Haren: A Framework for Ad-Hoc Thread Scheduling Policies for Data Streaming Applications

Dimitris Palyvos-Giannas  
Chalmers University of Technology  
Gothenburg, Sweden  
palyvos@chalmers.se

Vincenzo Gulisano  
Chalmers University of Technology  
Gothenburg, Sweden  
vinmas@chalmers.se

Marina Papatriantafilou  
Chalmers University of Technology  
Gothenburg, Sweden  
ptrianta@chalmers.se

ABSTRACT
In modern Stream Processing Engines (SPEs), numerous diverse applications, which can differ in aspects such as cost, criticality or latency sensitivity, can co-exist in the same computing node. When these differences need to be considered to control the performance of each application, custom scheduling of operators to threads is of key importance (e.g., when a smart vehicle needs to ensure that safety-critical applications always have access to computational power, while other applications are given lower, variable priorities).

Many solutions have been proposed regarding schedulers that allocate threads to operators to optimize specific metrics (e.g., latency) but there is still lack of a tool that allows arbitrarily complex scheduling strategies to be seamlessly plugged on top of an SPE. We propose Haren to fill this gap. More specifically, we (1) formalize the thread scheduling problem in stream processing in a general way, allowing to define ad-hoc scheduling policies, (2) identify the bottlenecks and the opportunities of scheduling in stream processing, (3) distill a compact interface to connect Haren with SPEs, enabling rapid testing of various scheduling policies, (4) illustrate the usability of the framework by integrating it into an actual SPE and (5) provide a thorough evaluation. As we show, Haren makes it possible to adapt the use of computational resources over time to meet the goals of a variety of scheduling policies.

CCS CONCEPTS
• Information systems → Online analytical processing engines; • Software and its engineering → Scheduling; Middleware.

KEYWORDS
Stream processing, Scheduling, Middleware

1 INTRODUCTION
Data streaming is leveraged in applications dealing with heterogeneous data sources, variable input rates (and data distributions) as well as heterogeneous hardware (ranging from high-end servers to embedded edge devices). Stream Processing Engines (SPEs), the platforms running streaming queries (or simply queries), deploy the latter's operators to multiple SPE instances (i.e., processes) existing within or across multiple computational nodes. In this context, resource scheduling [3, 13, 27–29] chooses how and to which SPE instances to deploy queries' operators while thread scheduling, our focus, chooses how to allocate an SPE instance's threads to the operators deployed to it to meet specific performance goals.

Many related works have shown that custom thread scheduling (or simply scheduling) can reach better performance (e.g., lower processing latency) than that achieved when SPEs instantiate per-operator threads [7, 12, 24] which are scheduled by the Operating System (OS) [9]. Existing solutions discuss nonetheless specific scheduling policies (in combination with certain SPEs), without considering how to express the scheduling goals of a policy without the need to code its logic within an SPE.

This observation forms the basis of our work, which aims at identifying and generalizing the logic of a general scheduler that can encapsulate existing policies while decoupling its internals from those of an SPE. Thus, our research question is the following: Is it possible to define an all-purpose SPE scheduling framework, which (i) allows the user to easily plug-in custom scheduling policies, (ii) transparently enforces those policies at runtime and (iii) requires minimal programming effort? These requirements can be crucial, especially in large cyber-physical systems (such as smart grids and vehicular networks) in which users and analysts can continuously deploy applications of different criticality, priority or latency sensitivity [19, 23] and SPEs themselves can perform adaptive live reconfigurations (e.g., operator fusion and fission [11]) to adjust resources to query loads and costs. We provide an affirmative answer and present Haren, a general tool which can be used in combination with an SPE with minimal modifications. We evaluate it in combination with Liebre, a lightweight SPE for edge-computing [14]. In summary, we make the following contributions:

• We distill a compact set of primitives that can encapsulate the logic of the most common scheduling policies proposed in the literature, allowing users to define scheduling semantics without the need for altering the internals of the SPE.
• Together with these primitives, we define the facilities that the SPE needs to provide for custom scheduling to happen.
• We design and implement a framework that leverages such primitives in a lightweight fashion without dedicated threads but by sharing the job among threads running the analysis.
We perform a thorough evaluation for different scheduling policies (of different complexity) leveraging hardware that can be employed at the edge of large cyber-physical systems, where custom scheduling policies are needed the most [19]. As we show, Haren allows the user to define rich scheduling policies (even multiple dedicated ones when not all queries in an SPE instance share the same performance goals) and enforces them with minimal overhead, achieving performance goals that are not matched when the SPE relies on the underlying OS scheduler.

