scheduling plan and j_i is the specific core of t_i to be mapped.*

We look for the optimal scheduling plan p_{opt} for a given stream compression procedure. The problem can be mathematically formulated as Equation 1∼3.

$$minimize(E_{est} = \sum_i e_i) \tag{1}$$

s.t., $\forall t_i \forall j$,

$$L_{set} \geq L_{est} = \max(l_i) = \max(l_i^{comp} + l_i^{comm}), \tag{2}$$

$$C_j \geq \sum_{t_i \text{ at core } j} \eta_i \tag{3}$$

We refer to the energy consumption of each task t_i using the symbol e_i, and its compressing latency as l_i. Minimizing total energy consumption ($E_{est} = \sum_i e_i$) in Equation 1 is the major reason for adopting asymmetric multicores in IoT. As the formulas show, we consider two categories of constraints that the optimization algorithm needs to make sure the scheduling plan satisfies. Constraint in Equation 2 enforces that compression latency should never exceed the user-specified constraint, due to real-time restrictions in stream analysis [8]. Due to the pipeline execution, the estimated compressing latency L_{est} is constrained by the maximum latency of each task, i.e., $L_{est} = \max(l_i)$. The l_i can be further decomposed into computation latency (l_i^{comp}) and communication latency (l_i^{comm}) as $l_i = l_i^{comp} + l_i^{comm}$. Constraint in Equation 3 enforces that the aggregated demand of instructions requested to any core ($\sum_{t_i \text{ at core j}} \eta_i$) must be smaller than its computation capacity (C_j). Task oversubscription has been studied in previous work [59], and is not the focus of this paper.

B. Cost Model

Our model estimates the energy consumption (e_i), instructions per unit time and energy (η_i and ζ_i, respectively), and compressing latency (l_i) of each task.

1) Estimation of e_i: The estimation of e_i is non-trivial as various properties of data and algorithms are involved. As shown in Equation 4, we estimate e_i as a proportional relationship to the instructions per unit time (η_i) and the latency (l_i), and an inverse proportional relationship to the instructions per unit energy (ζ_i).

$$e_i = \frac{\eta_i \times l_i}{\zeta_i} \tag{4}$$

2) Estimation of η_i and ζ_i: The instructions of unit time (η_i) is an intrinsic property of the asymmetric multicores. According to observations in Section III-A (especially the Figure 3), we use operational intensity (κ_i) to estimate η_i, by refining and formulating the aforementioned roofline graph [48]. Specifically, we estimate η_i as a four-region piece-wise linear function to κ_i, with L1 and L2 cache-awareness [60] for better modeling on asymmetric multicores, as shown in Equation 5.

Profiling of κ_i: Due to the single ISA property of asymmetric multicores, the operational intensity (κ_i) of each task t_i is not changing under varying scheduling plans. By definition, κ_i is related to *the number of instructions* and *memory accesses* of t_i. We feed each task t_i with a moderate size of data (which is large enough to prevent randomness and also within the memory capacity), and then profile the total number of instructions with *perf* [61]. The memory access is statically analyzed by the specific step of the stream compression algorithm.

Estimation of η_i: We can estimate η_i by using κ_i in Equation 5, and there are four segments as shown previously in Figure 3. First, when κ_i is low and does not put significant pressure on L1D (i.e., within the first boundary κ_{L1}), η_i grows relatively fast with a growth rate a_{L1} and intercept b_{L1}. Second, condition $\kappa_{L1} < \kappa_i <= \kappa_{L2}$ means a higher κ_i that increases L1D missing and has significant impacts on the core. In this case, η_i grows slower with growth rate a_{L2} and intercept b_{L2}. Third, with κ_i increasing into $\kappa_{L2} < \kappa_i <= \kappa_{roof}$, the L2 missing has major effects and therefore changes the a_{L2} and b_{L2} into $a_{ExceedL2}$ and $b_{ExceedL2}$, respectively. Fourth, after κ_i reaches the last boundary κ_{roof}, the η_i stays at the maximum value η_{max} instead of increasing with κ_i. When running on a certain core j, η_{max} is equal to its computational capacity (C_j). The specific value of the aforementioned parameters varies in 'big cores' and 'little cores' due to asymmetric computation effects, and we can use piece-wise linear fitting [62], [63] on the list of κ and its resulting η to acquire them.

$$\eta_i = \begin{cases} \kappa_i * a_{L1} + b_{L1} & , \kappa_i <= \kappa_{L1} \\ \kappa_i * a_{L2} + b_{L2} & , \kappa_{L1} < \kappa_i <= \kappa_{L2} \\ \kappa_i * a_{ExceedL2} + b_{ExceedL2} & , \kappa_{L2} < \kappa_i <= \kappa_{roof} \\ \eta_{max} & , \kappa_i > \kappa_{roof} \end{cases} \tag{5}$$

Estimation of ζ_i: Another intrinsic property of asymmetric multicores, i.e., instructions of unit energy consumption (ζ_i) can also be estimated in a piece-wise linear form like Equation 5. Typically, the estimation of ζ_i involves different parameter values including the boundary of regions (i.e., κ_{L1} and κ_{L2}), the growth rate (i.e., a), and the intercept (i.e., b).

3) Estimation of l_i: The compression latency (l_i) of a task t_i is the sum of two non-overlapping components l_i^{comm} and l_i^{comp}.

Estimation of l_i^{comp}: We use l_i^{comp} to denote the general computation time spent for executing the task t_i. For tasks that have a constant workload characteristic, it is simply determined by the input size (i.e., i_i), and the mapped core j_i. As a result, Equation 6 depicts the simple linear relationship used to estimate it, with a system overhead ω_{j_i} of the mapped core j_i and a growth rate λ that are constant.

$$l_i^{comp} = \lambda \times i_i + \omega_{j_i}, \tag{6}$$

l_i^{comp} (under both big core and little core) can be acquired by a dry-run profiling. We can also use machine learning techniques (e.g., logistic regression) to train a prediction model to predict l_i^{comp}.

