STRETCH: Scalable and Elastic Deterministic Streaming Analysis with Virtual Shared-Nothing Parallelism

Hannaneh Najdataei, Yiannis Nikolakopoulos
Marina Papatriantafilou, Philippas Tsigas, Vincenzo Gulisano
Chalmers University of Technology
{hannajd,ioaniko,ptrianta,tsigas,vinmas}@chalmers.se

1 INTRODUCTION

Data streaming builds on efficient one-pass analysis of unbounded streams of tuples. It is widely adopted thanks to two decades of research results and thanks to open-source Stream Processing Engines (SPEs) [7, 10, 27, 28]. When such analysis is stateful, its resulting output tuples can depend on arbitrarily-long portions of the input.

In the literature, many solutions study how to deploy stateful analysis and efficiently leverage multi-core architectures by means of parallelism and elasticity (i.e., by multi-threaded execution in which resources as threads are adjusted over time) [4, 5, 9–12, 15, 25, 29]. Such techniques focus on optimizing the parallelization of specific operators [9, 11, 12, 25, 29], managing the distributed state of parallel operators [4, 5], providing operator-oblivious parallelization and elasticity [5, 10] or guaranteeing deterministic execution [8, 12, 30], among other aspects.

Challenges. Existing SPEs provide good support for parallelization and elasticity of simple stateful analysis (e.g., continuous per-key summation), but leave to users how to use their APIs efficiently when programming complex stateful analysis. Consider, as a motivating example, a user trying to parallelize the analysis of an application that tries to find, on a per-hash tag basis, how many times an ordered sequence of words is found within or across consecutive tweets. Intuitively, the user could parallelize the application by assigning the analysis of tweets carrying different hashtags to distinct threads. Key-by partitioning, provided by Apache Flink [7] (or simply Flink) and Apache Storm [28], could not be directly leveraged, though, for tweets carrying two or more hashtags assigned to different threads. Users would need to either create single-hash tag copies of tweets if the latter carry multiple hashtags (and then use key-by-partitioning) or define a custom tuple-to-thread assignment scheme. In both cases, data duplication would unnecessarily hampers performance if the threads share memory and could instead access the same copy of each tweet. To complicate the matter further, if the SPEs arbitrarily interleave tuples forwarded to a parallel thread from multiple streams, the user would also need to program how to deterministically merge such streams (e.g., sort them on a per-tuple basis [12] or with watermarks [1], in order to prevent the arbitrary interleaving from affecting whether the ordered sequence of words is found or not. Finally, the user might also need to program how to serialize/deserialize the state of the analysis run in parallel for the SPE to trigger reconfigurations (e.g., provisioning or decommissioning of threads).

Addressing efficiently these connected challenges, that hold also for other stateful streaming operators such as joins, is not trivial for the average programmer. With our work, we aim at advancing the front of SPEs’ automation and tools available to end users.
Contributions. These challenges have conflicting needs wrt “right amount of sharing” to (i) enhance independence among threads, as well as to (ii) enable efficient coordination for consistent redistribution of work when needed, while (iii) supporting determinism. The implied trade-offs relate to the efficiency in synchronization and state sharing among threads: on one hand, shared-nothing processing maximizes parallelism, but is costly in reconfigurations and in making copies of the data when needed; on the other hand, sharing processing state might introduce contention in maintaining parallelism, but facilitates the workload shifting among threads by not needing state transfer protocols. Based on this, our motivating research question is: Can an intra-node streaming framework take the best of two worlds, shared-nothing and shared-memory parallelism, (i) allowing users to program parallel and elastic stateful operators, (ii) without partitioning but rather sharing input tuples with all threads and specify which ones the latter should process or ignore, while (iii) supporting deterministic execution and (iv) ensuring high efficiency in terms of throughput, latency and reconfiguration times?

We propose virtual shared-nothing parallelisation and provide a framework leveraging it. The framework, called STRETCH, manages efficient processing for interconnected streaming operators, supporting determinism even with varying degree of parallelism. In more detail, the contributions include (i) a generalization of previous results (e.g., [12, 13]) in supporting efficient sharing and synchronization among parallel threads, building on ScaleGate, an established object for communication among operators in SPEs, which has been shown to facilitate efficient deterministic fine-grained stream processing; (ii) a novel, virtual shared-nothing state manager that provides to each thread exclusive access to a portion of an operator state while also allowing to efficiently change ownership of such portions at runtime for elastic reconfigurations; (iii) an extended API for ScaleGate, which we call ESG (elastic ScaleGate) and the algorithmic implementation for it, to allow dynamic number of threads accessing it, as well as a protocol describing the interactions between ScaleGate objects and the state manager for load balancing, thread provisioning and thread decommissioning; (iv) correctness proofs for the determinism guarantees of the methods and a model for the performance of virtual shared-nothing parallelism; and (v) an extensive evaluation, empirically validating our model, as well as showing that with the proposed methods, fast work redistribution is possible, with minimal overhead both in latency and throughput.

The paper is organized as follows: § 2 introduces preliminary concepts, § 3 presents our system model and objectives. We outline STRETCH in § 4 and provide algorithmic implementation details and correctness arguments in § 5, § 6 and § 7. We model the behavior of virtual shared-nothing parallelism comparing it also with shared-nothing parallelism in § 8 and evaluate the model and STRETCH in § 9. We discuss related work in § 10 and conclude in § 11.

2 PRELIMINARIES

Data streaming. A data streaming continuous query (or simply query in the remainder) is a directed acyclic graph (DAG) of streams (carrying information) and operators (manipulating such information). A stream is an unbounded sequence of tuples sharing the same schema composed by attributes $\langle ts, A_1, \ldots, A_n \rangle$. Attribute $t.ts$ represents the (event) time at which the tuple has been created. An operator is the minimum processing unit that defines at least one input stream (delivering the tuples to be processed) and one output stream (to forward the output tuples it produces).

Tuples sharing the same schema and being input to the same operator can come from multiple sources or operators. Hence we distinguish between physical and logical streams. The former represents one stream between a pair of operators while the second represents the set of streams defining the same schema and carrying the same type of information to the same operator. Following a common assumption in the streaming literature (e.g., [2, 10–12]), we assume that each physical stream contains timestamp-sorted tuples. If this is not the case, sorting tools such as [20] can be leveraged. We use the term stream without specifying whether it is logical or physical if it can be deduced by the context.