Outline: § 2 covers preliminaries about data streaming and scheduling. § 3 presents our goals and system model. § 4 overviews Haren while § 5 and § 6 discuss its internals. § 7 presents our evaluation of Haren. Last, § 8 covers related work and § 9 concludes the paper.

2 PRELIMINARIES

Streams & Operators. A stream is an unbounded sequence of tuples sharing a schema composed by attributes \(a_1, \ldots, a_n\). A query is a DAG of operators connected by streams. External data sources generate tuples to be processed by the operators of the query. These tuples, which are referred to as ingress tuples, are delivered to queries by Ingress operators (also called Sources or Spouts [7, 12, 24]), are pushed through the rest of the operators of the query, possibly resulting in new tuples, and are eventually delivered as egress tuples to Egress operators (also called Sinks [7]), which forward them to users or other applications. Streaming operators define at least one input stream and one output stream. The only exceptions are Ingress, which has no input and a single output stream, and Egress, which has one input but no output streams. The output tuples of an operator that are waiting to be processed by another operator connected to it are maintained in a queue shared between the two.

Clock time attribute. We assume that, apart from the user-defined attributes, all tuples carry a clock time \(ta\) attribute\(^1\). This attribute carries the clock time at the moment in which the tuple is forwarded by the Data Source producing it\(^2\). If a tuple \(t\) is created by an operator of the query, its clock time is set to the respective value of the latest input tuple triggering the creation of \(t\) at the operator. By extension, each tuple \(t\) that is not an ingress tuple carries the clock time of the latest ingress tuple triggering its creation.

Sample Query. Figure 1 presents a sample query composed by two operators (plus one Ingress and one Egress). For each input tuple, operator \(op_1\) creates an output tuple carrying the same \(ta\) attribute of the input tuple plus an attribute \(c\) for the sum of \(a\) and \(b\). Operator \(op_2\) produces tuples carrying, for each fixed window [2] of 10 minutes, the attribute \(ta\) of the latest tuple contributing to the window, the attribute \(d\) containing the maximum value of \(c\) observed in the window and the attribute \(w\), to specify the start time of the window. The figure also shows the tuples currently present in each queue. Since attribute \(ta\) is set for each output tuple created by an operator to the value of the latest input tuple contributing to such output tuple, it can be observed that all the tuples in a queue of a particular operator have \(ta\) values that are smaller than or equal to those of the latter’s input queues. We use in the remainder the terms upstream and downstream peers to refer to the operators preceding and following an operator, respectively (e.g., \(op_1\) is the upstream peer of \(op_2\), while \(op_2\) is the downstream peer of \(op_1\)).

Scheduling. An SPE instance running a set of operators (from one or more queries) has access to one or more CPU cores (mapped to hardware threads). In our work, scheduling refers to the process of periodically deciding which operators (possibly of different queries) should be run and in which order, within an SPE instance. A scheduled operator runs its code inside a hardware thread. At any moment, an operator can be run by at most one thread.

A challenge in scheduling streaming operators is that, because of the varying rates and data distribution of data sources (which in turn affect the rates, data distribution and behavior of the queries’ operators), scheduling policies cannot be defined statically at compile time, but need to be continuously refined over time.