Estimation of l_i^{comm}: Task t_i involves communication delay l_i^{comm} to fetch the data from its upstream task t_u. l_i^{comm} is determined by the fetched data size (i.e., the i_i) and the relative distance to upstream task t_u. If a scheduling plan p maps t_i and t_u respectively at core j_i and j_u, l_i^{comm} can be estimated by Equation 7.

$$l_i^{comm} = \begin{cases} 0, \text{if } j_i = j_i' \\ \frac{i_i \times L^{comm}_{j_{i'},j_i}}{\text{cache line size}} + \omega_{j_{i'},j_i}, \text{otherwise} \end{cases}$$
$$\text{, where } j_i \text{ and } j_{i'} \text{ are determined by } p. \qquad (7)$$

When task t_i is collocated with its upstream $t_{i'}$, or t_i is just at the beginning step (i.e., *read*), the communication latency l_i^{comm} is 0. Otherwise, it experiences the cross-core ($j_i \neq j_{i'}$) communication as discussed in Section III-A. Formula 7 estimates the communication latency based on the total size of data to be transferred (i_i), cache line size, static overhead ($\omega_{j_{i'},j_i}$) between core j_i and $j_{i'}$, and the worst unit communication latency ($L^{comm}_{j_{i'},j_j}$) between core j_i and $j_{i'}$. It is worth noting that $L^{comm}_{j_{i'},j_j} \neq L^{comm}_{j_i,j_{j'}}$ and $\omega_{j_{i'},j_j} \neq \omega_{j_i,j_{j'}}$ if core j_i and $j_{i'}$ are located in different clusters, due to the previous observations from Table II. For each possible j_i and $j_{i'}$, their $L^{comm}_{j_{i'},j_i}$ and $\omega_{j_{i'},j_i}$ can be dry-run measured by setting up a producer thread at $j_{i'}$ and a consumer thread at j_i. Note that, the accurate estimation of l_i^{comm} is difficult as the i_i may vary depending on the compressibility. In this work, we assume that i_i does not vary much in a short period of time, given the same dataset and the same compression algorithm.

C. Model-guided Scheduling

We search p_{opt} by enumerating all possible plans with the cost model. We adopt dynamic programming [64] to speed up the plan searching as different plans may overlap for the scheduling of subsets of tasks. For each plan p_{enum} enumerated, we first predict the compressing latency (l_i) and instruction per unit time (η_i) on all of its tasks (t_i), according to Equations 5 $\sim$ 6, and then check whether constraints in Equations 2 and 3 are met. If so, we continue to predict energy consumption(e_i) of all t_i by Equation 4 and get the total estimated energy consumption E_{est} of p_{enum}; if not, we just ignore p_{enum} and continue the search. The plan with minimal E_{est} and meet all constraints is p_{opt}.

D. Adaptive to Dynamic Environment

Our model is initially instantiated with a small number of input data (10 $\sim$ 100 batches). However, data stream characteristics, such as data size and entropy, can vary over time, and *CStream* needs to be re-optimized in response to workload changes. To adapt to dynamic scenarios, we adopt a feedback-based regulation [65] in *CStream*. Specifically, we periodically (i.e., every 50 ms) measure the compressing latency and its predicted value. If the difference is larger than a threshold, we collect the subsequent batches of data to calibrate the cost model. We base our model calibration on PID control [66], [67], [68], which is the most common form of feedback control. More details can be found in our report [54]. The scheduling is then replanned using the updated model by gradually migrating the current plan to the optimal one. The overhead of performing such feedback-based dynamic regulation is negligible as our cost model involves solving simple linear equations and rescheduling is incrementally conducted. However, its response may be lagged when facing a bursting workload. More sophisticated controllers [69] that monitor workload statistical information in the datastream may achieve an even better response to workload changes but beyond the scope of this work.

VI. Methodology

In this section, we first introduce the examined competing mechanisms, followed by benchmark workloads including both varying compression algorithms and datasets. Then, we discuss the instrument of targeting performance metrics.

A. Competing Mechanisms

We compare *CStream* with the following five competing mechanisms in parallelizing the stream compression procedures: OS, CS, RR, BO, and LO as follows: 1) **Operating System** (*OS*). The replicated tasks of the whole stream compression procedure (without decomposition) are scheduled by the Linux 5.10 kernel at thread level with the *energy-aware-scheduling (EAS)* [70] strategy. 2) **Coarse-grained Scheduling** (*CS*)**.** Following prior work of coarse-grained workload scheduling on asymmetric multicores [32], we can schedule each replica of the entire stream compression procedure as a single task with our asymmetry-aware scheduling scheme. 3) **Round Robin** (RR). Under RR, we apply fine-grained decomposition of *CStream*, and the decomposed tasks are scheduled in a round-robin manner, i.e., sequentially mapped to each core. 4) **Big-core Only** (BO). Under BO, the decomposed tasks are randomly scheduled to the big cores of rk3399 (core4 to core5), and little cores (core0 to core3) are left idle. 5) **Little-core Only** (LO). Under LO, the decomposed tasks are randomly scheduled to the little cores of rk3399 (core0 to core3), and big cores (core4 to core5) are left idle.

Under OS and CS, the stream compression procedure is replicated into multiple tasks in achieving data parallelism (without decomposition and pipelining parallelism). The tasks are subsequently scheduled by the Linux kernel under OS, or with asymmetry-aware scheduling of *CStream* under CS. Under RR, BO, and LO, the stream compression procedure is decomposed in a fine-grained manner as *CStream* does, but task scheduling is not asymmetry-aware.

B. Input Workloads

CStream supports varying stream compression algorithms and datasets. We select a wide range of algorithms and datasets with distinct characteristics for a comprehensive evaluation.

1) Algorithms: We focus on parallelizing the following three lightweight stream compression algorithms in our evaluation. Nevertheless, *CStream* can be easily extended to support other stream compression algorithms. *1) tcomp32* cuts off unused bits of each 32-bit symbol, which is a stateless stream compression following Algorithm 1's abstraction. *2)* *lz4* [55] is a popular LZ77-based [38] stateful compression (i.e., Algorithm 2). It uses a hash table as its state to replace the traditional *dictionary* of LZ77. *3) tdic32* is a simplified variable length coding created by combining the two above, which is also stateful. Specifically, it borrows the hash table from lz4 and employs a memory I/O pattern similar to tcomp32 (i.e., byte-unaligned encoding for each 32-bit single symbol).