Streaming operators are distinguished into stateless and stateful. Such a stateful operator, which we also use later in the paper, is the Join [12]: it defines a left ($L$) and a right input stream ($R$) and produces output tuples combining the attributes of tuples $t_L \in L$ and $t_R \in R$, for each pair of tuples $\langle t_L, t_R \rangle$ satisfying a given predicate and being closer in time than a given window size WS (i.e., $|t_L.ts - t_R.ts| \leq WS$). We assume that the timestamp of an output tuple $t_o$ produced by a stateful operator is equal to that of the latest processed tuple that triggers the output of $t_o$ (additional timestamps related to the tuples contributing to $t_o$ can be other attributes of $t_o$’s schema).

Determinism. Deterministic execution of a sequential operator requires that each processing step depends on the notion of event time carried by the tuples themselves (attribute $ts$) and is affected neither by the latency incurred in transmitting tuples from an operator to another operator nor by the interleaving of tuples to an operator with multiple input streams. For a parallel operator, determinism is enforced when its results (for the same sequence of input tuples) are equivalent to those of its sequential counterpart [10–12]. As explained in [11, 12], a sufficient condition for determinism for both cases is to require merging the timestamp-sorted physical streams delivering tuples to a stream and processing such tuples in timestamp-order once they are ready, as defined in [11]: $t^i_j$, being the $i$-th tuple from timestamp-sorted stream $j$, is ready to be processed when $t^i_j.ts \leq merge_{t_j}$, where $merge_{t_j} = min_k(max(l^k_j.ts))$ is the minimum among the latest $l$ tuple timestamps, one from each timestamp-sorted stream $k$. ScaleGate [11, 12] is a shared data object, leveraged and extended in this work, that (i) efficiently supports concurrent deterministic merging of timestamp-sorted streams into a timestamp-sorted stream of ready tuples, while (ii) allowing a number of reader entities to consume all the ready tuples of the latter stream. Its lock-free algorithmic implementation was shown to facilitate efficient deterministic processing [11, 12, 30, 31].

Load Balancing and Elasticity. As discussed in [10], the computational cost of a streaming application varies over time, depending on the rate with which input tuples are fed to it and depending on the tuples’ data distribution. Because of this, a parallel execution in which the distribution of work to threads is statically decided at deploy time can lead to imbalances in the work of threads. When
the overall work is unbalanced but could be carried out by the available threads as a whole, a load balancing adaptive reconfiguration is needed to change the work distribution. If more threads are to be provisioned, the new work distribution should also assign some work to the newly allocated threads. Notice that it is essential to adjust resources such as threads, since over-provisioned systems can lead to high latency [30] or unnecessary costs [10]. Because of this, elastic reconfigurations should also be triggered when the work can be done by fewer threads (independently of whether they are unbalanced), thus decommissioning some threads and changing the work distribution. We use the term reconfiguration to refer to any of these.

3 PROBLEM MODELING AND OBJECTIVES

State, buckets and streaming parallelization model. We target a general-purpose intra-node streaming tool, where users can implement parallel stateful analysis, without explicit handling of complexities inherent to determinism or fast reconfigurations. We adopt a known parallelism model of the literature [7, 8, 10, 28], that allows users to define and maintain the state of a streaming operator over a set of buckets. Each bucket, a fine-grained portion of the operator’s state, can be accessed and updated (based on the tuples being processed) by one of the threads that run instances of the operator. The number of buckets of an operator is commonly chosen to be greater than that of the maximum threads that can be in charge of running its instances. The portion of state assigned to each thread is thus a partition of the operator’s buckets. In the following, M refers to a buckets-to-threads mapping function, where M[k] denotes the thread id to which bucket k is assigned to.

In relation to the twitter example (§ 1), the state of an operator running such analysis could consist of per-hash index (of the next word to find) and counters (of sequences matched so far). To maintain such state, the user could assign each hash index to exactly one bucket (e.g., using a hash-function) and parallelize the analysis by letting each parallel thread update the indexes and counters of hashtags contained in the buckets assigned to it.

Notice that the state of an operator can depend on all the tuples observed so far, or on a window of them, as in the case of stream Joins (§ 2). In this sense, the problem does not impose any restriction on the length of such portion. As discussed in § 1, we aim at a parallelization approach in which users do not need to partition input tuples to threads (as in key-by-partitioning). In turn, this does not limit the problem to a type of parallelism in which each parallel thread runs the same analysis on different portions of input data. The stream join we use as one example (§ 4.3) presents such a case, in which all parallel threads carry out some of the processing for all input tuples.

Execution epochs. Under the assumption that all threads are fed with the same sequence of tuples, a reconfiguration implies a change in M to hold true from a certain tuple onward. We use the term epoch to refer to the period spanning tuples in between two event times (i.e. between timestamps of a pair of tuples), during which the mapping of buckets to threads does not change. Hence, being E_i the current epoch, T_i^p the set of processing threads and M_i the mapping in E_i, a reconfiguration implies the beginning of a new epoch E_{i+1} for which a new mapping M_{i+1} is used for a (possibly different) set of processing threads T_{i+1}^p.

Problem statement. Our goal is a framework that can facilitate the programming of stateful analysis when efficient parallelism cannot be simply achieved by partitioning input tuples to threads (e.g., as in key-by-partitioning). From an SPE perspective, and with the goal of combining the benefits of shared-nothing and shared-memory approaches, we aim at designing and implementing a framework for intra-node parallel and reconfigurable stateful analysis with the following objectives:

O1 A programmable interface that does not require thread-safe programming of stateful analysis (i.e., as in shared-nothing parallelism), thus granting exclusive read and write access to a portion of the operator’s state (i.e., a portion of the buckets maintaining it) to each parallel processing thread.
O2 Support for deterministic sharing of all input tuples to all processing threads in the same order (without requiring the user to define any input tuple partitioning scheme) and deterministic merging for all threads fed by multiple streams.
O3 Support for fast reconfigurations.

We do not place restrictions on the logic with which reconfiguration actions are taken. Instead, we assume the existence of an external orthogonal policy in charge of triggering the reconfigurations. One such policy, triggering reconfigurations based on the threads’ CPU consumption is used in our evaluation (§ 9).