The goal of custom scheduling for SPE instances is to control the performance characteristics of the queries. We quantify the performance of one or multiple queries as follows. Starting from the operator level, we quantify its performance over a period of time with the following metrics:

1. **Throughput**, the number of tuples processed by the operator.
2. **Latency**, the average clock time elapsed between the operator’s processing of each tuple \(t\) and \(t\)’s clock time (i.e., the clock time of the latest ingress tuple triggering the production of \(t\)).
3. **CPU Utilization**, the average CPU utilization (%) of the operator.
4. **Memory Cost**, the maximum amount of memory consumed by the tuples maintained in the operator’s input queues.

Extending the performance characterization from operators to whole queries, we define (i) the query throughput as the average throughput of the query’s Ingress operators, (ii) the mean and max query latency as the average and maximum latency observed at its Egress operators, respectively, (iii) the CPU utilization as the sum of the CPU utilization of the query’s operators, and (iv) the memory cost as the sum of the memory costs of all the operators of the query. These definitions can be extended to multiple queries by means of the sums, averages, and maximums of all their operators.

It should be noticed that these performance metrics depend on multiple aspects such as (i) the scheduling decisions, (ii) the arrival pattern of the incoming data, as well as (iii) the data distribution of the input values. For example, both the throughput as well as the memory cost depend on the CPU time allocated for a certain operator as well as the rate of the data source(s).

With respect to the example of Figure 1, one can observe that (i) given the tuples currently stored in the operators’ queues and (ii) assuming that the next scheduled operator can process all its shown input tuples, the scheduling choice would depend on the desired performance metric. More concretely, scheduling the Egress operator would minimize the query’s latency, scheduling operator \(op_1\) would minimize the overall memory used while scheduling the Ingress operator would maximize the query’s throughput.

3 GOALS AND SYSTEM MODEL

We aim at designing and implementing a general purpose scheduling framework that allows users to define ad-hoc scheduling policies. More concretely, we want to allocate the threads of an SPE instance in a streaming-application-aware fashion that can meet
G1 Distilling a compact interface for a scheduling middleware that allows the implementation of custom, user-defined rules for features. Tuples, operators, and queries have various features that characterize their behavior and state. A general-purpose scheduling framework must be aware of the changing nature of these features to make informed decisions and orchestrate the execution of queries’ operators according to a user-defined scheduling policy.

G2 Allowing the implementation of custom, user-defined rules for both inter-thread scheduling (i.e., specifying the assignment of operators to threads) and intra-thread scheduling (i.e., specifying the scheduling of operators within each thread).

G3 Distributing and sharing scheduling overheads among available physical threads to take advantage of multi-core nodes.

3.1 System model

Tuples, operators, and queries have various features that characterize their behavior and state. A general-purpose scheduling framework must be aware of the changing nature of these features to make informed decisions and orchestrate the execution of queries’ operators according to a user-defined scheduling policy.

Not all features are equal in terms of how they change and in terms of how their changes can be observed. A first distinction can be made between static features (e.g., the type of an operator) and dynamic features (e.g., the selectivity of an operator, which depends among other things on the data being fed to it). A second distinction can be made between features that are immediately derived from an operator (e.g., its number of input streams) and features that are derived from the input/output queues of an operator and/or the tuples maintained in such queues (e.g., the clock time of the earliest tuple maintained in any of the operator’s input queues). This second distinction is crucial because it results in two critical observations. First, certain features can change over time independently of whether the operator is scheduled or not. This is the case, for instance, for the earliest clock time of any tuple in the input queues of an operator, given that it could change if any of its upstream operators are scheduled. Second, it might not be possible to update particular features of some operators unless these operators are executed (e.g., the average time needed by an operator to process one tuple, also referred to as the cost of an operator). Based on these observations, we introduce the following definitions.

Definition 3.1. A feature $F$ of operator $op_i$ is independent if it can change only upon execution of $op_i$.

Definition 3.2. An operator $op_i$ is feature-dependent on operator $op_j$ for feature $F$ if $F$ for $op_j$ can change upon execution of $op_i$.

Definition 3.3. A feature $F$ of operator $op_i$ is dependent if it can change upon execution of $op_j$ as well as operators on which $op_j$ is feature-dependent.

