CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows

Siachamis, G.; Psarakis, K.; Fragkoulis, M.; van Deursen, A.; Carbone, Paris; Katsifodimos, A

DOI
10.1109/ICDE60146.2024.00309

Publication date
2024

Document Version
Final published version

Citation (APA)
Siachamis, G., Psarakis, K., Fragkoulis, M., van Deursen, A., Carbone, P., & Katsifodimos, A. (2024). *CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows*. 4030-4043. Paper presented at 2024 IEEE 40th International Conference on Data Engineering (ICDE), Utrecht, Netherlands. https://doi.org/10.1109/ICDE60146.2024.00309

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00309

CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows

George Siachamis
Delft University of Technology
g.siachamis@tudelft.nl

Kyriakos Psarakis
Delft University of Technology
k.psarakis@tudelft.nl

Marios Fragkoulis
Delft University of Technology
m.fragkoulis@tudelft.nl

Arie van Deursen
Delft University of Technology
arie.vandeursen@tudelft.nl

Paris Carbone
KTH Royal Institute of Technology
parisc@kth.se

Asterios Katsifodimos
Delft University of Technology
a.katsifodimos@tudelft.nl

***Abstract*—Stream processing in the last decade has seen broad adoption in both commercial and research settings. One key element for this success is the ability of modern stream processors to handle failures while ensuring exactly-once processing guarantees. At the moment of writing, virtually all stream processors that guarantee exactly-once processing implement a variant of Apache Flink's coordinated checkpoints – an extension of the original Chandy-Lamport checkpoints from 1985. However, the reasons behind this prevalence of the coordinated approach remain anecdotal, as reported by practitioners of the stream processing community. At the same time, common checkpointing approaches, such as the uncoordinated and the communication-induced ones, remain largely unexplored.**

This paper is the first to address this gap by i) shedding light on why practitioners have favored the coordinated approach and ii) investigating whether there are viable alternatives. To this end, we implement three checkpointing approaches that we surveyed and adapted for the distinct needs of streaming dataflows. Our analysis shows that the coordinated approach outperforms the uncoordinated and communication-induced protocols under uniformly distributed workloads. To our surprise, however, the uncoordinated approach is not only competitive to the coordinated one in uniformly distributed workloads, but it also outperforms the coordinated approach in skewed workloads. We conclude that rather than blindly employing coordinated checkpointing, research should focus on optimizing the very promising uncoordinated approach, as it can address issues with skew and support prevalent cyclic queries. We believe that our findings can trigger further research into checkpointing mechanisms.

I. Introduction

Streaming queries constitute a crucial component of cloud applications, such as online advertising, fraud detection, real-time analytics, and Internet of Things (IoT) use cases. Streaming queries are commonly executed within multi-tenant distributed environments, subject to varying service level agreements (SLAs) regarding fault-tolerance, processing guarantees (e.g., at-least/exactly-once processing), and uptime.

The first generations of streaming engines delegated the responsibility of correctness mechanisms to the application programmers [6], [11], [22]. With the advent of cloud computing, modern streaming engines, such as Apache Flink [19], Google Millwheel [7], SEEP [28], IBM Streams [25], Hazelcast Jet [30], and Microsoft Trill [20] have adopted more advanced fault tolerance mechanisms, that achieve exactly-once processing guarantees [18], [44], without the need for programmers to change the business logic to cater for failures.

At the moment of writing, there is consensus in the use of the classic coordinated checkpointing protocol [23] and its variants for rollback recovery across production-grade stream processing engines, following its initial undertaking in Apache Flink [18]. Coordinated checkpointing protocols leverage special messages, known as markers, to capture a consistent checkpoint of the distributed global state in a coordinated fashion. Once a failure occurs, a streaming pipeline can recover by rolling all operators back to their latest checkpoint and resuming processing from an offset of the streaming input.

Despite its wide adoption, the coordinated approach has been criticized for two main drawbacks. The first is that, in large deployments, the coordination can block operators with a large number of inputs (e.g., joins or aggregates) during the marker alignment phase [2], [5]. The second issue is that in case of backpressure [1], [4], the markers cannot travel through the dataflow graph, and the checkpointing mechanism stalls, eventually halting the processing of new messages.

At the same time, multiple approaches have been proposed in the past, stemming from the original uncoordinated [15], [47] and communication-induced [10], [16], [31] checkpoints (CIC). Uncoordinated protocols allow processes to take checkpoints independently, without coordination via markers, but i) they require storing logs of in-flight messages, ii) they need to execute a recovery-line algorithm before recovery, and ii) the number of messages that need to be replayed upon recovery can be substantially large (depending on the recovery line found). To alleviate these issues, the communication-induced family of protocols can limit the rollback propagation during recovery by breaking the patterns that lead to invalid checkpoints with forced checkpoints during normal execution.

Despite this convergence of the stream processing engines to the coordinated checkpointing protocol, no substantial experimental evidence currently supports this system design decision against other options (e.g., uncoordinated and communication-induced checkpoints). This lack of experimental evidence can lead future streaming engines to adopt the predominant coordinated protocol along with its drawbacks, while alternative options that could behave better are ignored. Therefore, further investigation is crucial to facilitate both research and practice toward classifying checkpointing protocols and reasoning about the protocol choices that meet the needs of different workloads.

In addressing these gaps, this work is the first to revisit checkpointing for stream processing from its first principles. First, we present and analyze the theoretical cost of existing approaches. We then experimentally evaluate the three prominent checkpointing protocol families by implementing them in a testbed system built for the needs of this evaluation. We push the protocols to their limits on diverse workloads, resulting in various topologies and processing needs, including a cyclic query. Finally, we measure the performance and the impact of the protocols both on failure-free execution as well as under failure in both uniform and skewed workloads.

In summary, this paper makes the following contributions:

- A comprehensive survey of three families of checkpointing approaches and the conditions under which they can guarantee exactly-once processing.
- A theoretical account of the advantages and drawbacks of those three checkpointing approaches in streaming dataflows.
- An open-source streaming dataflow testbed system that enables accurate and isolated comparison of different checkpointing protocols.
- The first experimental evaluation of three checkpointing approaches on different workloads using NexMark queries [46] and a custom query that causes cycles in the dataflow graph.
- The first experimental evidence showing that:
  - Under *uniformly* distributed workloads, the coordinated approach outperforms all other approaches;
  - Under *skewed* workloads, the uncoordinated approach outperforms the coordinated one despite its expensive in-flight message logging;
  - The uncoordinated approach in practice does not suffer from the (theoretical) domino effect [27] in any of our experiments.
  - The communication-induced approach is not competitive in any scenario due to its large message overhead that it requires to avoid the (improbable, in our experiments) domino effect.

The rest of the paper is structured as follows. In Section II, we summarise all the necessary background knowledge required to understand checkpointing. Then we discuss in detail the benchmarked protocols (Section III) and the system used for the benchmarking (Section IV). In Section VII, we describe the experimental setup and present and comment on our results. In Section VIII, we discuss related existing works. Section IX concludes this paper.

The code of CheckMate can be found online:
https://github.com/delftdata/checkmate

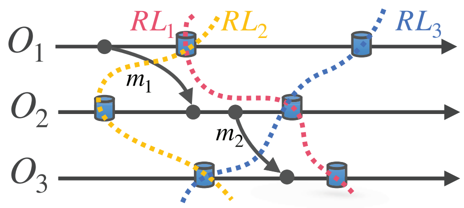


Fig. 1: Examples of valid recovery lines when in-flight messages are included in the global state.

II. Preliminaries

In what follows, we discuss all the necessary concepts to understand better and evaluate the checkpointing protocols, particularly processing semantics and consistency in the face of failures.

A. Processing Semantics

Different applications have different processing needs. Stream processing engines and their fault tolerance mechanisms provide specific processing semantics to accommodate these needs even when a failure occurs. A recent survey [29] identifies three predominant semantics with regard to processing: *at-most-once*, *at-least-once*, and *exactly-once*.

For data analytics, monitoring, or other applications that can tolerate incomplete data, a stream processing engine that provides *at-most-once semantics* is sufficient. We define *at-most-once* semantics as follows:

Definition 1 (At-most-once). A stream processing engine provides *at-most-once* processing semantics when it ensures that each streaming operator will process each ingested record once or not at all.

At-most-once semantics are the weakest guarantees a stream processing engine can provide. This processing guarantee has been termed *gap recovery* in the past [32]. In case of a failure, in-flight records can be lost and never be processed by downstream operators.