Notice that, in contrast to objective O3, which can be only evaluated empirically, O1 and O2 can be met if a set of sufficient properties is satisfied [10–12, 30], distinguished into intra-epoch and inter-epoch ones (i.e. to hold within each epoch, respectively when transitioning from E_i to E_{i+1}). Intra-epoch properties to be enforced are:

P1 All threads observe all input tuples in the same order.
P2 For a given mapping M_i, each thread has exclusive read/write access to the bucket(s) mapped to it.
P3 The tuples received by any thread from multiple streams are merged into a sequence of tuples processed by the thread once ready.

Additional, inter-epoch properties to be enforced are:

P4 If a bucket is mapped to a processing thread p_i in epoch E_i (i.e., if such a bucket is potentially modified based on the tuples processed by p_i) and to processing thread p_{i+1} in epoch E_{i+1}, then the first tuple belonging to E_{i+1} is processed by p_{i+1} after the last tuple belonging to E_i is processed by p_i.
P5 Each reconfiguration takes place atomically (either by being applied in its entirety or not being applied).
P6 Tuples sharing the same timestamp belong to exactly one epoch.

4 OVERVIEW OF STRETCH

Figure 1 outlines STRETCH’s overall architecture, utilising as main components the State Manager (SM) and Elastic ScaleGate (ESG) objects. For ease of presentation of the STRETCH and SM’s architecture and functionality, before outlining them in § 4.2, we outline the ESG’s API and functionality in § 4.1. In § 4.3, we provide an example implementation for a join operator, which we also use in the experimental evaluation.
4.1 The Elastic ScaleGate (ESG) data object

As mentioned in § 2, the Elastic ScaleGate, ESG, extends ScaleGate, which allows (i) a number of source threads to each insert in it a timestamp-sorted stream of tuples, and (ii) a number of reader threads to retrieve ready tuples from it in timestamp order, through the methods addTuple and getNextReadyTuple, respectively. They both encapsulate the necessary communication between sources and readers, to know whether a tuple is ready or not.

- addTuple(tuple, sID): allows a tuple from source sID to be merged by ScaleGate in the resulting timestamp-sorted stream of tuples.
- getNextReadyTuple(rID): provides to the calling reader rID the next earliest ready tuple that has not been yet consumed by rID. Note that each tuple, once it becomes ready, will be returned to all readers invoking the method.

For ESG, the API extension to enable changes in the sets of threads is listed below, outlining the additional methods and their behaviour, while their algorithmic implementation is described in § 7.

- announceReaders(List reader_IDs, rID): can be invoked by an existing ESG reader rID. The new readers can return ready tuples starting from the one that rID returned before calling this method.
- removeReaders(List reader_IDs): removes the denoted list of readers from ESG.
- announceSources(List source_IDs, min_ts): adds new source threads. To comply with the requirements for the identification of ready tuples, the method expects min_ts, the earliest timestamp the new sources can add, to be greater than the timestamp of the latest tuple retrieved by any reader when the method is invoked.
- removeSources(List source_IDs): removes from ESG the list of sources source_IDs. Any potential new tuple insertions by the latter will be ignored.

For all the above, among concurrent invocations and subsequent invocations with the same parameters, only one succeeds.

4.2 The STRETCH framework’s architecture

Recalling from Figure 1, STRETCH uses a State Manager (SM) and two ESG objects, i.e. one for input tuples (ES Gin) and one for output tuples (ESGout). Several threads interact with these components:

- input threads deliver input tuples to ESG in, processing threads process input tuples and interact with SM, delivering results to ESG out, while output threads retrieve the output tuples delivered to ESG out.
- Despite not discussed for simplicity, STRETCH also defines a thread pool for provisioning and decommissioning. For ease of explanation, we focus our description on the parallel, elastic and deterministic execution of one stateful operator. The description extends for multiple operators, considering that the input (resp. output) threads of an operator are the processing threads of its upstream (resp. downstream) peers.

To instantiate an operator, the STRETCH user initially provides:

\[
\{ M_0, \tau_0^I, \tau_0^P, \tau_0^O, \text{BucketImpl, filter} \}
\]

\(M_0\) defines the overall number of buckets and their initial mapping to the processing threads of epoch \(E_0\), while \(\tau_0^I, \tau_0^P\) and \(\tau_0^O\) are the sets of input, processing and output threads for epoch \(E_0\), respectively.\(^1\) At runtime, each bucket is an instance of the BucketImpl class, defined by the user to implement the stateful analysis’ logic. The class BucketImpl is expected to define a method process (to be invoked by \(\tau_0^I\)). The filter function, for each thread \(p\) and each tuple \(t\) the thread works on, aims at filtering out the buckets of \(p\) that need not be updated due to \(t\). Shortly in this section we outline how filter can be used to speed-up the analysis. Once a stateful operator is instantiated, STRETCH defines the method:

\[
\text{switchEpoch}(M)
\]

which will change the mapping of buckets to threads (and the number of threads, if the reconfiguration is provisioning or de-commissioning them). STRETCH relies on special tuples, named epoch-switch tuples to perform an epoch switch.

Each bucket is assigned to exactly one thread based on \(M\). \(T^I\) (Alg. 1) are the source entities for ESG in, \(T^P\) (Alg. 2) are the reader and the source entities for ESG in and ESG out, respectively, and \(T^O\) (Alg. 3) are the reader entities of ESG out.

![Figure 1: Overview of STRETCH.](image)

**Algorithm 1: Input threads (\(T^I\)) - main loop**

1. while executing do
2. \(\text{retrieve / produce next tuple } t\)
3. \(\text{add } t \text{ to } ESG_{in}\)

\(T^I\) threads deliver each tuple (e.g., retrieved from the network or another operator) to ESG in (Alg. 1, L 1-3). At the same time, each \(T^P\) thread \(p\) retrieves each next ready tuple \(t\) (Alg. 2, L 2) and checks whether \(t\) is an epoch-switch tuple or a regular one (Alg. 2, L 3). In the former case, \(p\) stores \(t\) to later trigger a reconfiguration (Alg. 2, L 4), at an appropriate time-point, to ensure determinism. In the latter case, the norm is to invoke process if needed; however, \(p\) first checks if \(t\) signifies the appropriate time-point to trigger reconfiguration, i.e., it checks whether there exists some previously stored epoch-switch tuple yet to be processed and if \(t\)’s timestamp is greater than (i.e. not equal) that of the previous regular tuple (prev_ts). If so, \(p\) requests a new epoch, synchronizing with both ESGs and the SM (we provide details on synchronization in § 6) and retrieves the buckets mapped to it in such new epoch (Alg. 2, L 8-10).