Definition 3.4. A feature $F$ of operator $op_j$ is execution-intrinsic if it can be updated only upon execution of $op_j$.

Since operators, queues and tuples are accessed by the SPE, we assume the latter provides an interface (such as a metrics system [7, 12]) for Haren to retrieve up-to-date feature values. Although the features used by Haren can be arbitrarily complex, to keep the discussion tractable, we will focus on a specific set of them. These features, presented in Table 1, are required to implement most of the scheduling policies proposed in the literature, including the policies we later use in our evaluation. The table displays features along with their abbreviated id and type. For brevity, we do not include static features that can be trivially computed, such as the type of an operator. As aforementioned, the costs of the average time spent by an operator to process a single input tuple. The selectivity defines the average number of output tuples produced per processed input tuple. Observe that it can be higher than 1 for operators that generate multiple output tuples for every input tuple (e.g., an operator that splits a sentence into words). Cost and selectivity are used in many scheduling policies, to optimize for different metrics.

<table>
<thead>
<tr>
<th>ID</th>
<th>Feature</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>Cost</td>
<td>Dynamic, independent, execution-intrinsic</td>
</tr>
<tr>
<td>$s$</td>
<td>Selectivity</td>
<td>Dynamic (except for Egress operators), independent, execution-intrinsic</td>
</tr>
<tr>
<td>$l_H$</td>
<td>Head clock time</td>
<td>Dynamic, dependent on upstream and downstream peers, execution-intrinsic for Ingress operators</td>
</tr>
</tbody>
</table>

Table 1: Table of features considered in the paper.
such as the average latency or the memory cost of the queries [23]. The head clock time is the earliest clock time of the tuples at the head of the input streams of an operator. This feature is also used in various scheduling policies (e.g., to optimize the maximum latency of a query based on its operators head latency, which is derived from their head clock time [6]).

We assume in the remainder that the SPE instance has \( K \) active CPU cores which correspond to hardware Processing Threads (PTs). We refer to the \( i \)-th processing thread as \( PT_i \mid i \in \{ 1, \ldots , K \} \) and to the \( i \)-th operator of the \( j \)-th query as \( OP_{ij} \) (but omit the query number if it is not essential for the discussion).

4 OVERVIEW

Streaming applications have a live and changing nature, with varying input stream rates and data distributions. In order to correctly enforce a scheduling policy, features and priorities that change over time need to be periodically updated. Since changes in features can depend on scheduling decisions (see § 3), information about scheduling decisions must also be collected over time.

Figure 2 shows the two main tasks executed by Haren’s PTs, namely execution (\( T_E \)) and scheduling (\( T_S \)). PTs run task \( T_E \) during the majority of the time and switch periodically to task \( T_S \). These tasks isolate the portions of time during which scheduling information is gathered for priority updates (i.e., when PTs must synchronize) from those during which PTs can be dedicated to running the operators deployed to the SPE instance. The separate tasks give fine-grained control over the scheduling overhead, which is proportional to the time spent gathering information about scheduled operators and updating features and priorities.

As stated in § 3, we aim at distributing and sharing the scheduling overhead among all PTs (Goal G3). Because of this, Haren parallelizes the costly parts of \( T_S \) and lets all PTs (in a random fashion) take care of the portions of \( T_E \) that can be run more efficiently in a sequential fashion. We overview in the following tasks \( T_E \) and \( T_S \) and refer the reader to § 5 and § 6 for more detailed descriptions.

![Figure 2: Alternation of \( T_E \) (execution task) and \( T_S \) (scheduling task) during the runtime execution of Haren.](image)

**Overview of \( T_E \).** During this task, each PT locally executes the operators that were assigned to it, keeping track of the executed operators, in order to share this information during the following \( T_S \). To make certain that fresh values of the operators’ features are available, PTs also ensure that operators with execution-intrinsic features do not stay unscheduled for an excessive period of time.