To accommodate applications that are intolerant of losing messages, streaming engines support *at-least-once* semantics.

Definition 2 (At-least-once). *At-least-once* processing semantics are provided when each ingested message is processed one or more times by each streaming operator.

By providing at-least-once semantics, a streaming engine can avoid data loss, but at the same time, it is amenable to accounting for the same message more than once. For sensitive applications, such as bank transaction handling or aggregations, duplicate processing can cause serious anomalies. In such cases, *exactly-once* semantics are necessary.

Definition 3 (Exactly-once). *Exactly-once* semantics guarantee that each ingested message is processed exactly once in each operator, i.e., any state changes that occur from processing a message are reflected exactly-once on the checkpointed state.

Exactly-once semantics define strict guarantees, and they can ensure that processing under failures is identical to failure-free processing. Note that there is a distinction [29] between exactly-once *processing* and exactly-once *output* [26]. In exactly-once processing, an external system consuming the output can still observe duplicates. For instance, in case of fault recovery, the streaming system will resume processing after the latest checkpoint and produce some output that it had already produced (but not yet checkpointed the corresponding state) prior to the failure.

In the rest of the paper, we only consider *exactly-once* processing guarantees.

B. Consistency of Global State

Data stream execution is data-driven, where processing is orchestrated by messages being sent and received between tasks, triggering local computation. Without loss of generality, a distributed stream execution consists solely of `send` and `receive` operations corresponding to each message.

Modern distributed stream processing engines refer to the *global state* as the collection of the states of all operators of a streaming pipeline. We refer to an operation (`send` or `receive`) being part of the global state if it occurred before the respective state acquisition. Furthermore, the state of the communication channels can also be included in the global state. These messages are also known as *in-flight messages* or *channel state*. The *consistency* [23] of the global state is of major importance here. In order to define what a consistent state entails, we first define the concept of orphan messages:

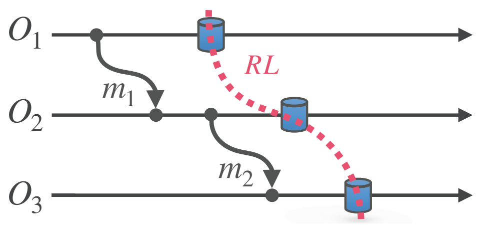
Definition 4 (Orphan message). Given a global state checkpoint G, an *orphan message* has been received prior to the receiver's local checkpoint S in G, but it was not sent prior to the sender's checkpoint S' in G.

The global state of a streaming pipeline becomes inconsistent in the presence of a dropped or an orphan message [17], [27], [45]. Following the seminal processing model of Chandy-Lamport [23], we define consistent global state as follows:

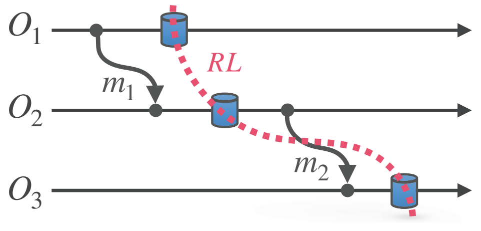
Definition 5 (Consistent global state). The global state G of a streaming pipeline is *consistent* if for each message m :

- **No Orphans:** if `receive(m)` happened before the checkpoint acquisition, the corresponding `send(m)` operation should also happen before the checkpoint.
- **No Dropping:** if `send(m)` happened before the checkpoint acquisition then either `receive(m)` happens before the checkpoint or m is added in the checkpoint as an *in-flight* message.

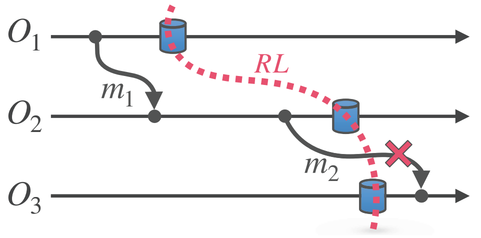
In principle, consistency is straightforward to maintain and reason about under the normal operation of a streaming system. In the face of failures, however, a streaming system ought to roll back to a previously consistent global state in order to resume its operation and regain consistency. At that point, the recovery mechanism attempts to recover such a global state from the collection of existing operator checkpoints.



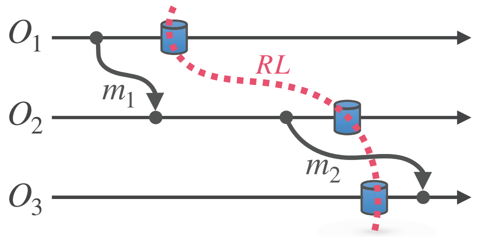
(a) Consistent state after recovery



(b) Inconsistent state after recovery



(c) Inconsistent state without capturing in-flight messages



(d) Consistent state by capturing in-flight messages

Fig. 2: Cases of inconsistent and consistent state after recovery for stateful operators O_1, O_2 and O_3.

Recovery line. A *recovery line* consists of a collection of operator checkpoints that can be used to recover the global state (Figure 1). Since not all candidate recovery lines lead to a consistent state, the recovery mechanism must find the most recent recovery line corresponding to a consistent state. Checkpoints that cannot belong to a consistent recovery line are considered *invalid*.

In Figure 2, we provide example cases that illustrate when a recovery line and its corresponding global state are consistent. Figure 2a showcases a consistent global state since all messages are sent and received before the checkpoints that compose the recovery line. In Figure 2b, message m_2 is an orphan message since its side-effects are reflected in the checkpoint of O_3 but not in the checkpoint of the sender operator O_2. Therefore, the global state corresponding to this recovery line is inconsistent, and the recovery line is unsuitable for recovering from a failure.

	Blocking (markers)	In-flight Logging	Deduplication Required	Message Overhead	Independent Checkpoints	Straggler Stalls	Unused Checkpoints	Forced Checkpoints
Coordinated	○	–	–	–	–	○	–	–
Uncoordinated	–	○	○	–	○	–	○	–
Communication-induced	–	○	○	○	○	–	○	○

TABLE I: Summary of the features of the checkpointing protocols explored in Section III

If in-flight messages (i.e., channel state) are not captured, then a different type of global state inconsistency appears. In Figure 2c, operation `send(`m_2`)` occurs before O_2 acquires its checkpoint, whereas, `receive(`m_2`)` occurs after O_3 takes its checkpoint. Using this recovery line without a captured channel state will result in never processing m_2 at operator O_3 and, therefore, dropping messages. In this case, to achieve a consistent global state, capturing the channel state and replaying in-flight messages is necessary (Figure 2d). To ensure exactly-once semantics when in-flight messages are replayed, some form of message deduplication must be employed.

III. Checkpointing Protocols

In what follows, we describe the three main checkpointing protocols and discuss their core ideas and some possible drawbacks. In table I, we summarise the necessary mechanisms employed for each protocol to ensure exactly-once processing and the main side effects and features of each protocol.

A. *Coordinated Aligned Checkpointing (COOR)*

To the best of our knowledge, virtually every stream processing engine in production that guarantees exactly-once processing, implements a variation of the coordinated checkpointing protocol [18], [20], [25], [30]. Typically, in stream processing engines that implement a coordinated checkpointing protocol, the operators will block processing to allow the alignment of a checkpoint across the system. The checkpoint can be used to create a recovery line in case of a failure. The most adopted version of such a protocol is the Chandy-Lamport marker-based algorithm [23] and its adaptation for acyclic dataflow graphs [18]. In what follows, we describe the core ideas of the protocol, and we illustrate its core functionality with an example.

At its core, the coordinated aligned checkpointing protocol works as follows:

- A checkpoint round initiates at source operators by taking a checkpoint. After taking its checkpoint, each source operator forwards a marker to all its outgoing channels and continues processing.
- When an operator (excluding source operators) receives a marker from an incoming channel, it blocks that channel and buffers the channel's traffic.
- When an operator receives a marker from all its incoming channels, it takes a checkpoint, unblocks processing in all incoming channels, and forwards a marker to all its outgoing channels.
- When the markers reach the end of the pipeline, and the checkpoints are stored in durable storage, the coordinated checkpoint round finishes.

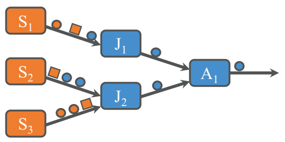
By blocking processing until all markers are received from the upstream operators, we achieve the alignment of the checkpoints. This alignment guarantees exactly-once processing without the need to capture in-flight messages and the channel state, as it creates a frontier of processed messages through the use of markers.