\(^1\)We will skip index \(i\) for \(T^I, T^P, T^O\), \(M\) in contexts not focusing on a specific epoch.
4.3 Example: **STRETCH**-implemented Join

ScaleJoin is a stream join that performs deterministic and efficient parallel stream processing. As described in detail in [12], in its parallelization approach, each of the \( n \) processing threads is responsible for running approx \( 1/n \) of the overall comparisons incurred by each input tuple. This is achieved by having all processing threads process each tuple but exactly one maintaining it in its local state, thus being responsible for the comparisons of future tuples with it. In **STRETCH**, this strategy can be implemented by having exactly one bucket (and thus exactly one thread) responsible for storing each tuple. Alg. 4 presents how the BucketImpl class can implement ScaleJoin’s semantics. Whenever process is invoked, the tuple is used to purge the opposite window, check the predicate against the tuples of the opposite window and eventually adds itself to its window if the counter modulo the number of buckets \( (B) \) is equal to the thread id. Since each input tuple needs to be compared with all the tuples stored in any bucket, the filter function \( (\text{Alg. 5}) \) returns the entire set of buckets of the processing thread. Alg. 4, once run by **STRETCH**, guarantees the join semantics, since:

1. The method process of each bucket, is invoked for all the tuples taken from \( ESG_{in} \) in the exact order in which such tuples are retrieved and is never invoked concurrently (for a given bucket) by two or more threads (Theorem 5.1).
2. By (1) we have that all buckets update the counter consistently.
3. By (1) and (2) we have that each tuple is stored in exactly one bucket.
4. By (1) and (3) we have that each stored tuple is compared with all the tuples needed, according to the join semantics.

**Algorithm 5:** filter function for ScaleJoin [12]

```
1 Function BucketImpl[] filter(BucketImpl[] b, Tuple t)
2 return b
```

5 **INTRA-EPOCH PROCESSING**

We detail here how the data structures and threads presented in § 4 interact within each epoch, i.e., for a given mapping \( M \) we first introduce the API of SM and then show how, by the processing defined for the processing threads, **STRETCH** satisfies the intra-epoch properties listed in § 3, namely that all processing threads \( T^P \) (i) observe all input tuples in the same order \( (P_1) \), each with exclusive read/write access to its buckets \( (P_2) \) and (ii) produce streams of output tuples that are deterministically merged into a logical sequence of output tuples \( (P_3) \). The description is later extended in § 6 for inter-epoch processing. SM’s API methods are:

- setup(\( M_0, \text{BucketImpl} \)) initializes the SM associated to the stateful operator. Based on \( M_0 \), SM knows how many buckets should be maintained and the thread id to which each one is assigned to initially.
Algorithm 6: SM implementation

1 BucketImpl[] buckets
2 Function setup(M₀, BucketImpl)
3   store M₀
4   for i = 1...size(M₀) do
5     store new BucketImpl instance in buckets[i]
6 Function BucketImpl[] getBuckets(Thread id)
7   BucketImpl[] threadBuckets
8   for i = 1...size(M) do
9     if M[i]==id then
10        add a pointer to buckets[i] to threadBuckets
11 return threadBuckets

5.1 Enforcing properties P1-P3 in E₀

At this point, we argue that STRETCH satisfies properties P1-P3 during the first epoch E₀ (i.e., from the moment a certain stateful operator is deployed, to its first reconfiguration or for the entire execution of a parallel operator with a static mapping of buckets to threads). It should be noticed that, since no reconfiguration is defined before E₀, the behavior of the two ESGs is equivalent to that of the base ScaleGate. This argumentation is later extended to any epoch Eᵢ by induction, after showing that properties P4-P6 are met when transitioning across epochs.

Theorem 5.1. STRETCH satisfies properties P1-P3 in E₀. □

Proof. (Sketch) Property P1 is satisfied by leveraging the ESGin and Alg. 2, since the former delivers all input tuples (once ready) in the same order to all Tᵢ^P threads while the latter does not discard any input tuple. Property P2 is enforced by Alg. 2 and the SM’s implementation (Alg. 6) since the former retrieves the buckets exactly once (upon processing of the first tuple) while the latter returns each bucket to one and only one thread in Tᵢ^P. Property P3 is satisfied because each thread in Tᵢ^P delivers a non-decreasing timestamp-sorted stream of output tuples and the merging of such streams is carried out deterministically by ESGout.

6 INTER-EPOCH PROCESSING

Here we describe how STRETCH transitions from one epoch to another, while guaranteeing properties P4-P6 (§ 3). We first give a high-level description of the protocol and later provide more detail and we prove that properties P4-P6 are met while switching from epoch Eᵢ to epoch Eᵢ₊1, thus extending Theorem 5.1 to any epoch Eᵢ.

At this point recall that input tuples can be of type regular or epoch-switch. The former refers to regular tuples, the latter refers to special control tuples used by STRETCH when switching epochs. In a nutshell, when a special epoch-switch tuple t∗ is received in Eᵢ by the Tᵢ^P threads, the epoch switch protocol is triggered as soon as the first regular tuple t with a timestamp greater than the latest timestamp observed before t∗ is received (as also shown in Alg. 2 L 8-10).

As we further elaborate in the following, this implies that property P6 holds. Independently of the nature of the switch from epoch Eᵢ to epoch Eᵢ₊1 (i.e., decommissioning, load balancing or provisioning), all Tᵢ^P threads invoke the method requestNewEpoch(Mapping M) of SM and block there until the method returns. When threads are provisioned, one of the current Tᵢ^P threads activates the necessary new threads from the thread pool and connects them with ESGout and ESGin. When threads are decommissioned, the latter are disconnected from ESGin and ESGout and are returned to the thread pool.

6.1 Switching epochs

As outlined in § 4, STRETCH provides the method switchEpoch(M) to express the intention of switching the current epoch Eᵢ to the epoch Eᵢ₊1, in which mapping M is enforced. When this function is invoked, an epoch-switch tuple carrying M is inserted in ESGin by each Tᵢ^P thread. For each Tᵢ^P thread, the timestamp of such epoch-switch tuple is set to that of the latest tuple added to ESGin by the Tᵢ^P thread. This, combined with the definition of ready tuples, implies that at least one of these epoch-switch tuples is immediately ready for Tᵢ^P threads process.