2) Datasets: We use three real-world and one carefully designed synthetic dataset in our evaluation. These datasets cover varying statistical properties, such as 1) *vocabulary duplication*s [38], 2) *dynamic range of the symbol* [58], [50], and 3) *symbol entropy* [19]. Since the tcomp32, lz4, and tdic32 share a 32-bit reading of data, we define data within 32-bit as a *symbol*, while more than 32-bit of data is treated as a *vocabulary*. The datasets are as follows. *1) Sensor* [71] represents a type of full-text streaming data that is generated by automated sensors. Its most compressible part is the *symbol entropy*, which is packed in an XML format with only ASCIL code. Furthermore, the XML pattern can result in partial *vocabulary duplication*. We let every 16 ASCIL characters in Sensor form one 128-bit tuple in our evaluation. *2) Rovio* [72] continuously monitors the user actions of a given game to ensure that their services work as expected, and is packed in $(64-bit\ key, 64-bit\ payload)$. Its high key duplication leads to significant *vocabulary duplication* [73]. *3) Stock* [74] is a real-world stock exchange dataset packed in $(32-bit\ key, 32-bit\ payload)$ binary format. Unlike Rovio, its key duplication is much lower. *4) Micro* is a synthetic dataset used to easily evaluate the impact of varying workload properties. Each tuple in Micro is a 32-bit plain value.

C. Instrument of Performance Metrics

Throughout this study, we focus on two important performance metrics. The first is *compressing latency constraint violation* ($CLCV$ for short). For each test, we repeat the measurement 100 times, and $CLCV$ refers to the fraction of measurements violating L_{set} and the total measurements. The second is *energy consumption*, denoted as E_{mes}. We let E_{mes} refer to the overhead for compressing each unit of data (i.e., in $\mu J/byte$). The system overhead of each mechanism, such as profiling and scheduling in *CStream* are included in E_{mes}. We have further developed an energy meter that provides accurate measurement with low overhead. More details can be found in our report [54].

VII. Evaluation

We use the Radxa Rockpi 4a [75] for evaluation. This platform is equipped with an rk3399 asymmetric multicores processor, and please refer to our technical report [54] for its detailed specifications. By default, each core runs at its highest frequency (i.e., 1.8GHz for A72, and 1.416GHz for A53). To eliminate the impact of network transmission overhead during input data feeding, the input datasets are first populated (synthetic dataset) or loaded (real datasets) in memory. We set the batch size (B) as 932,800 bytes and L_{set} as $26\mu s/byte$ unless otherwise stated.

A. End-to-End Comparison

In this section, we show the end-to-end comparison between *CStream* and five competing mechanisms.

Energy Consumption Comparison. Figure 5 reports the energy consumption of different mechanisms on handling different datasets. In general, we can see that *CStream* always leads to the least energy consumption. For example, *CStream* can save up to 53.23% energy consumption on the lz4-Stock procedure compared with BO. *CStream* determines the core mapping of decomposed stream compression tasks according to their varying operational intensity. In contrast, LO and BO underutilized either the big cores or little cores on asymmetric multicores, while operational-intensity-unconscious parallelization (i.e., OS), or coarse-grained parallelization (i.e., CS) fails to explore the suitable task-core mapping between stream compression procedure and asymmetric multicores.

Compressing Latency Constraint Violation Comparison. Figure 6 presents the $CLCV$ of different mechanisms on handling different datasets. We can see that *CStream* can avoid any compressing latency constraint violations for the evaluated workloads even under such a strict latency constraint. We find that it is mainly because the statistical characteristics of datasets in our experiments are relatively stable over time. We show the evaluation of more dynamic workloads shortly later. In contrast, LO and RR fail to preserve the compressing latency constraints due to the low utilization of high-performance 'big cores' in conducting high operational intensity steps (e.g., the $s2$ in Algorithm 1) of the stream compression procedure. On the contrary, CS and BO underutilize 'little cores' in conducting low operational intensity steps (e.g., the $s0$ in Algorithm 1). In particular, CS tries to firstly schedule as much as possible to 'big cores' before utilizing the 'little cores', as the operational intensity of the whole stream compression procedure is relatively high (e.g., about 200 for tcomp32-Rovio). OS fails for two reasons. First, its frequent migration of tasks leads to extra overhead. Our further investigation reveals that OS scheduler involves about 60,000 context switches while *CStream* only involves about 10 context switches for compressing every megabyte of input data. Second, OS treats stream compression tasks as a black box, and it hence leads to a relatively inaccurate estimating of latency and causes compressing latency constraint violations.

Dynamic Workloads. The stream properties (e.g., entropy or dynamic range of symbols) may change on the fly. We now evaluate the adaptivity of *CStream* to the dynamic workload by using the tcomp32-Micro procedure. We set the dynamic range of symbols to 500 at the beginning, and increase it to 50000 immediately after the whole fifth batch is compressed. The energy consumption and compressing latency constraint violation of *CStream* with and without feedback-based regulation (Section V-D) are demonstrated in Figure 7. Workload changes after the fifth batch and adaptation finishes at the ninth batch. We observe that the compressing latency

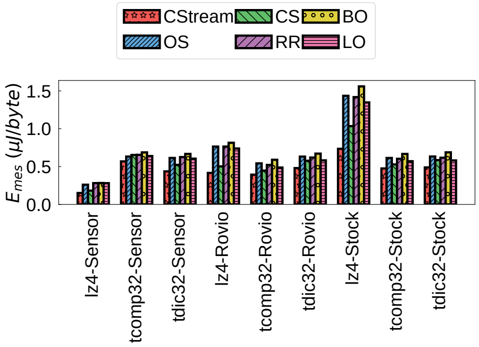


Fig. 5: Energy consumption comparison.

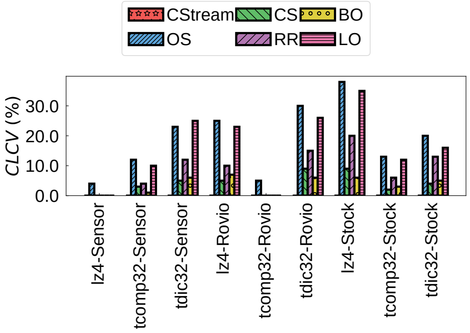
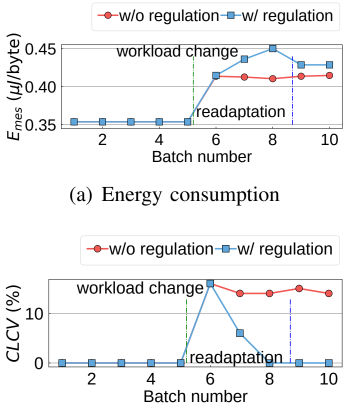


Fig. 6: Compressing latency constraint violation comparison. Note that, *CStream* introduces zero violation.