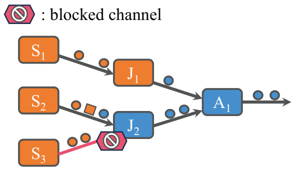
Figure 3 illustrates an example protocol execution. The execution graph presented consists of only the first couple operators of the pipeline. Operators $S_{\{1-3\}}$ are parallel source operators, operators J_1 and J_2 are parallel stateful join operators, and operator A_1 is a stateful aggregation operator. A coordinated checkpoint round is initiated at the source operators by taking a checkpoint. When a parallel source operator finishes with its own checkpoint, it sends a checkpointing marker to all its outgoing channels (fig. 3(a)) and continues processing. In fig. 3(b), operator J_1 has received the marker from its sole incoming channel and takes a checkpoint. On the other hand, operator J_2 has received a marker from source operator S_3 and blocks processing in that channel while it waits for the marker from S_2. J_2 takes a checkpoint when it has received all markers, while J_1, after taking the checkpoint, forwards a marker to its downstream operator and unblocks processing in all the incoming channels (fig. 3(c)). Finally, J_2 also forwards a marker and continues processing after taking a checkpoint (fig. 3(d)). The markers will then be received by A_1, and the checkpointing process will continue in the same way until it reaches the end of the pipeline.

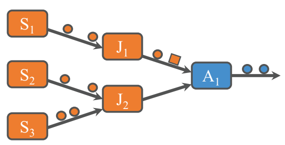
Strengths. Compared to the in-flight message logging required in uncoordinated approaches (Section III-B), the markers used by the coordinated protocol are lightweight and are not affected by the message size. Additionally, since aligned checkpoints compose a consistent global state, an algorithm that identifies the recovery line is not required.

Drawbacks. One important downside of marker circulation surfaces in cases of stragglers, e.g., due to skewed workloads and/or backpressure. For example, if most of the load falls on a single operator, its downstream operators would have to block the other channels, wait for the straggler to finish processing, and then forward a checkpoint marker. Additionally, in case of shuffling, the protocol needs to transfer as many markers as the parallel instances of the receiving operators (one to each parallel instance). In essence, coordinated checkpoints could take a substantial amount of time in complex topologies due to the markers having to pass through the entire dataflow graph to be completed.

Another drawback of the coordinated protocol is that it does not support cyclic streaming workloads out of the box.


(a) Initiate checkpoint round


(b) J_1 takes a checkpoint while J_2 blocks one channel waiting for the other marker.


(c) J_1 forwards the marker, while J_2 takes a checkpoint.

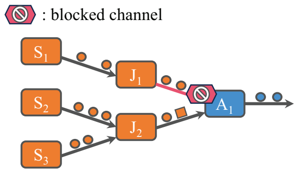

(d) Join operator J_2 forwards the marker and unlocks processing in both channels.

Fig. 3: Example execution of the coordinated aligned checkpointing protocol. Messages are represented as circles, and markers are squares. Different colors denote different coordinated rounds.

Cycles are an integral aspect of iterative computations such as fixpoint calculations, which are common in graph queries [37]. Accounting for cycles in the coordinated checkpointing protocol entails a) special handling of markers in order to avoid deadlocks owed to the blocking of the cyclic input channel by a marker and b) additional progress tracking mechanisms.

B. Uncoordinated Checkpointing (UNC)

The *uncoordinated checkpointing* (UNC) [47] protocol allows each operator to decide individually when to take a checkpoint. In contrast to the coordinated approach, there are no markers since there is no need for coordination, and the protocol can only provide at-most-once processing semantics since the checkpoints only contain the operator state. Thus, capturing the channel state between operators is necessary to provide stronger guarantees. To do so, log-based recovery and upstream backup [13], [24] need to be implemented. Pairing uncoordinated checkpointing with a log for keeping track of the channel state allows the replay of messages after recovery, achieving at-least-once semantics. For the protocol to achieve exactly-once semantics, message *deduplication* must be employed when replaying messages from the message log.

Algorithm 1 Rollback propagation algorithm [47]

```
Require: all available checkpoints $CP$ ordered by freshness
    for each operator, a checkpoints graph
Ensure: a consistent recovery line
1: include in $root\_set$ the latest $CP$ of each operator;
2: mark all $CPs$ in the $root\_set$ that are strictly reachable
    from any other $CP$ in the $root\_set$;
3: while $\exists CP.marked \in root\_set$ do
4:     $\forall CP.marked \in root\_set$ replace by the next unmarked
       $CP$ from the same operator;
5:     mark all $CPs$ in the $root\_set$ that are strictly reachable
       from any other $CP$ in the $root\_set$;
6: end while
7: return $root\_set$
```

Finding Recovery Lines. The freedom of taking checkpoints independently per operator comes with a cost when recovering after a failure. Since the checkpoints are not coordinated, we cannot simply use the most recent operator checkpoints as a recovery line, as it might not correspond to a consistent global state. Therefore, we need to employ a recovery line algorithm to find a suitable recovery line, i.e., one that provides a consistent global state and has the minimum rollback distance. The algorithm for finding such a recovery line is the *rollback propagation algorithm* [47], which requires a checkpoint dependency graph. There are two approaches to creating such a graph, the *rollback dependency graph* [14] and the *checkpoint graph* [47]. Both of these approaches result in the same recovery line, and in this work, we opt for the *checkpoint-graph* [47] since it is more intuitive.

The checkpoint graph has checkpoints as nodes and directed edges between two checkpoints $c_{i,x}$ and $c_{j,y}$ if:

- $i \neq j$, i.e., the checkpoints belong to different operators, and there is at least one orphan message that was sent from operator i after checkpoint $c_{i,x}$ was captured and was processed from operator j before checkpoint $c_{j,y}$ was taken.
- $i = j$ and $y = x + 1$, i.e., $c_{i,x}$ and $c_{j,y}$ are consecutive checkpoints of the same operator.

In Figure 4, we provide an example of a checkpoint graph and showcase step by step how the rollback propagation algorithm uses the checkpoint graph to find a suitable recovery line. To create the checkpoint graph, we include the IDs from channel state logs for the last received and last sent messages alongside the checkpoints. We can identify orphan messages using these IDs and add directed edges in the checkpoint graph (Figure 4(a)). The rollback propagation algorithm uses this graph to find the recovery line. First, the algorithm will include the last checkpoints of all operators in a set called the root set (Figure 4(b) - step 1). The next step is to identify the nodes in the root set that are strictly reachable from other nodes in the root set and mark them (Figure 4(b) - step 2). Then, each marked checkpoint in the root set is replaced by the next most fresh checkpoint for the same operator, and the newly added checkpoints are checked and marked if applicable (Figure 4(b) - step 3). When the algorithm reaches a root set that does not include any marked checkpoint, it returns this root set as the desired recovery line.

Strengths. The primary strength of any coordination-free protocol is that it does not block waiting for markers from a coordinator node or its upstream operators, leading to lower

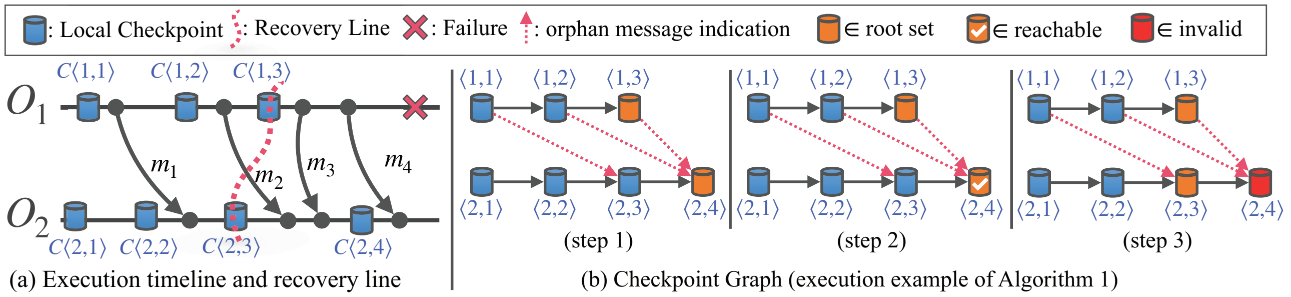


Fig. 4: Example overview of Rollback propagation algorithm on a given execution timeline

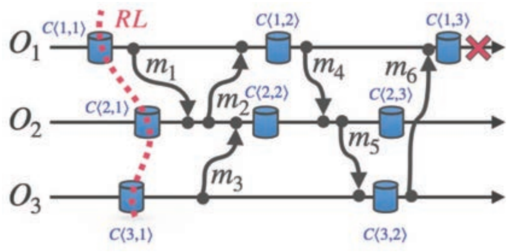


Fig. 5: Domino effect of invalid checkpoints on a cyclic query.

latency in the event of a skewed workload. Another benefit yet to be explored by literature is the configurability of such an approach. For instance, the stateless, non-source operators in the uncoordinated approach do not need to participate in the checkpointing pipeline, which is not the case in the coordinated approach because they still would have to propagate the markers. Furthermore, different operators can have different checkpoint intervals, making them adaptive to the current system's needs (e.g., a windowed aggregation operator can checkpoint right after the aggregate is calculated in order to avoid storing the large window's contents).