As shown in Alg. 2, all epoch-switch tuples retrieved by a thread in Tᵢ^P are initially stored in its local list pendingEpochSwitchTuples (L 4). At any execution point, there could be many epoch-switch tuples stored by Tᵢ^P threads, either referring to the same switchEpoch invocation (remember all Tᵢ^P threads forward one such tuple) or to different invocations of the switchEpoch method (if the latter is invoked when pending epoch transitions are still to be completed). Each Tᵢ^P thread checks whether one or more epoch-switch tuples are in pendingEpochSwitchTuples, only when an incoming regular tuple t with a timestamp greater than (i.e., not equal to) the previous one is retrieved from ESGin (§ 4). Since all Tᵢ^P threads retrieve all tuples in the same order from ESGin, all Tᵢ^P threads have the same set of epoch-switch tuples with timestamp equal to or smaller than tᵢ in their pendingEpochSwitchTuples lists at the time tᵢ is processed. Because of this, if one or more epoch-switch tuples exist in pendingEpochSwitchTuples upon processing of tᵢ, the most recent epoch-switch tuple tᵢ refers to Eᵢ (i ≥ i) is processed (if any), while the rest of them are discarded.

Alg. 7 presents the steps followed by the Tᵢ^P threads to switch epoch. First, the most recent unprocessed epoch-switch tuple tᵢ is
Algorithm 7: Switching epoch protocol (for a thread in $T^p_i$ upon retrieving of $T$)

1. retrieve most recent epoch-switch tuple $t^*$ to be processed from pendingEpochSwitchTuples and discard the rest
2. if $t^*$ refers to $E_j$ where $j \geq i$ then
3.   $SM.requestNewEpoch(t^*.M)$. // blocking call
4. else if $t^*.M$ requests provisioning of threads then
5.   $ESG_{out}.announceSources(\text{new threads ids}, T, ts)$
6.   $ESG_{in}.announceReaders(\text{new threads ids, this thread id})$
7. else if $t^*.M$ requests decommissioning of threads then
8.   $ESG_{in}.removeReaders(\text{removed ids})$
9.   $ESG_{out}.removeSources(\text{removed ids})$

Algorithm 8: Method requestNewEpoch(M) (SM)

1. Method requestNewEpoch(Mapping $M$)
2.   block until the union of processing threads for this epoch invokes this method
3.   use $M$ as mapping from now on

retrieved and the rest of epoch-switch tuples are discarded from the pendingEpochSwitchTuples. Then, if $t^*$ belongs to $E_j$ where $j \geq i$, the blocking method requestNewEpoch of SM (Alg. 8) is invoked by all $T^p_i$ threads passing the new mapping $t^*.M$. This method will change the mapping used by $SM$ to $t^*.M$ as soon as all the $T^p_i$ threads have invoked the method. If the set of processing threads defined by $t^*.M$ is larger than that of the current epoch (i.e., if threads are to be provisioned), then these are announced by all $T^p_i$ threads as sources to $ESG_{out}$ and readers to $ESG_{in}$. Alternatively, if the set of $T^p_i$ threads defined by $t^*.M$ is smaller than that of the current epoch (i.e., if threads are to be decommissioned), methods removeReaders and removeSources are invoked for $ESG_{in}$ and $ESG_{out}$, respectively.

6.2 Satisfying properties P4-P6 from $E_i$ to $E_{i+1}$

Theorem 6.1. STRETCH satisfies properties P4-P6 when switching from $E_i$ to $E_{i+1}$.

Proof. (Sketch) In order to prove P4 is satisfied, it should be noted that, based on Algs. 2, 7 and 8, the following invariants hold:

1. $\exists$ unique regular tuple $T$ that is seen in the same relative position in the stream by all threads of $E_i$ and that distinguishes epochs (i.e., mapping of buckets to threads); $T$ is the first tuple of the new epoch. Specifically:
   (a) $\forall t' \forall t'.ts < T.ts, t' \in \text{old epoch}$
   (b) $\forall t'' | t''.ts \geq T.ts, t'' \in \text{new epoch}$
2. Let $\text{out}(t)$ denote the set of output tuples triggered by a tuple $t$. Then:
   (a) $\forall t' | t'.ts < T.ts, \text{out}(T)$ is read by threads $T^O$ after $\text{out}(t')$
   (b) $\forall t'' | t''.ts \geq T.ts, \text{out}(T)$ is read by threads $T^O$ after $\text{out}(T)$.

Moreover, such $T$ is chosen to be the first regular tuple with a timestamp greater than (i.e., not equal to) the previous one (observe that $T$, as all tuples in $ESG$, is seen by all $T^p_i$). This, together with the assumption on output tuples timestamps and the fact that $ESG$ preserves ordering, implies P6. Finally, note that, while multiple epoch-switch tuples can be stored at the same time, two threads in $T^p_i$ cannot be more than one epoch away because of the blocking method requestNewEpoch; this implies P5.

6.3 Satisfying properties P1-P3 in $E_i$, $\forall i > 0$

Theorem 6.2. $\forall i > 0$, STRETCH satisfies properties P1-P3 in $E_i$.

Proof. (Sketch) All the $T^p$ threads of epoch $E_i$ process all tuples in $E_i$. This is because of one of the following cases: (i) If threads have been provisioned for $E_i$, the new readers can return ready tuples starting from the latest ready tuple gotten by the calling reader that succeeded to execute announceReaders, which is $T$ (Alg. 7 precondition (caption), and L 5-6). Hence the new readers will retrieve their assigned buckets before processing $T$ (Alg. 2 L 7). The new threads are also already registered as sources to $ESG_{out}$ (Alg. 7 L 5-6), so if the processing of $T$ or any subsequent tuple triggers any output tuple, the latter will be deterministically delivered by $ESG_{out}$ once ready. (ii) Alternatively, if threads have been decommissioned, threads only existing in $E_{i-1}$ are no longer readers of $ESG_{in}$ or sources of $ESG_{out}$ (once one of the calls by any existing threads to methods removeReaders and removeSources has completed) and have terminated. Hence, properties P1-P3 hold in any arbitrary $E_i$ as they do in $E_0$. □

7 ALGORITHMIC IMPLEMENTATION OF ESG

STRETCH, similarly to ScaleGate, builds a list where tuples are maintained in timestamp order, along with some auxiliary book-keeping structures. The protocol for adding and accessing tuples is customized for the needs of data streaming operator pipelines. Recall that each source thread adds tuples in timestamp order, while each reader traverses the sorted stream of tuples, so that each tuple $t$ will be returned to each invoking reader, once $t$ becomes ready. The algorithmic implementation of all the methods is outlined below.

addTuple, getNextReadyTuple: The algorithmic implementation constructs a skip list, with auxiliary book-keeping structures—essentially acting as thread-specific data for the sources and readers—and fine-grained synchronization to avoid global locking. The book-keeping structures contain handles to the skip list, for sources and readers, to continue inserting or reading nodes (tuples) respectively. As shown in Fig. 2, readers’ handles traverse
the list from head to tail, retrieving the next tuple only if the latter is not pointed by a source’s handle (thus returning only ready tuples). At the same time, sources’ handles point to their last inserted tuples and facilitate the sorted insertion of subsequent tuples (also leveraging the skip list shortcuts). Since each source adds a timestamp-sorted stream, each next insertion “falls” after its previous one (i.e., closer to the tail). All the tuples before (i.e., with earlier timestamps) the earliest tuple pointed by the source are ready.