(a) Energy consumption

(b) Compressing latency constraint violation

Fig. 7: Adaptation to dynamic workload.

constraint will be violated after workload change if there is no feedback-based regulation, as the previous profiling leads to an inaccurate estimation of compressing latency. With the feedback-based regulation, *CStream* is able to adapt to the changing workload eventually from the ninth batch and switch to the new optimal scheduling plan, which requires higher energy consumption to avoid compressing latency violation. There are some vibrations during the adapting process, such as high energy consumption at the eighth batch, due to the continuous calibration process with the PID-based controller (Section V-D). It is worth noting that the overhead of such an adapting process is marginal, i.e., 0.9% energy consumption and 6% processing time of compressing each batch of data.

B. Workload Sensitivity Study

In this section, we show the superiority of *CStream* under varying workload characteristics, including procedure settings and data statistic properties.

1) Procedure Settings: We vary the compressing latency constraint (L_{set}) and batch size (B) of tcomp32-Rovio stream compression procedure as follows.

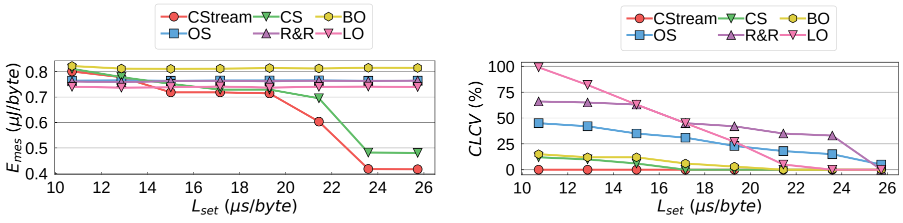
Varying Compressing Latency Constraint (L_{set}). We show the impact of varying compressing latency constraints shown in Figure 8. While OS, RR, BO, and LO have constant energy consumption, *CStream* and CS achieve more on energy saving under a larger L_{set}, and generate a lower latency scheduling plan under a smaller L_{set}. However, the coarse-grained way of CS leads to 1) higher energy consumption at each L_{set} when comparing with *CStream* and 2) underutilization of 'little cores' and failure of meeting the tight settings (i.e., smaller) of L_{set}.

Varying Batch Size (B). We study the energy consumption under different sizes of batch (i.e., B) in Figure 9. We observe that the energy consumption remains nearly stable when there is a large enough batching of data (i.e., $B > 10^3 byte$). A small batch size may increase energy consumption slightly due to the constant system overheads such as context switching and OS system calls [73]. In general, energy consumption is determined $\frac{\eta_i}{\zeta_i}$ proportion (Equation 4) of each task t_i, and both η_i (instructions per unit time) and ζ_i (instructions per unit energy) are determined by the task's operational intensity (κ_i). *CStream* can select the minimal value of $\frac{\eta_i}{\zeta_i}$ for each task to minimize energy consumption, by utilizing 1) the fine-grained decomposed operational intensity (κ_i) when compared with CS, and 2) the asymmetry-aware scheduling when comparing with OS, RR, LO and BO.

2) Data Statistic Properties: We tune the data statistic properties including vocabulary duplication, symbol duplication, and dynamic range. Specifically, we use the synthetic dataset Micro to study their impacts.

Vocabulary Duplication. We first conduct lz4-Micro procedure under varying vocabulary duplication. The lz4 is most sensitive to vocabulary duplication, and operational intensity (κ_i) of lz4 tasks changes differently with vocabulary duplication due to their different functions. For instance, the κ_i of tasks conducting *state update* ($s2$ in Algorithm 2) decreases with increasing vocabulary duplication, as the hash table of state can be updated less. However, the κ_i of tasks conducting *state-based encoding* ($s3$ in Algorithm 2) will increase, as lz4 is more likely to conduct 'backward searching' for expanding match [55]. In Figure 10, we vary the vocabulary duplication from low to high and have two observations. First, *CStream* is able to decompose the lz4 procedure and select the best scheduling plan of tasks no matter how vocabulary duplication changes. Second, each mechanism has its highest energy consumption when duplication is moderate, this is because the aforementioned different tendencies are reconciled under moderate duplication and lead to total maximum energy consumption.

Symbol Duplication. We next conduct the tdic32-Micro procedure while varying the duplication of the 32-bit symbol. The tdic32 is most sensitive to symbol duplication, and the operational intensity (κ_i) of most tdic32 tasks decreases with the increasing symbol duplication. While tasks conducting $s0$



(a) Energy consumption

(b) Compressing latency constraint violation

Fig. 8: Varying L_{set}.

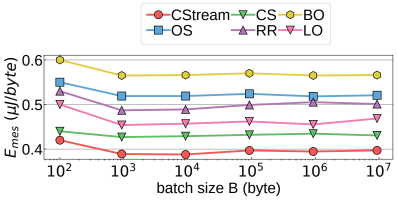


Fig. 9: Varying batch size B.

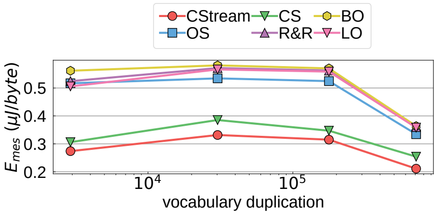


Fig. 10: Varying vocabulary duplication.

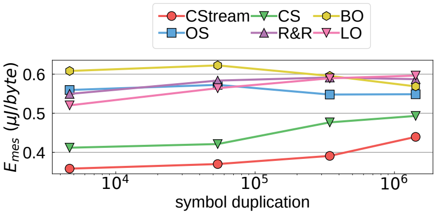


Fig. 11: Varying symbol duplication.