Drawbacks. To provide exactly-once semantics, message logging is required. However, message logging is costly and can considerably impact the system's performance. Moreover, since checkpoints are not aligned, some captured checkpoints may be rendered invalid when looking for the appropriate recovery line (an invalid checkpoint cannot take part in any recovery line). As seen in Figure 5, this problem could be aggravated when dealing with cyclic queries, leading to a phenomenon known in the literature as the *unbounded domino effect* [27], where during recovery, one checkpoint after the other is rendered invalid leading to a considerable rollback distance or even starting from scratch. In Figure 5, the first option would be a recovery line consisting of the checkpoints $C_{<1,3>}$, $C_{<2,3>}$, and $C_{<3,2>}$; however, this is invalid due to the orphan message m_6. The next option is the recovery line consisting of $C_{<1,2>}$, $C_{<2,3>}$, and $C_{<3,2>}$ with again m_4 making this invalid. m_5 makes the $C_{<1,2>}$, $C_{<2,2>}$, and $C_{<3,2>}$ invalid. The domino effect continues with the rest of $m_{3,2,1}$ leading to $C_{<1,1>}$, $C_{<2,1>}$, and $C_{<3,1>}$ being the only available recovery line option.

C. *Communication-induced Checkpointing (CIC)*

The *communication-induced checkpointing* (CIC) protocol is built on top of UNC and provides a loose coordination of the checkpoints in order to tackle the problem of the *unbounded domino effect*. This loose coordination happens through encapsulating information related to the protocol in the messages containing records across the pipeline. This protocol recognizes two different types of checkpoints: a) *local checkpoints* (equivalent to uncoordinated checkpoints), and b) *forced checkpoints*, which are inserted by the protocol to prevent the domino effect.

Communication-induced protocols are tightly connected to Z-paths and Z-cycles [27] based on the fact that a given checkpoint is invalid if and only if it is part of a Z-cycle. A CIC protocol tries to detect Z-cycles and break them by forcing checkpoints before processing messages that will lead to a cycle. Alvisi et al. [10] have shown that a CIC protocol can handle cyclic communication patterns without the risk of a domino effect, but they may introduce significant overhead.

The most complete and well-documented CIC protocols are BCS [16] and HMNR [31]. Initial tests indicate that the HMNR has better performance than BCS. Therefore, in this paper, we adopt HMNR as our CIC protocol. In short, in HMNR each operator keeps a Lamport clock and a vector clock plus three boolean vectors with a length equal to the number of operators participating in the pipeline. Every operator updates his Lamport clock by increasing its value when it takes a new checkpoint. The vector clock *ckpt* stores how many checkpoints have been taken by each operator from the perspective of the current operator. A boolean vector *sent_to* keeps information about messages sent to other operators since the last checkpoint of the current operator. Another boolean vector *taken* stores the existence of Z-paths since the last known checkpoint. The last boolean vector *greater* stores the information whether the operator's clock is greater or not from each other operator's clock. The operator's Lamport clock, the vector clock *ckpt*, the boolean vector *taken*, and the boolean vector *greater* are piggybacked to every message. The protocol uses all these structures to detect cycles and decide when to force a checkpoint. When an operator receives a message, it checks if there is a message previously sent from it to the sender and the sender's clock is larger than its own or if there is a Z-path detected in the current checkpoint interval of the sender operator. More details on the cycle detection and the forced checkpoints can be found in the original paper [31].

Strengths. The primary strength of the CIC protocol is the forced checkpoints mechanism, leading to a smaller rollback distance and, most importantly, eliminating the domino effect.

Drawbacks. The main drawback of a CIC protocol is the overhead it introduces. For big and complex pipelines, the vector clocks and the boolean vectors can be rather large and greatly impact the size of the messages flowing throughout the system.

IV. Testbed System

We compared the checkpointing protocols in Styx [40], the backend of Stateflow [39]. For the requirements of our experiments, we developed all necessary protocol mechanisms (e.g., message logging and coordination) and streaming operators (i.e., map, filter, window, join, aggregates).

The Stateflow cluster consists of the typical architecture. A coordinator node is responsible for scheduling/deploying the dataflow graph to workers and running the coordination logic of the checkpointing protocols. Worker nodes execute the dataflow logic and take checkpoints asynchronously based on the checkpointing algorithm. Finally, Stateflow uses Apache Kafka as a replayable fault-tolerant source and Minio as a persistent state store for the operator state checkpoints.

We choose Stateflow for the following reasons: i) unlike other streaming dataflow systems such as ApacheFlink, Stateflow allows for cycles in the dataflow graph; ii) Stateflow provides a sandboxed environment, where we can evaluate the different protocols in isolation, without additional overhead; iii) Other systems (e.g., Apache Flink) base their entire design on coordinated checkpoints – when implementing uncoordinated protocols on Apache Flink, we realized that we needed to virtually rewrite the complete system itself.

V. Metrics

Although there is a significant body of work in benchmarking and evaluating stream processing systems and fault tolerance (Section VIII), no metrics are established to measure the performance of a checkpointing protocol meaningfully. In this work, we argue that the following metrics should be used to evaluate the performance of such a protocol.

End-to-end Latency. A standard metric to evaluate the performance of stream processing systems is the *end-to-end latency*, i.e., the time it takes for a record to result into output in the sink from the moment it is available in the input queue. Although latency is mainly related to the deployed query and the underlying system rather than the checkpointing protocol itself, it allows us to measure the impact of each protocol on normal execution, as the overhead it introduces in terms of latency. We opt to measure the 50th and 99th percentiles.

Sustainable Throughput. Another common metric in stream processing literature is the *maximum sustainable throughput* [34]. The *maximum sustainable throughput* indicates the maximum throughput that the system can handle for a long period of time without provoking backpressure. Backpressure leads to constantly increasing latencies and an average processing throughput that is lower than the rate of incoming messages. Similarly to end-to-end latency, it allows us to assess the impact of the checkpointing protocol on the overall performance.

Average Checkpointing Time. In this work, we measure the *average checkpointing time*, i.e., the average time it takes for each protocol to take a checkpoint. The fundamental differences between the protocols lie in checkpoint triggering and the additional information that needs to be captured apart from the internal state. Therefore, measuring how these differences affect the time it takes to capture a checkpoint is crucial. Also, as the checkpointing time rises, a significant impact on the processing performance is expected.

Restart & Recovery Time. Restart time consists of all the time the system spends to reload all the needed states and be ready to process data. The recovery time, on the other hand, informs us how long it takes to recover from a failure. The measurement starts when the failure is detected and finishes when the system has managed to return to normal execution. The higher the recovery time, the bigger the impact of a failure. Recovery time also encompasses restart time.

Invalid Checkpoints. Depending on the checkpointing protocol, invalid checkpoints may exist, i.e., checkpoints that cannot be part of a consistent recovery line and, thus, cannot be used for recovering after failure. The existence of invalid checkpoints can be problematic as the state grows since a lot of expensive storage space is occupied by information that will never be used. Moreover, invalid checkpoints can lead to significant rollback distance, which will result in replaying and reprocessing a significant number of messages. Therefore, the number of invalid checkpoints is a good indicator of the performance of a checkpointing protocol. The fewer invalid checkpoints exist, the better a protocol is performing.

Message Overhead. Each protocol introduces messages and requires specific information to be exchanged between workers or sent to the coordinator. Measuring the size of protocol-related information that circulates the system during execution allows us to capture the overhead that the protocol introduces in network usage. A higher percentage of protocol-related information means that a significant portion of our network is used, and additional serialization/deserialization CPU time is spent on information unrelated to processing.