**announceReaders(List reader_IDs, rID), removeReaders(List reader_IDs):** As mentioned above, a reader has access to one of ESG’s nodes through its own handle. A new reader p to the ESG simply needs a handle to a node that is ready, so that p can safely traverse the rest of the list in timestamp order in subsequent getNextReadyTuple invocations. Since announceReaders is called by an existing reader, the caller’s handle to the most recently read node of such reader is used, so that all the new readers have a handle to the ESG. Removing a reader is as simple as removing the thread-specific structures of that reader.

**announceSources(List source_IDs, min_ts):** A new source to be registered in ESG needs its own related book-keeping structures, i.e., its own handles, which essentially can be copying the handles of an existing source. For the sake of the new thread, an initial dummy tuple with timestamp min_ts is inserted, to initialize the functionality of its handles. Dummy tuples are not returned as ready to readers invoking getNextReadyTuple, but enable other tuples with smaller timestamps to be characterized as ready and be returned to readers. At this point it is useful to recall that the announceSources operation is called for ESG.out after the processing thread has returned from the blocking method requestNewEpoch (Alg. 8) with min_ts = T. ts, thus ensuring that the source calling announceSources on ESG.out is still pointing to a tuple with a timestamp smaller than min_ts, thus guaranteeing the method is always invoked with appropriate timestamp for the respective parameter (cf. also Theorem 6.2). Adding more than one source at a single time is delegated to a single thread that will add a block of new book-keeping structures and dummy tuples (each pointing to the respective new sources).

**removeSources(List source_IDs):** Removing a source consists mainly of adding, on behalf of the source, a specially marked flush tuple in ESG, with timestamp equal to the latest insertion of the source. The effect of such tuple is that it will essentially push the tuples in mainly of adding, on behalf of the source, a specially marked dummy tuple to the ESG. NextReadyTuple can safely traverse the rest of the list in timestamp order in subsequent getNextReadyTuple, but enable other tuples with smaller timestamps to be characterized as ready and be returned to readers. At this point it is useful to recall that the announceSources operation is called for ESG.out after the processing thread has returned from the blocking method requestNewEpoch (Alg. 8) with min_ts = T. ts, thus ensuring that the source calling announceSources on ESG.out is still pointing to a tuple with a timestamp smaller than min_ts, thus guaranteeing the method is always invoked with appropriate timestamp for the respective parameter (cf. also Theorem 6.2). Adding more than one source at a single time is delegated to a single thread that will add a block of new book-keeping structures and dummy tuples (each pointing to the respective new sources).

Concurrent calls of the same method that updates the set of threads (e.g. concurrent calls to announceReaders), or similar calls in the same epoch may happen; synchronization is in place (using a TestAndSet variable) so that only one of each type takes effect. Concurrent calls among competing such methods (e.g. announceReaders and removeReaders) are not supposed to happen, as they both need to modify the thread-specific book-keeping structures (indeed such invocations are not done in STRETCH). If an ESG implementation wants to allow that, it will need to enforce synchronization to protect consistency; since these are low-contention operations, a simple lock can do. If regular operations (to add and get tuples) are concurrent with those that update the set of threads and the respective book-keeping structures, the latter can overwrite, causing the former to have no effect. Note that their use in STRETCH imply that such invocations do not interfere.

**8 MODELLING STRETCH’S PERFORMANCE**

Before evaluating STRETCH, we model in this section the expected scalability behavior of its virtual shared-nothing parallelism (VSN) and that of pure shared-nothing parallelism (PSN).

Let us consider a setup in which three threads t I, tP and tO are defined for T I, T P and T O, respectively. Moreover, let us assume that dI, dP and dO are the per-tuple expected processing times for t I, tP and tO, respectively. If we aim at scaling the analysis of these threads by provisioning more threads to T P, then tP is the bottleneck of the pipeline. That is, dI = max(dI, dP, dO). Otherwise, the threads to be provisioned should be dedicated to T I if dI = max(dI, dP, dO), or T O if dO = max(dI, dP, dO). If we opt for PSN, we need to define a mechanism for t I to route tuples to T P’s threads. The routing overhead will depend on the semantics of the stateful operator (e.g., if key-by-partitioning can be leveraged or if tuples should be broadcast to T P’s threads). A mechanism to merge-sort deterministically tuples at tO is also required. When n threads are defined for T P, we refer to dRnPSN, dPnPSN and dMnPSN as the time spent by t I to route tuples, the new per-thread processing time of T P’s threads and the time spent by tO to merge tuples deterministically.

Similarly, we can expect VSN to scale to the highest n so that:

\[ dP_nPSN = \max(dI + dR_nPSN, dP_nPSN, dO + dM_nPSN) \]

Similarly, we can expect VSN to scale to the highest n so that:

\[ dP_nPSN = \max(dI + dC_nPSN, dP_nPSN, dO + dM_nPSN) \]

Hence, VSN allows for better scalability than PSN for all n so that:

\[ \begin{align*}
  &dI + dC_nPSN < dP_nPSN < dI + dR_nPSN \\
  &dO + dM_nPSN < dP_nPSN < dO + dM_nPSN
\end{align*} \]
We first empirically validate the model of § 8, including virtual shared-nothing parallelism \( V_{SN} \) (in \textsc{stretch}) and shared-nothing parallelism \( P_{SN} \) (in Flink), for both synthetic and real-world data, from Twitter. Then, we evaluate \textsc{stretch}'s performance for the ScaleJoin usecase (§ 4.3) and compare it with that of the original implementation [12], focusing on intra-epoch throughput and latency for the maximum sustainable rate and studying the scalability for increasing number of threads. Lastly, we evaluate \textsc{stretch}'s elasticity by provisioning/decommissioning threads, measuring the reconfiguration time and its effect on throughput and latency.