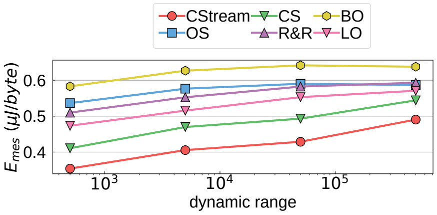


Fig. 12: Varying dynamic range.

and $s1$ remain nearly the same, higher symbol duplication leads to 1) less state update for tasks conducting $s2$ and 2) shorter average encoding and writing for tasks about $s3$ and $s4$. The results of parallelizing tdic32 are shown in Figure 11. We can see that LO has an increasing energy consumption with increasing symbol duplication. This is because when symbol duplication increases, more tdic32 tasks have their operational intensity κ_i dropped to the $30 \sim 70$ region, which suffers from the in-order stall in L1-I cache as depicted in Figure 3. In contrast, BO becomes more energy efficient with increasing symbol duplication as 'big cores' are out-of-order. *CStream*, OS, CS, and RR utilize both big and little cores, and their different utilization on the asymmetric multicores leads to different energy consumption at changing symbol duplication. Nevertheless, *CStream* is always able to achieve the least energy consumption, which reaffirms its superiority compared to competing mechanisms.

Dynamic Range. Finally, the results of tcomp32-Micro procedure under the varying dynamic ranges of the 32-bit symbol, are shown in Figure 12. The tcomp32 is most sensitive to dynamic range, as increasing dynamic range makes its arithmetic computation in $s1$ and writing output data in $s2$ more costly. Therefore, operational intensity (κ_i) and compressing latency (l_i) of most tasks t_i in the procedure are increased, resulting in a higher energy consumption according to Equations 4. *CStream* always outperforms others, but its energy saving becomes less significant when the dynamic range is high, as there is less room for optimization.

C. System Configuration Analysis

In this section, we evaluate the system sensitivity by tuning core frequency statically and dynamically. The tcomp32-Rovio procedure is used as an illustration example.

Static Frequency Regulation. We vary core frequency statically and evaluate how it affects the measured compressing energy consumption (E_{mes}).

TABLE III: Model correctness under optimal scheduling plans.

Estimation object	method variable	lz4	tcomp32	tdic32
Compressing Latency	$L_{est}(\mu s/byte)$	25.5	23.2	23.3
	$L_{pro}(\mu s/byte)$	23.6	21.7	25.3
	$relative\_error^L$	0.08	0.07	0.08
Energy Consumption	$E_{est}(\mu J/byte)$	0.47	0.43	0.44
	$E_{pro}(\mu J/byte)$	0.42	0.40	0.48
	$relative\_error^E$	0.14	0.08	0.09

The results are shown in Figure 13. Obviously, the *CStream* outperforms other mechanisms under varying frequency settings. It is also worth noting that low frequency doesn't imply lower energy consumption. Although lower frequency results in low power (i.e., measured in Watts), their increased latency in stream compression may lead to even higher energy consumption, especially when tasks run on 'little cores'.

Dynamic Frequency Regulation. We also consider dynamically regulating the frequency by using the DVFS [76], [77], [78]. we report how each of the six mechanisms cooperates with different DVFS strategies in Figure 14. We use the "default" strategy for comparison by fixing each core at its highest frequency. The "conservative" and "on-demand" are two DVFS methods trying to reduce energy consumption by frequency reconfiguration on the fly, and their major difference is that the "conservative" strategy changes frequency less when compared with "on-demand".

We have three observations here. First, *CStream* always achieves the least energy consumption and least compressing latency constraint, regardless of using what kind of DVFS strategy. Second, the conservative" strategy can further reduce energy consumption for all mechanisms compared with their default conditions. However, the compressing latency constraint violation for all mechanisms is increased by using such a strategy. This is because the "conservative" DVFS only offers *relative coarse-grained guarantee of meeting the compressing latency constraints*. Specifically, there is

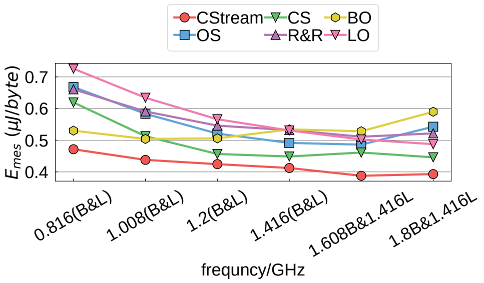
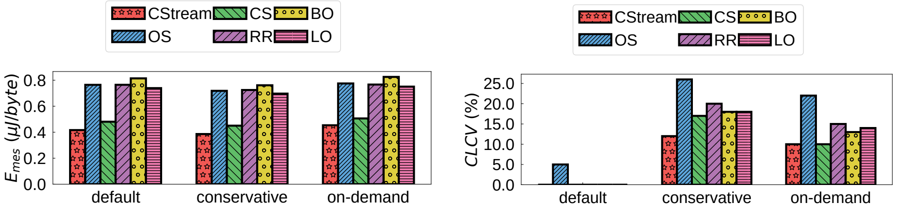


Fig. 13: Impacts of statically varying core frequency. "B" denotes the big cores while "L" denotes the little cores.



(a) Energy consumption (E_{mes})

(b) Compressing latency constraint violation ($CLCV$)

Fig. 14: Impacts of DVFS strategies.

TABLE IV: Comparison among decomposed tasks

Task	operational intensity κ	compressing latency l ($\mu s/byte$)		energy consumption e ($\mu J/byte$)	
		big core	little core	big core	little core
t_0	320	15.0	32.6	0.29	0.27
t_1	102	13.5	21.7	0.32	0.10
t_{all}	220	28.3	53.2	0.59	0.34
t_{re}×2	220	15.0	27.1	0.75	0.51

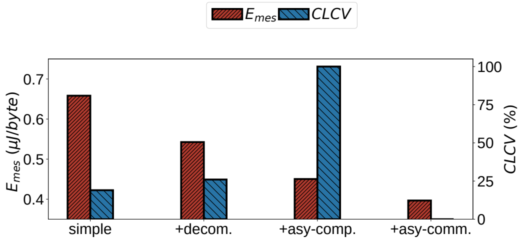


Fig. 15: Factor analysis about energy dissipation consumption (E_{mes}) and compressing latency constraint violation ($CLCV$).

uncertain extra overhead in dynamic frequency regulation, leading to more compressing latency constraint violations. Third, the "on-demand" strategy leads to no improvement but more compressing latency constraint violation and energy consumption. This is because it changes the frequency too often and involves much extra overhead in the frequency switching.

D. Break-down Analysis

Since *CStream* incorporates fine-grained decomposition and asymmetry-ware (i.e., both computation and communication) task scheduling as two key contributions, we conduct a break-down analysis to study their impacts respectively. We use tcomp32-Rovio procedure as the illustration example, and the following break-down factors will be studied.