VI. Streaming Query Workload

To evaluate the checkpointing protocols, we employ four distinct queries from NexMark [46] and our adaptation of the cyclic query introduced in [21].

NexMark Queries. NexMark benchmark [46] simulates an e-commerce application and provides streaming queries with different properties and needs. We selected the following four queries, which allow us to measure the performance and the impact of the checkpointing protocols in different conditions:

- *Query 1* is a stateless map query that transforms the bid values. There is no shuffling.

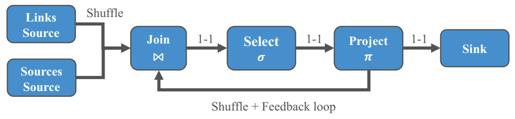


Fig. 6: Execution graph of the reachability query.

- *Query 3* implements an incremental stateful join, which joins persons with auctions. It involves a complex topology and shuffling between operators.
- *Query 8* employs a windowed join between users and auctions. We opt for a processing time tumbling window; however, the type of the time window does not affect the checkpointing protocol's performance. It employs a complex topology, shuffling, and the complexity of the windowing. To meaningfully measure the impact of the protocols on the latency during execution, we implement a running window, i.e., the processing is triggered on record arrival, and the window is cleaned when it expires.
- *Query 12* employs a windowed count over bids. Similarly to query 8, we choose the running version of a processing time tumbling window. The query performs aggregation over time windows and includes minor shuffling.

Fundamental processing operators in modern stream processing engines [8], [19], [30] include maps, joins, windows, and aggregates. The queries we choose represent those fundamental operations and sufficiently cover the operations appearing in the NexMark suite.

Cyclic Query. Most stream processing engines do not support cyclic queries. However, there is existing research on cyclic or recursive queries in stream processing [21], [37], [38]. To further enable research on cyclic streaming queries and to encourage stream processing engines to support such queries, it is essential to evaluate existing checkpointing protocols with cyclic queries. For our evaluation, we adapt the reachability query employed by FFP [21]. Given a static set of nodes, the goal of the query is to identify all reachable nodes from the available source nodes based on the available directed links between the nodes and provide the corresponding paths. The available source nodes and the directed links between the nodes are not known a priori, but they are processed on the fly and are temporal. Figure 6 illustrates the execution graph of the query. The query ingests two streams, the directed links between the nodes and the source nodes. Directed links are joined with sources that contain the starting node of the link as a reachable node. In the select operator, we check if the end node of the directed link of a joined pair is contained in the path of the source of the pair, and we discard such pairs. In the project operator, we discard unnecessary information and create a new source with the same source node, the end node of the link as a reachable node, and the path augmented by the pair's link. The new source is provided as output and recursively as input to the join operator. Finally, the join operator can receive direct messages when a specific link or source node is unavailable. In that case, it will remove every link or source affected from its state.

VII. Experimental Evaluation

A. Evaluation setup

The experiments are conducted on a local cluster with AMD EPYC 7H12 2.60GHz CPUs and 512GBs of memory. We deploy our benchmarking system using docker and docker-compose. Each worker uses 1 CPU for processing and handles a single parallel instance of each of the operators of the deployed pipeline. We do not use any limits on memory usage. Apache Kafka is used as the source and the sink of our system. Minio is used as a persistent storage for the checkpoints. We extend the NexMark generator from [33], [43] to provide the input in the required format of the system, and we provide a generator that creates source nodes and corresponding links for our cyclic query. We evaluate the three checkpointing protocols using the NexMark queries and our cyclic query. We implement and compare the vanilla versions of the protocols as described in Section III in order to ensure a fair comparison of their core concepts that is not affected by optimizations tailored to specific system properties.

B. Results

In what follows, we present the results of our experimental evaluation of the three checkpointing protocols concerning the metrics for benchmarking checkpointing protocols that we previously discussed in Section V. For the NexMark queries, we distinguish two settings: a balanced setting where the distribution of our input follows the uniform distribution and a skewed setting where we leverage NexMark's generator to provide different percentages of hot items.

NexMark Queries. In practice, streaming systems are over-provisioned, ensuring a stable execution that does not cause backpressure in case of input rate fluctuations or transient system issues (e.g., garbage collection). In our experiments, we run all queries at 80% of the maximum sustainable throughput that each protocol achieves for each query and parallelism. We found 80% to be the most stable configuration. Each run lasts for 60 seconds with 30 seconds of warmup. We introduce a failure on the 18th second of a 60-second run.

– *Maximum Sustainable Throughput (MST).* In Figure 7, we present the maximum sustainable throughput (MST) each protocol achieved normalized by the MST of the checkpoint-free execution for each query. For Q1, Q8, and Q12, the coordinated approach outperforms the rest and reaches the same MST as the checkpoint-free execution until we reach 70 workers. For 70 and 100 workers, we observe a slight decrease in MST for Q1 and Q12, which results in approximately 90% of the checkpoint-free MST. The impact of the increase in parallelism is more significant for Q8, which employs a join. The uncoordinated protocol follows closely, achieving an MST around 10% lower than the coordinated approach in all cases. On the other hand, the communication-induced protocol fails to keep up and, in higher parallelism, can reach an MST lower than 50% of the checkpoint-free MST. None of the protocols can keep up with the checkpoint-free execution for Q3. However, the coordinated and uncoordinated protocols

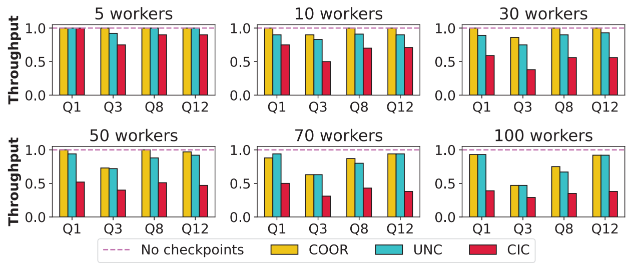


Fig. 7: Normalized maximum sustainable throughput per query achieved by each protocol for different parallelism.

TABLE II: Ratio of message overhead with respect to an execution without checkpoints.

	10 workers				50 workers			
Protocol	Q1	Q3	Q8	Q12	Q1	Q3	Q8	Q12
COOR	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
UNC	1.00x	1.00x	1.00x	1.00x	1.00x	1.01x	1.01x	1.00
CIC	2.10x	1.82x	1.74x	1.79x	2.53x	2.58x	2.49x	2.58x

achieve an MST higher than 70% of the optimal for Q3 in most cases while maintaining an MST of 50% of the optimal for the edge case of 100 workers. On the contrary, the communication-induced protocol fails to achieve an MST higher than 50% for Q3 primarily due to the high message overhead it introduces. In terms of MST, the coordinated approach outperforms the others, while only the uncoordinated can remain competitive.

– *Message Overhead.* The overhead of the protocol-related information transferred throughout the system can either be in the form of additional protocol messages and/or piggy-backed information to process messages. The only protocol-related overhead for the coordinated approach is the messages between workers and the coordinator when starting and concluding a coordinated round, and the markers forwarded from the sources to the pipeline sinks. The uncoordinated protocol requires the operators to send the metadata of every checkpoint they take to the coordinator. Table II shows that the overhead that the coordinated and the uncoordinated introduce is insignificant in all cases. On the contrary, as explained in Section III, additionally to the information required by the uncoordinated protocol, the communication-induced protocol piggybacks to the process messages all the information required to decide on forcing a checkpoint. The size of this information depends on the number of total instances of the operators employed. As Table II indicates, even for a parallelism of 10, the overhead can double the size of the messages that are communicated between the workers and the coordinator, while for 50 workers, the message size can reach up to 2.58x the size of messages of a checkpoint-free execution. Increased message size does not only result in the need for higher network bandwidth but also cripples the processing power of our system as it has to serialize and deserialize much larger messages. Therefore, it significantly affects the maximum sustainable throughput we can achieve using the communication-induced protocol, as shown in Figure 7.

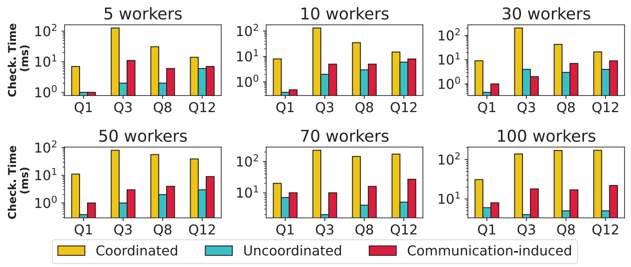


Fig. 8: Average checkpointing time on different parallelisms.