Evaluation setup. Experiments run on a 2.10 GHz Intel(R) Xeon(R) E5-2695 CPU with 2 sockets (each with 18 cores), 72 logical threads with hyper-threading and 64 GB memory. \textsc{stretch} is implemented in Java and tested with Java HotSpot(TM) 64-Bit Server VM with default garbage collection settings. For \( P_{SN} \), we use Flink 1.6.0.

All experiments show the average end-to-end latency and throughput over five distinct runs. The former is the timestamp difference of each output tuple and the latest input tuple that contributes to it. To compute the latter, we let input streams inject tuples at full speed while using flow control to handle backpressure. The implemented flow control mechanism is similar to that of Flink, where the rate of a stream is adjusted by an intermediate bounded queue.

9.1 \( V_{SN} \) vs \( P_{SN} \) scalability - synthetic dataset

To evaluate our model (§ 8), we used two synthetic queries in Flink. In our implementation, \( d^I \) is approximately 0.7 msec (i.e. it takes 0.7 msec for \( t^I \) to create and forward an input tuple). Following the costs presented in § 8, the values of all parameters are then calculated for different number of threads for \( T^P \). Flink dedicates one thread per source, per parallel instance of an operator, and per sink. To match the number of threads used by Flink to that of \textsc{stretch}, we define one source for each \( t^I \) thread, we set the operator parallelism to the number of \( T^P \) threads, and define one sink for each \( T^O \) thread. We use Flink’s broadcast primitive to share tuples between the sources and the parallel operator instances.

Figure 4a shows the evaluation results. To compare the observed behavior with that of the model, the figure also includes the expected scalability based on the model (i.e. as in Fig. 3). As shown, the throughput figure of \( P_{SN} \) matches the corresponding one from the model. The behaviour of \( V_{SN} \) also matches that of the model (despite the larger deviation between the observed and expected behavior due to the higher overheads in the system). For both \( V_{SN} \) and \( P_{SN} \), the rate of \( T^I \) threads grows to match that of \( T^I \) while the latency decreases due to the smaller delay for tuples to be processed. \( V_{SN} \) achieves higher throughput and lower latency than \( P_{SN} \), as also expected based on the model. Once \( T^O \) threads become the bottleneck of the pipeline, further increasing the number of threads for \( T^P \) results in a growing latency. In this case, the buffers used to handle the backpressure by Flink (not discussed in our model) stabilize the latency by controlling the input rate.

9.2 \( V_{SN} \) vs \( P_{SN} \) scalability - Twitter dataset

In this case, we study the maximum throughput for a deterministic operator with two logical streams. \( T^P \) threads pass tuples downstream without performing any processing. By removing the processing cost, the maximum throughput is then given either by the speed of \( T^I \)’s threads or the sorting costs of \( T^P \) or \( T^O \) threads. We create two similar queries in Flink (as \( P_{SN} \)) and \textsc{stretch} (as \( V_{SN} \)), and process a dataset consisting of 4.3 million tweets, between the 1st and 2nd of October 2018. Also in this case, our Flink implementation is defined so that the number of threads used to forward and process tuples matches \textsc{stretch}'s one. We use two threads for \( T^I \) to read and forward input tuples. In \textsc{stretch}, since \( T^P \) and \( T^O \) threads receive tuples from ESGs in timestamp order, there is no need for sorting. However, in Flink, \( T^P \) and \( T^O \) threads are responsible for sorting the tuples to support determinism.

Figure 4b shows the operators’ scalability. For an increasing number of \( T^P \) threads, \textsc{stretch} faces a slight decrease in throughput due to the synchronization overheads induced by the higher number of threads. \( P_{SN} \) in Flink starts from a lower throughput and decreases faster for an increasing number of threads. Moreover, the average latency in \( P_{SN} \) in Flink, regardless of the number of \( T^P \) threads, is higher than 200 msec, while \( V_{SN} \)’s in \textsc{stretch}’s is always less than 20 msec.
We continue our evaluation focusing on a specific operator with two logical streams: ScaleJoin [12]. The performance of this parallel operator can be hampered in existing SPEs because of the need for broadcasting tuples to all \( T^P \) threads. Because of this, and since Flink only supports parallel EquiJoin, we compare our results in this case with those achieved by the ScaleJoin implementation.

We follow the same benchmark used in [12, 25] to join two logical streams of \( R \) and \( S \) tuples. \( R \) tuples carry attributes \( \langle t_s, x, y \rangle \), where \( x \) is type of \( \text{int} \) and \( y \) is \( \text{float} \) while \( S \) tuples carry attributes \( \langle t_S, a, b, c, d \rangle \) where \( a, b, c, \) and \( d \) are types of \( \text{int}, \text{float}, \text{double} \), and \text{boolean}, respectively. For each pair of tuples \( t_R \) and \( t_S \), an output tuple with schema \( \langle t_S, x, y, a, b, c, d \rangle \) is produced if:

\[
t_S.a - 10 \leq t_R.x \leq t_S.a + 10 \text{ and } t_S.b - 10 \leq t_R.y \leq t_S.b + 10
\]

Attributes \( x, y, a \) and \( b \) are randomly selected from a uniform distribution with interval \([1, \ldots, 10000]\) which results on average in an output tuple every 250000 comparisons.

### Intra-epoch performance
To assess the intra-epoch performance of \( \text{STRETCH} \), we first check its scalability regarding the number of \( T^P \) threads. Also, we show the performance of a single thread, processing the tuples sequentially. In the single thread implementation, we use one thread per \( T^I \), \( T^P \), and \( T^O \) sets. There is one bounded buffer between the threads of \( T^I \) and \( T^P \) and one buffer between the ones of \( T^P \) and \( T^O \), to control the rate. Hence, the single thread implementation is similar to the ScaleJoin and \( \text{STRETCH} \) with one thread for \( T^P \), where the ScaleGate is substituted with bounded buffers. Figure 5 shows (i) the maximum sustainable input rate averaged over 5 different runs (and the standard deviation) for increasing number of \( T^P \) threads, (ii) the corresponding throughput, in terms of number of comparisons and (iii) the corresponding latency in logarithmic scale, for a time-based window of size 5 min.