- *simple* refers to following a symmetric-multicore-aware parallel data compression [79], which only exploits data parallelism. Specifically, the whole procedure is treated as a task t_{all}. When a single t_{all} fails to meet the compressing latency constraint, it is replicated into multiple equivalent tasks (each one is denoted as a t_{re}).
- *+decom.* adds the fine-grained decomposition on stream compression and enables the basic exposition of task-core affinity as discussed in Section IV. Specifically, the procedure is decomposed into two tasks namely t_0 (conducting $s0$ and $s1$ in Algorithm 1) and t_1 (conducting $s2$ in Algorithm 1). Both t_0 and t_1 will be randomly scheduled to asymmetric multicores.
- *+asy-comp.* adds the asymmetric-computation-awareness to schedule t_0 and t_1, including all modeling in Section V-B but ignoring the asymmetric communication effects. Specifically, the unit communication latency $L^{comm}_{j_{i'},j_i}$ and $L^{comm}_{j_i,j_{i'}}$ in Equation 7 are treated the same for any core $j_{i'}$ and j_i that non-equal.
- *+asy-comm.* adds the consideration of asymmetric communication to schedule t_0 and t_1, and is the fully functional *CStream* as introduced in Section III.

Impacts of Fine-grained Decomposition. By comparing *+decom.* with *simple*, we found that the fine-grained decomposition provides opportunities for better utilization of both big cores and little cores, which can reduce energy consumption a lot. To further comprehend the effects of decomposition, we compare the operational intensity (κ), compressing latency (l), and energy consumption (e) of the decomposed tasks t_0/t_1 , single thread all procedure (t_{all}), and the replicated version of t_{all} (i.e., $t_{re} \times 2$) in Table IV. There are three takeaways. First, t_0 is better to be scheduled to 'big cores' than other tasks due to its highest operational intensity. Specifically, it can reduce 53% compressing latency when scheduled to a 'big core' instead of a 'little core', with only 8% increased energy consumption. Second, t_{all} and $t_{re} \times 2$ lead to much underutilization of asymmetric multicores, as they simply *reconcile* the operational intensity of t_0 (i.e., 340) and t_1 (i.e., 102) to a medium value 220, ignoring the fact that they have large difference and should be treated differently. Third, only applying *+decom.* is far from optimal. Specifically, both t_0 and t_1 should be scheduled to the "right place" (i.e., big core and little core respectively as shown in table) in achieving energy saving or compressing latency reduction, but *+decomp.* schedules them randomly to asymmetric multicores.

Impacts of Asymmetry-aware Scheduling. The asymmetry-aware scheduling includes the awareness of both asymmetric computation and asymmetric communication, and we can observe their impacts respectively from *+asy-comp.* and *+asy-comm.* First, *+asy-comp.* adds asymmetric computation awareness to guide the scheduling of fine-grained tasks t_0 and t_1. It can always correctly determine the varying task-core affinity among them according to their different operational intensity (Table IV), which reduces energy consumption. Nevertheless, due to the ignorance of asymmetric communication effects, it is too aggressive for energy saving and violates the latency constraint (L_{set}) frequently. Second, *CStream* further adds asymmetric communication awareness (i.e., *+asy-comm.*) compared to *+asy-comp*. It can model and plan well before executing t_0 and t_1, and thus prevent the compressing latency constraint violation while ensuring the least energy consumption.

As asymmetry-aware scheduling is guided by the cost model, we evaluate its correctness here. We show the relative error rate associated with estimated compressing latency (L_{est}) or energy consumption (E_{est}) per procedure by our model. Specifically, we define $relative\_error^L = \frac{|L_{pro}-L_{est}|}{L_{pro}}$ and $relative\_error^E = \frac{|E_{pro}-E_{est}|}{E_{pro}}$, where L_{pro} and E_{pro} are the measured compressing latency and energy consumption of the tested procedure, respectively. Table III shows the model accuracy of all evaluated algorithms under their optimal scheduling plans in compressing Rovio. Overall, our estimation approximates the measurement well for the L_{est} and E_{est} of all three algorithms. It is therefore able to determine the optimal scheduling plan as shown previously. The inaccuracy is mainly caused by the difficulty in accurately estimating the unit overhead of communication ($L_{j',j}^{comm}$) on the fly, as it is affected by multiple factors such as memory access patterns and hardware prefetcher units.

VIII. Related Work

In this section, we review the related work and reveal the limitations that motivate this work.

Parallel Data Compression. A lot of compression algorithms have been proposed since 1950s [80], focusing on improving the theoretical compressibility [16] and reducing compressing complexity [18], [19]. On the FPGA, Milward et al. [81] implemented a novel dictionary-based parallel compression implementation, Sano et al. [82] achieved float-point data compression, Tian et al. [83] proposed parallel compression for scientifical data, and Bark et al. [22] parallelized the lz4 algorithm. On the GPU, the parallelization of several LZ algorithms [38] have been conducted such as [84] and [85], and Huang et al. [86] have also parallelized trajectory-specific compression algorithms. Due to the significant difference in hardware architectures, it is unclear how can those works be applied to parallel compression on asymmetric multicores. Researchers have also utilized the Symmetric Multicore Processors (SMPs) for parallel compression: to compress the float-point data [87] and to drill the inner parallelism (in a SIMD-like manner) of LZ77 [88]. Recently, Knorr et al. [89] achieved high throughput of large volumes of scientific data scaling up to 24 threads, and Dua et al. [90] compressed the hyperspectral images on a supercomputer with a hyper-cube structure. Our work differs significantly from existing works in three aspects: 1) hardware architecture (i.e., we focus on asymmetric multicores), 2) compression algorithms (i.e., we conduct data stream compression), and 3) performance metrics (i.e., we consider both compressing latency constraint violation and energy consumption).