– *Average Checkpointing Time.* We showcase the average checkpointing time for each protocol for all settings in Figure 8. The uncoordinated and communication-induced protocols have an average checkpointing time of a few milliseconds for all settings. The coordinated approach requires a full checkpointing round to be completed to consider its checkpoints as valid. Therefore, in contrast to the other protocols, it incurs an average checkpointing time of up to two magnitudes higher for Q3, Q8, and Q12, which involve shuffling. This is especially the case for Q3, which employs a complex topology and has a high computational complexity, as well as for the higher parallelisms that result in a higher degree of shuffling. The latency overhead caused by the increased checkpointing time in Q3 is also visible in Figure 9 for 10 workers.

– *Impact on the 50th and 99th percentile of latency.* In Figure 9 and Figure 10, we present the 50th and 99th percentiles per second for each protocol and query for different parallelisms. Due to space limitations, we include 10, 30, and 50 workers in our discussion. However, the other settings follow a similar trend. The 50th percentile latency allows us to evaluate the mean performance of the protocols, while the 99th percentile highlights the stragglers and the outliers. For the settings of 10 and 30 workers, the 50th percentile for all protocols for the simpler queries Q1, Q8, and Q12 is similar before the failure occurred and after the system recovered to a stable execution. However, for the 50-worker case, the communication-induced protocol requires piggybacking additional protocol information of significant size at every message. This results in a slight increase observed in the 50th percentile, which is considerably higher in Q8 because it employs a costly join. As for Q3, the coordinated approach suffers from latency spikes every time a checkpoint is taken, which is more evident as the state grows and for the 10-worker case. The 99th percentile follows the same patterns as the 50th percentile for the execution period prior to the failure. Q3 employs an incremental join; the spikes and the increasing instability in latency that we observe are expected and attributed to a combination of the query's nature and checkpointing.

– *Recovery & Restart Time.* Recovery time is the time passed from detecting the failure until the system returns to normal and stable execution. Looking at the 50th percentile (Figure 9), all protocols require around 10 seconds to recover to normal execution for Q1 for a parallelism of 10, while very small differences are also observed for Q1 for 30 workers. For 50

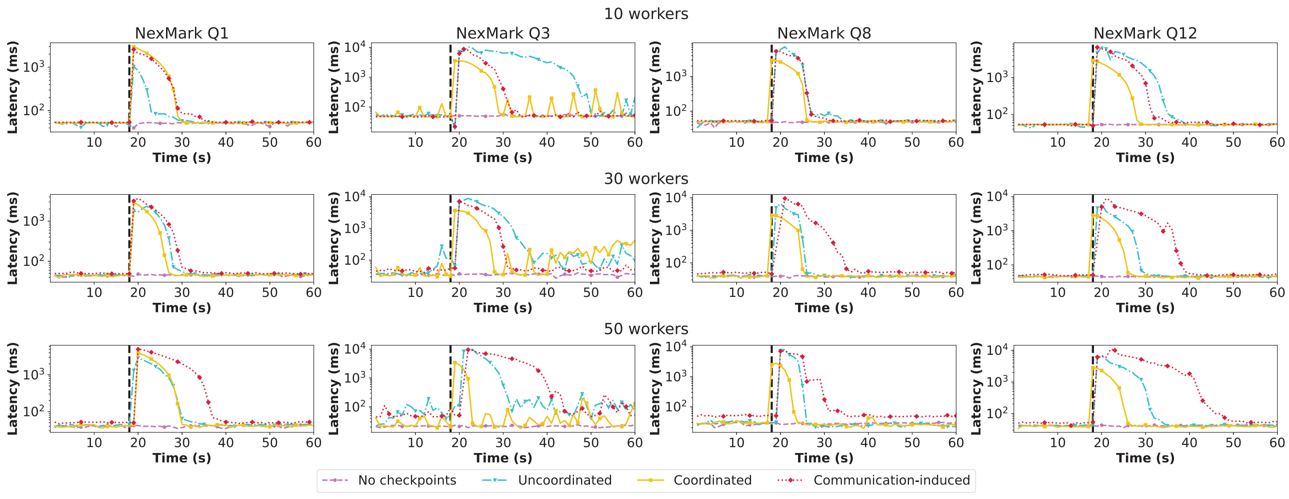


Fig. 9: 50th percentile latency. The black dashed vertical line indicates the moment of failure.

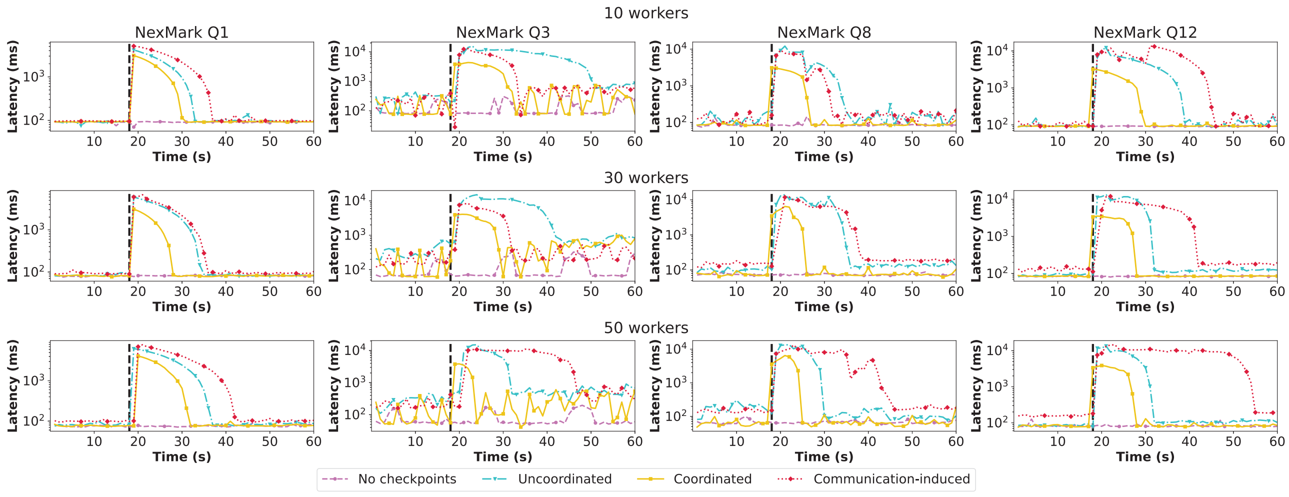


Fig. 10: 99th percentile latency. The black dashed vertical line indicates the moment of failure.

workers, the communication-induced protocol requires around 10 more seconds to recover due to the significant message overhead it introduces. For Q8 and Q12, the communication-induced protocol performs marginally better than the uncoordinated protocol for 10 workers, but it falls behind when the parallelism increases as it requires around 10 more additional seconds to recover. In Q3, the communication-induced protocol has a smaller recovery time than uncoordinated by up to 20 seconds for 10 and 30 workers, resulting from replaying fewer messages due to forced checkpoints closer to the failure. On the other hand, it requires 10 additional seconds for 50 workers. On average, the coordinated protocol greatly outperforms the other protocols regarding recovery time, mostly because the uncoordinated and communication-induced protocols have to replay many messages.

The restart time (Figure 11) is part of the recovery time and reflects the time passed from detecting the failure until the system is ready to restart processing. On average, the coordinated protocol restarts faster than the other two protocols. This is especially evident for a larger number of workers. For example, the restart process for the uncoordinated and communication-induced protocols can take up to 10 times longer than the coordinated for 100 workers. The UNC and CIC protocols need to fetch and prepare the messages to replay and, therefore, take more time to restart. On the other hand, finding the recovery line has an insignificant cost.

– *Invalid checkpoints.* The percentage of invalid checkpoints

TABLE III: Total checkpoints and percentage of invalid checkpoints.