As shown in the intra-epoch column of the figure, the throughput of ScaleJoin and \( \text{STRETCH} \) with one thread for \( T^P \) is similar to the single thread but the latency for the single thread is lower than the other two, which is the cost of using ScaleGate. However, as expected (and as discussed in detailed in [12]), by increasing the number of \( T^P \) threads, the throughput for ScaleJoin and \( \text{STRETCH} \) grows linearly. Although hyper-threading (after 36 physical threads) causes a degradation, by keeping increasing the number of \( T^P \) threads, both \( \text{STRETCH} \) and ScaleJoin are still capable of scaling. The latency for the highest throughput achieved by different numbers of threads for \( T^P \) shows that the \( \text{STRETCH} \) achieves latency in the same order as that of ScaleJoin.

### Inter-epoch performance
As discussed in §3, \( \text{STRETCH} \) provides a general API that allows any external policy to take decisions about when to provision or decommission a certain number of threads (based on the statistics provided by the \( \text{STRETCH} \) framework itself or other external information). In these experiments, similarly as done in [10], we trigger the provisioning or decommissioning of threads based on the processing capacity of the threads. More concretely, we define an upper, a target and a bottom processing capacity threshold. When the current processing load of active threads exceeds the upper threshold, the smallest amount of new threads needed to bring the average processing capacity below the target threshold is provisioned. When the current processing load of active threads is below the bottom threshold, the smallest amount of underutilized threads needed to bring the average processing capacity below the target threshold is decommissioned. In our experiments, the upper, target and bottom thresholds are set at 90%, 70%, and 45% of the threads maximum throughput, respectively.

To evaluate the elasticity of the framework, we increase (decrease) the load after filling the window and add (remove) threads while measuring the latency, throughput and reconfiguration time. For the provisioning experiments, we start with input rate 70% of the maximum rate that can be sustained by the corresponding number of \( T^P \) threads for time-based window with window size 5 minutes. After 6 minutes, when the window is full and the system is stable, we increase the rate to 120% of the maximum sustainable rate which requires provisioning of new threads for \( T^P \) in order to keep up with the input rate, and therefore trigger an epoch switch. For the decommissioning experiments, we start with 70% of the maximum sustainable rate and then after 6 minutes, similarly as the previous experiment, decrease the rate to 30%. In this case, the system needs decommissioning a few number of active \( T^P \) threads in order to utilize resource usages. Table 2 presents the number of threads that need to be provisioned or decommissioned depending on the number of \( T^P \) threads running in the system.

<table>
<thead>
<tr>
<th>( T^P )</th>
<th>5</th>
<th>9</th>
<th>18</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>starting ( T^P )</td>
<td>9</td>
<td>16</td>
<td>31</td>
<td>52</td>
<td>69</td>
</tr>
<tr>
<td>( T^P ) after Provisi</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>12</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 2: Provisioned / decommissioned threads depending on the number of \( T^P \) threads.
throughput without affecting the latency. In the decommissioning procedure, when decreasing the workload, not only STRETCH achieves the same throughput as ScaleJoin, as expected, but it also shows slightly lower latency.

Moreover, we measure the reconfiguration time, which starts from the moment STRETCH receives a reconfiguration command till it successfully finishes executing it. As shown at the rightmost column in Fig. 5, the reconfiguration time is always less than 40 msec, which indicates there is no significant degradation during the epoch switch. Furthermore, at the same column, we show the load imbalance, in terms of coefficient of variation percentages. As it can be observed, in most cases there is at most 1% difference, while in all cases the difference is at most 2%.

### Summary of the evaluation results

The empirical study (i) validates the model in § 8 (Fig. 4 and 3), (ii) demonstrates very fast reconfiguration possibilities, of just a few msec (Fig. 5 inter-epoch parts), enabled by STRETCH, while it also (iii) shows the low-overhead induced for in achieving these (Fig. 5 intra-epoch column), while preserving determinism in STRETCH.

### 10 RELATED WORK

Several scalable and elastic parallel approaches have been discussed in the literature, e.g. [10, 22, 24]. For a systematic review, we refer the reader to [14]. This work does not focus on a particular strategy or a specific operator, but rather provides a general framework with the goal of promoting virtual shared-nothing parallelism (also showing it can embrace existing parallelism schemes), which, to the best of our knowledge, has not been studied before. The following is a summary of the state of the art in elasticity for intra-operator parallelization, including relation to our results. We organize the discussion in terms of key goals and properties, i.e. number of threads that can change per reconfiguration, the roles of state transfer and of the triggering mechanism, the focus on determinism and the reaction time vs overhead of frequent reconfigurations.

Regarding changes in number of threads, the literature provides methods for provisioning and decommissioning one thread at a time (e.g., [26], [19]) or more threads (e.g., [17]), as in our case. Differently from us, nonetheless, [17] does not actively target determinism.

Regarding state-transfers, one issue, orthogonal to our work, is related to load balancing, a combinatorial problem, related to packing (cf. [3, 10, 16, 17] and references therein). Another issue is that of the cost of transferring, since the overheads of state serialization and deserialization can degrade the SPE’s performance. This can be alleviated by techniques aiming at reducing latency spikes, such as the ones in [15], or at recreating small states at the downstream thread by sending to the latter previously sent tuples (rather than transferring the upstream’s thread’s state), or by distributing the work to nodes through hashing, in ways that minimize the changes when rehashing [8]. Our work enables possibilities for efficiency due to sharing and advances the front of scaling-up by not requiring any state transfer, thus making these issues orthogonal and existing methods complementary to ours.

Targeting determinism needs appropriate synchronization. Determinism has been formalized in the context of parallel SPEs and in the context of algorithms for parallel data streaming operators, for instance by [11, 12] and is also referred to as safety by [18] and semantic transparency by [10]. Here we show sufficient conditions for determinism, also under reconfigurations.

Another key issue is when to trigger reconfigurations, as it is related to trade-offs between overheads and time to react when reconfigurations are needed. In the literature there exist proactive and reactive approaches, load-based approaches and application-performance-related approaches (cf. [6, 10, 19, 21, 23, 32] and references therein). Various triggering mechanisms can be combined...
We thanks our shepherd, Paris Carbone, and the anonymous reviewers for their constructive comments and suggestions. The work was supported by the SSF proj. “FiC” nr. GMT14-0032, by the Chalmers Energy AoA framework proj. INDEED and STAMINA and by the Swedish Research Council proj. “HARE” nr. 2016-03800.

REFERENCES