Efficient Utilization of asymmetric multicores. The approaches to effectively manage the asymmetric multicores include modeling [24] and predicting [25] of performance/energy, CPI stack [36] and DVFS-calibrated scheduling [78], [91], [26]. The performance/energy model has been merged into the popular mainline Linux from version 5.0 onwards as the *Energy Aware Scheduling (EAS)* [92]. Yu et al. [35] recently proposed a collaborative OS scheduler addressing comprehensive multiple optimization objects on asymmetric multicores. However, these models treat the software running on asymmetric multicores as a black box due to the isolation by OS and system-level statistics, which overlooks many optimization opportunities as demonstrated in our experiments. There are also a few existing user-space research projects working on energy or latency optimization on asymmetric multicores, including virtual machine [31], web browser [33], game governor [93] and artificial intelligence framework [32]. However, all of them focus on scheduling the whole workload, which is relatively coarse-grained. For instance, Wang et al. [32] optimized the matrix operation tasks which are entirely computation intensive. In contrast, we exploit the fine-grained behavior (i.e., the different operational intensity among steps, Section III-A) of stream compression procedures.

IX. Conclusion

This paper introduced *CStream*, a novel framework to parallelize stream compression on asymmetric multicores. *CStream*'s superiority is gained by both fine-grained decomposition and asymmetry-aware scheduling strategy. We have experimentally demonstrated that *CStream* achieves the following desired properties: 1) when the compressing latency constraint (L_{set}) set by the user is relatively loose, it can achieve the least energy consumption; and 2) when encountering a tight L_{set}, its latency constraint violation is always minimized. In the future, we plan to further exploit *CStream* on more stream compression algorithms and on other hardware architectures such as Intel Agilex and Nvidia Jetson to achieve energy-efficient and low latency stream compression for a wide range of IoT applications.

Acknowledgement

We thank Dr. Feng Zhang for joining the initial discussion, and the anonymous ICDE reviewers for their valuable comments. This work is supported by the National Research Foundation, Singapore and Infocomm Media Development Authority under its Future Communications Research & Development Programme FCP-SUTD-RG-2022-005, FCP-SUTD-RG-2022-006, and a SUTD Start-up Research Grant (SRT3IS21164).

REFERENCES

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

[2] G. Theodorakis, F. Kounelis, P. Pietzuch, and H. Pirk, "Scabbard: Single-node fault-tolerant stream processing," *Proc. VLDB Endow.*, vol. 15, no. 2, p. 361–374, oct 2021. [Online]. Available: https://doi.org/10.14778/3489496.3489515

[3] J. Azar, A. Makhoul, M. Barhamgi, and R. Couturier, "An energy efficient iot data compression approach for edge machine learning," *Future Generation Computer Systems*, vol. 96, pp. 168–175, 2019.

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

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

[6] A. Ukil, S. Bandyopadhyay, and A. Pal, "Iot data compression: Sensor-agnostic approach," in *2015 data compression conference*. IEEE, 2015, pp. 303–312.

[7] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "Wan-optimized replication of backup datasets using stream-informed delta compression," *ACM Transactions on Storage (ToS)*, vol. 8, no. 4, pp. 1–26, 2012.

[8] S. Zeuch, A. Chaudhary, B. D. Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Breß, J. Traub, and V. Markl, "The nebulastream platform for data and application management in the internet of things," in *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

[9] M. Bansal, I. Chana, and S. Clarke, "A survey on iot big data: Current status, 13 v's challenges, and future directions," vol. 53, no. 6, 2020. [Online]. Available: https://doi.org/10.1145/3419634

[10] A. Lavric, V. Popa, and S. Sfichi, "Street lighting control system based on large-scale wsn: A step towards a smart city," in *2014 International Conference and Exposition on Electrical and Power Engineering (EPE)*, 2014, pp. 673–676.

[11] V. A. Memos, K. E. Psannis, Y. Ishibashi, B.-G. Kim, and B. B. Gupta, "An efficient algorithm for media-based surveillance system (eamsus) in iot smart city framework," *Future Generation Computer Systems*, vol. 83, pp. 619–628, 2018.

[12] P. Rizwan, K. Suresh, and M. R. Babu, "Real-time smart traffic management system for smart cities by using internet of things and big data," in *2016 international conference on emerging technological trends (ICETT)*. IEEE, 2016, pp. 1–7.

[13] (2020) Eclipse iot working group. iot developer survey 2018. [Online]. Available: https://https://blogs.eclipse.org/post/benjamin-cab%C3%A9/key-trends-iotdeveloper-survey-2018,2018.

[14] (2021) Arm cortex-a53 mpcore processor technical reference manual , https://developer.arm.com/documentation/ddi0500/j/. Last Accessed: 2021-05-12.

[15] (2021) Arm cortex-a72 mpcore processor technical reference manual , https://developer.arm.com/documentation/100095/0003. Last Accessed: 2021-05-12.

[16] (2021) 7-zip home page, https://www.7-zip.org/. Last Accessed: 2021-07-25.

[17] W. Li and Y. Yao, "Accelerate data compression in file system," in *2016 Data Compression Conference (DCC)*. IEEE Computer Society, 2016, pp. 615–615.

[18] A. Gupta, A. Bansal, and V. Khanduja, "Modern lossless compression techniques: Review, comparison and analysis," in *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. IEEE, 2017, pp. 1–8.

[19] A. Moffat, "Huffman coding," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–35, 2019.

[20] A. Ozsoy, M. Swany, and A. Chauhan, "Pipelined parallel lzss for streaming data compression on gpgpus," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE, 2012, pp. 37–44.

[21] K. K. Yong, M. W. Chua, and W. K. Ho, "Cuda lossless data compression algorithms: a comparative study," in *2016 IEEE Conference on Open Systems (ICOS)*. IEEE, 2016, pp. 7–12.

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

[23] A. Gupta, A. Bansal, and V. Khanduja, "Modern lossless compression techniques: Review, comparison and analysis," in *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. IEEE, 2017, pp. 1–8.

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

[25] 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.

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

[27] S. Srinivasan, N. Kurella, I. Koren, and S. Kundu, "Exploring heterogeneity within a core for improved power efficiency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 1057–1069, 2015.

[28] V. Petrucci, O. Loques, and D. Mossé, "Lucky scheduling for energy-efficient heterogeneous multi-core systems," in *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, 2012, pp. 7–7.

[29] M. Pricopi and T. Mitra, "Task scheduling on adaptive multi-core," *IEEE transactions on Computers*, vol. 63, no. 10, pp. 2590–2603, 2013.

[30] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "Armor: Defending against memory consistency model mismatches in heterogeneous architectures," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 388–400.

[31] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley, "The yin and yang of power and performance for asymmetric hardware and managed software," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 225–236.