	10 workers			50 workers		
	Total(Invalid)			***Total(Invalid)***		
Query	*UNC*	*CIC*	*COOR*	*UNC*	*CIC*	*COOR*
Q1	303(0%)	285(0%)	240(0%)	1437(0%)	1428(0%)	1200(0%)
Q3	455(4%)	471(3%)	400(0%)	2399(3%)	2517(4%)	2000(0%)
Q8	384(2%)	386(3%)	360(0%)	1924(2%)	1920(3%)	1800(0%)
Q12	282(3%)	282(4%)	240(0%)	1446(3%)	1451(3%)	1200(0%)

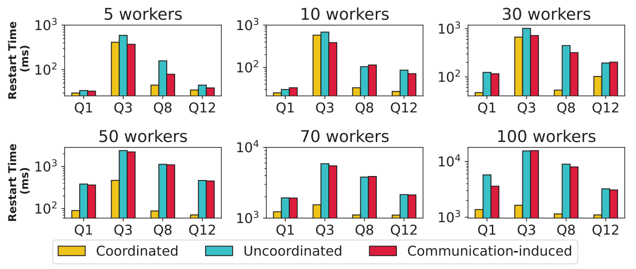
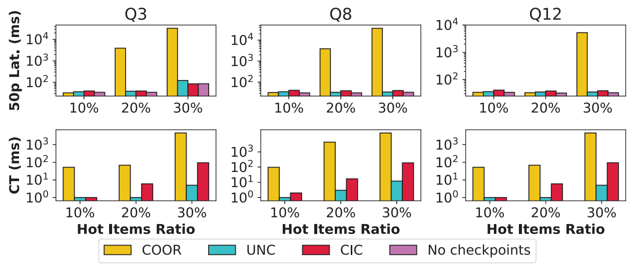


Fig. 11: Restart time after failure per query for each protocol on different levels of parallelism.

over the total checkpoints indicates how much the system rolled back. Low percentages show no domino effect and better utilization of the checkpointed state. The coordinated approach does not introduce any invalid checkpoints. Table III shows that for all the acyclic queries, the uncoordinated and communication-induced protocols introduce very few invalid checkpoints and result in similar total checkpoints. Overall, the uncoordinated and communication-induced protocols result in more checkpoints than the coordinated protocol since every operator independently decides when to take a checkpoint based on its worker's clock.

Skewed NexMark. Operating under a skewed workload usually results in workers straggling to process the excessive load they are responsible for. Although operating under such conditions is not preferable, avoiding it is not always feasible. Therefore, it is important to investigate how the different protocols perform under skew. To measure the impact of skew on the protocols' performance, we employ Q3, Q8, and Q12 under different hot item ratios provided by the NexMark generator. Q1 is not affected by skew as it involves non-keyed operations. Therefore, we omit it. We run Q3, Q8, and Q12 on 10 workers at 50% and 80% of the maximum sustainable throughput of the non-skewed execution of every protocol without introducing any failure. Both throughputs result in straggling workers. However, the latter stresses significantly more the system, resulting in fewer checkpoints taken and higher sensitivity to skew. We consider these settings representative of a real-world deployment, where overprovisioning is employed to handle spikes and unexpected skews. We employed three different hot item ratios to increase the skew gradually, from 10% to 30%. The straggling workers heavily affect the 99th percentile of latency, so we focus on the 50th percentile. We also report the average checkpointing time, as it is also heavily affected by the skew and can significantly affect the latency.

Unlike the non-skewed experiments, as illustrated in Figure 12, the coordinated protocol performs the worst regarding 50th percentile latency and average checkpointing time in both throughputs. With every increase in the hot items ratio, latency and checkpointing time increase by at least an order of magnitude for the lower throughput, while for the higher throughput, even the lowest skew ratio has a significant impact on Q3. The coordinated protocol is so heavily impacted by skew because not only are the straggling operators slow to take their checkpoints, but they also delay propagating their markers to downstream operators that block processing in other channels to wait for the delayed markers. Meanwhile, both UNC and CIC keep both metrics relatively low. In summary, the uncoordinated and communication-induced protocols can handle skew more effectively in every case.


(a) 50% of the MST of the non-skewed execution.

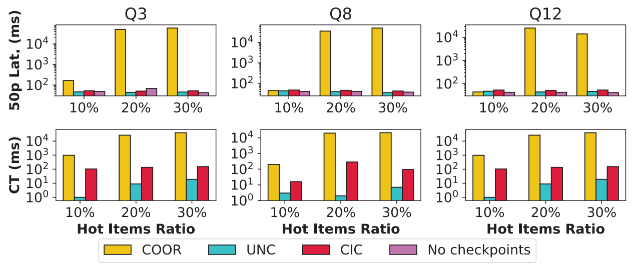

(b) 80% of the MST of the non-skewed execution.

Fig. 12: 50th percentile latency & average checkpointing time under different hot items percentages.

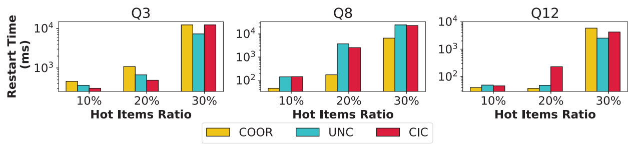


Fig. 13: Restart time after failure per query in the presence of skew.

Similar to the non-skewed experiments, we perform another run using the 50% MST, introducing a failure. Figure 13 shows the time needed to restart processing. Unlike the non-skewed experiments, where the coordinated outperformed the other approaches, the differences are mitigated under skew, and all protocols perform similarly. This is an immediate result of the coordination under skew with the stragglers. Invalid checkpoints remain the same under skewed and non-skewed conditions. We do not report recovery time since none of the protocols managed to recover within the time frame for 20% and 30% skew, while for 10% skew, the performance is similar to the non-skewed experiments.

Cyclic query. For the cyclic query, we only evaluate the *uncoordinated* and the *communication-induced* checkpointing protocols. The aligned version of the coordinated protocol cannot handle cyclic queries. That is because at least one operator would be waiting for a marker that originates from itself, thus leading to a deadlock.

TABLE IV: Average checkpointing time (CT), restart time (RT), and invalid checkpoints (IC) for the cyclic query.

	Uncoordinated			Communication-induced		
#Workers	CT	RT	IC	CT	RT	IC
5	0.01 ms	620 ms	1.4%	2.73 ms	347 ms	1.7%
10	1.38 ms	344 ms	1.4%	8.39 ms	399 ms	1.6%

We evaluate the protocols with two parallelisms, 5 and 10 workers. We refrain from using higher parallelisms since CIC is greatly affected by complex topologies and higher parallelism, as shown in Figure 7. For both deployments, we use the same configuration for our generator. It creates events with the following probabilities: 60% chance of creating a new link, 15% of creating a source node, 20% chance of deleting an existing link, and 5% of deleting an existing source node. The generator also assumes a static set of 1M nodes. We evaluate the two protocols with an input rate of 75% - 80% of their MST for the query. We run the experiments for 60 seconds and introduce a failure at the 48th second.

Regarding latency and maximum sustainable throughput, both protocols perform similarly to a checkpoint-free execution; therefore, we omit these metrics. We present the average checkpointing time, the recovery time, and the number of invalid checkpoints in table IV. Regarding average checkpointing time, the uncoordinated protocol is faster than the communication-induced protocol since the communication-induced protocol requires checkpointing additional protocol-related information apart from an operator's state. However, the difference between the two measurements is practically insignificant. The communication-induced protocol required less time to restart after a failure for a parallelism of 5 workers, as it forced checkpoints that led to fewer messages being prepared to be replayed. For 10 parallel workers, the uncoordinated protocol restarts slightly faster than the communication-induced protocol, although the difference is insignificant. Based on the literature and the core characteristics of both protocols, the uncoordinated protocol was expected to introduce many invalid checkpoints and lead to a domino effect. Although this might still hold in some extreme cases, our experiments show that both protocols unexpectedly share very similar percentages of invalid checkpoints for both parallelisms. Neither protocol outperforms the other when employed on top of cyclic queries in any meaningful aspect, and the uncoordinated protocol does not introduce a domino effect.

Summary. In our experiments, we explore three different cases: the NexMark queries with a uniformly distributed workload, the three more complex NexMark queries, i.e., Q3, Q8, and Q12 for a skewed input, and a cyclic query. In the first case, the coordinated approach outperforms the rest regarding latency, recovery time, and maximum sustainable throughput but has a significantly higher checkpointing time. Surprisingly, in contrast to the theoretical analysis, although parallelism and shuffling impact the checkpointing time of the coordinated protocol, they hardly affect the overall performance and only result in mild spikes in latency when a checkpoint is taken. Additionally, the uncoordinated protocol remains competitive in all queries and parallelisms. However, under skewed inputs, the uncoordinated greatly outperforms the coordinated one, which suffers both in terms of latency and checkpointing time. For the cyclic query, surprisingly, the uncoordinated does not showcase an increased number of invalid checkpoints (e.g., a domino effect) and performs slightly better than the communication-induced.

VIII. Related Work

This section presents the related work regarding *benchmarking* for stream processing systems and *experimental evaluation* of fault tolerance in stream processing.

Benchmarking for stream processing. Linear Road [12] is one of the first benchmarks proposed for stream processing that simulates a traffic monitoring application and evaluates the benchmarked solution in terms of latency, throughput, and accuracy. CityBench [9] and RioTBench [42] are real-time analytics benchmarks that employ real-world Internet of Things (IoT) data and extend the evaluation using metrics such as memory and CPU utilization and completeness of query results. SparkBench [35] is tailored to Apache Spark and targets CPU and memory utilization, network and disk I/O, job execution time, and throughput. NEXMark [46] is a widely adopted benchmark, also extended by Apache Beam [3], represents an e-commerce application, and provides streaming queries that cover all the fundamental processing workloads.

Experimental evaluation of fault tolerance. StreamBench [36] employs seven workloads on Spark and Storm and performs an evaluation focusing on throughput and latency. Qian et al. [41] evaluate fault tolerance, including additionally Samza and Kafka. However, their evaluation lacks representative workloads as they only consider a simple workload that consumes input and performs no operations.

IX. Conclusions

In this paper, we surveyed the three checkpoint protocol families for fault-tolerance in stream processing and discussed the theoretical advantages and drawbacks of each one of them. We developed an open-source testbed system that allows for isolated comparison of the approaches and performed a thorough experimental evaluation. While our experiments empirically confirmed the reasons behind the universal adoption of the coordinated approach, they also highlighted cases (e.g., skewed input) where the uncoordinated approach shows more robustness and better performance. Based on these results, we urge the research community to further research the uncoordinated approach since even a "vanilla" implementation of it was proven to perform well in uniformly distributed workloads, and it is the only viable solution for skewed workloads.

Acknowledgment

This publication is part of project number 19708 of the Vidi research program, partly financed by the Dutch Research Council (NWO). It is also partially funded by ICAI AI for Fintech Research Lab. We would like to thank Gianni Wiemers for his contribution to the early stages of development.

References

[1] From Aligned to Unaligned Checkpoints - Part 1: Checkpoints, Alignment, and Backpressure. https://flink.apache.org/2020/10/15/from-aligned-to-unaligned-checkpoints-part-1-checkpoints-alignment-and-backpressure/. Accessed: 2023-11-20.

[2] Improving Speed and Stability of Checkpointing [...]. https://www.alibabacloud.com/blog/599048. Accessed: 2023-11-20.

[3] Nexmark benchmark suite. https://beam.apache.org/documentation/sdks/java/testing/nexmark/. Accessed: 2023-11-20.

[4] Optimize checkpointing in your Amazon Managed Service for Apache Flink applications. https://aws.amazon.com/blogs/big-data/part-1-optimize-checkpointing-in-your-amazon-managed-service-for-apache-flink-applications-with-buffer-debloating-and-unaligned-checkpoints/. Accessed: 2023-11-20.

[5] Stateful Stream Processing. https://nightlies.apache.org/flink/flink-docs-release-1.13/docs/concepts/stateful-stream-processing/#exactly-once-vs-at-least-once. Accessed: 2023-11-20.

[6] Daniel Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. volume 5, pages 277–289, 01 2005.

[7] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[8] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

[9] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In Marcelo Arenas, Oscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015*, pages 374–389, Cham, 2015. Springer International Publishing.

[10] L. Alvisi, E. Elnozahy, S. Rao, S.A. Husain, and A. de Mel. An analysis of communication induced checkpointing. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, pages 242–249, 1999.

[11] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. STREAM: the stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[12] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 480–491. VLDB Endowment, 2004.

[13] Magdalena Balazinska, Jeong Hwang, and Mehul Shah. *Fault Tolerance and High Availability in Data Stream Management Systems*, pages 1–8. 01 2017.

[14] B. Bhargava and Shu-Renn Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. In *Proceedings [1988] Seventh Symposium on Reliable Distributed Systems*, pages 3–12, 1988.

[15] Bharat Bhargava and Shy-Renn Lian. Independent checkpointing and concurrent rollback for recovery in distributed system—an optimistic approach. 1987.

[16] D. Briatico, Augusto Ciuffoletti, and Luca Simoncini. A distributed domino-effect free recovery algorithm. In *Fourth Symposium on Reliability in Distributed Software and Database Systems, SRDS 1984, Silver Spring, Maryland, USA, October 15-17, 1984, Proceedings*, pages 207–215. IEEE Computer Society, 1984.

[17] Guohong Cao and M. Singhal. On coordinated checkpointing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1213–1225, 1998.

[18] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, aug 2017.

[19] Paris Carbone, Asterios Katsifodimos, † Kth, Sics Sweden, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38, 01 2015.

[20] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, and James F. Terwilliger. Trill: Engineering a library for diverse analytics. *IEEE Data Eng. Bull.*, 38(4):51–60, 2015.

[21] Badrish Chandramouli, Jonathan Goldstein, and David Maier. On-the-fly progress detection in iterative stream queries. *Proc. VLDB Endow.*, 2(1):241–252, aug 2009.

[22] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *First Biennial Conference on Innovative Data Systems Research, CIDR 2003, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*. www.cidrdb.org, 2003.

[23] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, feb 1985.

[24] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. 01 2003.

[25] Gabriela Jacques da Silva, Fang Zheng, Daniel Debrunner, Kun-Lung Wu, Victor Dogaru, Eric Johnson, Michael Spicer, and Ahmet Erdem Sariyüce. Consistent regions: Guaranteed tuple processing in IBM streams. *Proc. VLDB Endow.*, 9(13):1341–1352, 2016.

[26] Om P Damani and Vijay K Garg. How to recover efficiently and asynchronously when optimism fails. In *Proceedings of 16th International Conference on Distributed Computing Systems*, pages 108–115. IEEE, 1996.

[27] Elmootazbellah Elnozahy, Lorenzo Alvisi, Yi-min Wang, and David Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34, 06 2002.

[28] Raul Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. pages 725–736, 06 2013.

[29] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems. *VLDB Journal*, 2023.

[30] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz, Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, Mehmet Doğan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. Hazelcast jet: Low-latency stream processing at the 99.99th percentile. *Proc. VLDB Endow.*, 14(12):3110–3121, jul 2021.

[31] Jean-Michel Hélary, Achour Mostéfaoui, Robert H. B. Netzer, and Michel Raynal. Communication-based prevention of useless checkpoints in distributed computations. *Distributed Comput.*, 13(1):29–43, 2000.

[32] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *21st International Conference on Data Engineering (ICDE'05)*, pages 779–790, 2005.

[33] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, Carlsbad, CA, October 2018. USENIX Association.

[34] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518, 2018.

[35] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, New York, NY, USA, 2015. Association for Computing Machinery.

[36] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In

Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC '14, page 69–78, USA, 2014. IEEE Computer Society.

[37] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. Regular path query evaluation on streaming graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1415–1430, New York, NY, USA, 2020. Association for Computing Machinery.

[39] K. Psarakis, W. Zorgdrager, M. Fragkoulis, G. Salvaneschi, and A. Katsifodimos. Stateful entities: Object-oriented cloud applications as distributed dataflows. In *EDBT*, 2024.

[40] Kyriakos Psarakis, George Siachamis, George Christodoulou, Marios Fragkoulis, and Asterios Katsifodimos. Styx: Transactional stateful functions on streaming dataflows, 2024.

[41] Shilei Qian, Gang Wu, Jie Huang, and Tathagata Das. Benchmarking modern distributed streaming platforms. In *2016 IEEE International Conference on Industrial Technology (ICIT)*, pages 592–598, 2016.

[42] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257, 2017. e4257 cpe.4257.

[43] G. Siachamis, J. Kanis, W. Koper, K. Psarakis, M. Fragkoulis, A. van Deursen, and A. Katsifodimos. Towards evaluating stream processing autoscalers. In *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*, pages 95–99, Los Alamitos, CA, USA, apr 2023. IEEE Computer Society.

[44] Pedro F. Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. Clonos: Consistent causal recovery for highly-available streaming dataflows. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1637–1650, New York, NY, USA, 2021. Association for Computing Machinery.

[45] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, aug 1985.

[46] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. Nexmark–a benchmark for queries over data streams (draft). Technical report, 2008.

[47] Yi-Min Wang, Pi-Yu Chung, In-Jen Lin, and W.K. Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):546–554, 1995.