[32] M. Wang, S. Ding, T. Cao, Y. Liu, and F. Xu, "Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, 2021, pp. 215–228.

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

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

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

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

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

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

[39] X. Gu, P. S. Yu, and K. Nahrstedt, "Optimal component composition for scalable stream processing," in *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. IEEE, 2005, pp. 773–782.

[40] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Proceedings. 36th Annual IEEE/ACM*

International Symposium on Microarchitecture, 2003. MICRO-36. IEEE, 2003, pp. 81–92.

[41] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* IEEE, 2004, pp. 64–75.

[42] X.-P. Zhang and X.-M. Cheng, "Energy consumption, carbon emissions, and economic growth in china," *Ecological economics*, vol. 68, no. 10, pp. 2706–2712, 2009.

[43] U. Soytas and R. Sari, "Energy consumption, economic growth, and carbon emissions: challenges faced by an eu candidate member," *Ecological economics*, vol. 68, no. 6, pp. 1667–1675, 2009.

[44] K. Fang, R. Heijungs, and G. R. de Snoo, "Theoretical exploration for the combination of the ecological, energy, carbon, and water footprints: Overview of a footprint family," *Ecological Indicators*, vol. 36, pp. 508–518, 2014.

[45] (2021) Rockchip wiki rk3399, http://opensource.rock-chips.com/wiki_RK3399. Last Accessed: 2021-05-10.

[46] S. Z. Sheikh and M. A. Pasha, "Energy-efficient cache-aware scheduling on heterogeneous multicore systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 206–217, 2022.

[47] A. Suyyagh and Z. Zilic, "Energy and task-aware partitioning on single-isa clustered heterogeneous processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 2, pp. 306–317, 2019.

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

[49] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 661–672.

[50] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 194–203, 1975.

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

[52] (2022) Stream: Sustainable memory bandwidth in high performance computers , https://www.cs.virginia.edu/stream/. Last Accessed: 2022-06-29.

[53] (2021) An introduction to amba axi , https://developer.arm.com/documentation/102202/latest/. Last Accessed: 2021-12-05.

[54] X. Zeng and S. Zhang, "Parallelizing stream compression for iot applications on asymmetric multicores (technical report)," 2022, https://tonyskyzeng.github.io/downloads/tr_cstream/TR_CSTREAM.pdf.

[55] (2021) lz4 source code, https://github.com/lz4/lz4/. Last Accessed: 2021-07-25.

[56] S. Zhang, J. He, A. C. Zhou, and B. He, "Briskstream: Scaling data stream processing on shared-memory multicore architectures," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 705–722.

[57] S. Zhang, Y. Wu, F. Zhang, and B. He, "Towards concurrent stateful stream processing on multicore processors," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1537–1548.

[58] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 377–388.

[59] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng, "Oversubscription on multicore processors," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–11.

[60] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2013.

[61] (2021) Perf wiki, https://perf.wiki.kernel.org/index.php/Main_Page. Last Accessed: 2021-11-07.

[62] A. Magnani and S. P. Boyd, "Convex piecewise-linear fitting," *Optimization and Engineering*, vol. 10, no. 1, pp. 1–17, 2009.

[63] A. Toriello and J. P. Vielma, "Fitting piecewise linear continuous functions," *European Journal of Operational Research*, vol. 219, no. 1, pp. 86–95, 2012.

[64] R. Bellman, "The theory of dynamic programming," *Bulletin of the American Mathematical Society*, vol. 60, no. 6, pp. 503–515, 1954.

[65] S. Schneider, J. Wolf, K. Hildrum, R. Khandekar, and K.-L. Wu, "Dynamic load balancing for ordered data-parallel regions in distributed streaming systems," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2988336.2990475

[66] K. H. Ang, G. Chong, and Y. Li, "Pid control system analysis, design, and technology," *IEEE transactions on control systems technology*, vol. 13, no. 4, pp. 559–576, 2005.

[67] L. Shen, Z. Liu, Z. Zhang, and X. Shi, "Frame-level bit allocation based on incremental pid algorithm and frame complexity estimation," *Journal of Visual Communication and Image Representation*, vol. 20, no. 1, pp. 28–34, 2009.

[68] S. Tzafestas and N. P. Papanikolopoulos, "Incremental fuzzy expert pid control," *IEEE Transactions on Industrial Electronics*, vol. 37, no. 5, pp. 365–371, 1990.

[69] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 783–798.

[70] (2021) Energy aware scheduling, https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html. Last Accessed: 2021-05-10.

[71] (2021) Beach weather stations - automated sensors , https://catalog.data.gov/dataset/beach-weather-stations-automated-sensors/resource/3b820f68-4dca-4ea7-8141-f37d9237734d. Last Accessed: 2021-11-12.

[72] (2019) Creator of the angry birds game, www.rovio.com. Last Accessed: 2021-05-10.

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

[74] (2018) Shanghai stock exchange, http://english.sse.com.cn/. Last Accessed: 2021-11-12.

[75] (2021) Rock pi 4 wiki, https://wiki.radxa.com/Rockpi4. Last Accessed: 2021-05-10.

[76] W. Wolff and B. Porter, "Performance optimization on big.little architectures: A memory-latency aware approach," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 51–61. [Online]. Available: https://doi.org/10.1145/3372799.3394370

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

[78] T. Somu Muthukaruppan, A. Pathania, and T. Mitra, "Price theory based power management for heterogeneous multi-cores," ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 161–176. [Online]. Available: https://doi.org/10.1145/2541940.2541974

[79] J. Gilchrist, "Parallel data compression with bzip2," in *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems*, vol. 16, no. 2004. Citeseer, 2004, pp. 559–564.

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

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

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

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

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

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

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

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

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

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

[90] Y. Dua, V. Kumar, and R. S. Singh, “Parallel lossless hsi compression based on rls filter,” *Journal of Parallel and Distributed Computing*, vol. 150, pp. 60–68, 2021.

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

[92] A. Mascitti, T. Cucinotta, and M. Marinoni, “An adaptive, utilization-based approach to schedule real-time tasks for arm big. little architectures,” *ACM SIGBED Review*, vol. 17, no. 1, pp. 18–23, 2020.

[93] X. Li and G. Li, “An adaptive cpu-gpu governing framework for mobile games on big. little architectures,” *IEEE Transactions on Computers*, 2020.