**Overview of \( T_S \).** During this task, PTs update the scheduling decisions by sharing information about the operators scheduled during the previous \( T_E \). In § 3, we distinguished features into independent and dependent (Definition 3.1 and Definition 3.3). Although it is easy for a PT to detect if an independent feature of its operators needs to be updated, the same is not true for dependent features, because such features might depend on the actions of multiple PTs. Haren reduces overheads by defining a sequential portion of \( T_S \) in which exactly one PT (chosen randomly) updates all the dependent features that have potentially changed and, subsequently, redistributes the operators to all PTs. Then, each PT, in parallel, computes priorities for its operators and sorts these operators based on the recently updated priorities, concluding \( T_S \).

Note that an operator might be assigned to different PTs in distinct executions of \( T_S \). To prevent situations where two PTs try to execute the same operator at the same time (see § 2), the sequential portion of \( T_S \) also acts as a barrier that marks, for all PTs, the end of the current \( T_E \) and the beginning of the next. For the same reason, no operator is executed during \( T_S \).

4.1 Inter-thread and intra-thread scheduling

Since SPE instances can run on multiple threads, Haren allows users to specify how to (i) assign operators to PTs and (ii) decide the order with which each PT should schedule the operators assigned to it. It does this by means of an inter-thread scheduling function \( f \) and an intra-thread scheduling function \( g \). \( f \) being the set of available features and \( O \) the set of operators deployed to the SPE instance, we define these functions as follows.

The *inter-thread scheduling function* \( f : R^{[F \times |O|]} \rightarrow \{ 1, \ldots , K \} \) identifies which PT should execute which operator, for all the operators deployed to the SPE instance. Note that, when computing which PT should be in charge of executing a certain operator, the features of all the operators deployed to the SPE instance are given as input to \( f \). Thus, \( f \) can be used to implement both simple thread assignment policies (e.g., assign the operators of queries to PTs in a round-robin fashion), as well as much more complex ones (e.g., assign operators so that the load is equal for all the PTs).

The *intra-thread scheduling function* \( g : R^{[F \times |O|]} \rightarrow R^D \) maps the features of operator \( OP_i \) to a \( D \)-dimensional priority vector \( P_i = (p_{i1}, p_{i2}, \ldots , p_{iD}) \). Also in this case, the features of all operators deployed to the SPE instance are input to \( g \) when computing the priority of each operator. Each element of the priority vector reflects a priority dimension. The execution of operators is prioritized based on a lexicographic sorting of their priority vectors. For example, a possible priority vector might describe two dimensions (queryClass, cost) (each computed based on the operators’ features). In this case, operators with higher queryClass would be scheduled before others with lower queryClass, while operators with equal queryClass would be scheduled according to their cost.

4.2 Architecture

Figure 3 shows the APIs coupling an SPE instance with Haren, used by the latter to schedule the operators deployed to the former.

The user interested in running a set of operators belonging to queries \( Q_1, Q_2, \ldots \) with a particular scheduling policy can invoke
the SPE’s deploy function and pass the queries’ operators to be executed. She also initializes Haren with the inter-thread and the intra-thread scheduling functions f and g and the runtime parameters P, b and d (described in the figure). For simplicity and without loss of generality, the figure and our following discussion focus on a single SPE instance. When the queries’ operators are deployed to either one or multiple SPE instances (see § 1), each SPE instance is coupled with one instance of Haren. The SPE instance notifies the associated Haren instance of the new deployment by calling update. Internally, Haren inspects the queries in order to identify the set O of operators to be scheduled (and their interconnections) at the coupled SPE instance. Once Haren identifies the set F of features used by f and g, Haren’s PTs schedule the execution of the operators. This is done by invoking:

- SPE.canRun(i, j), to check whether operator \( op_i \) can be executed (i.e., if it has input tuples and space to place potential results in its output streams’ queues).
- SPE.run(i, j, b), to run \( op_i \), specifying \( b \) as the maximum number of tuples it can process during the function invocation (we refer to § 5 for more details about the role of \( b \) in scheduling).
- SPE.getFeature(i, j, F), to retrieve feature \( F \) for \( op_i \).

The SPE instance can also invoke the function update when, due to runtime reconfigurations (e.g., operator fusion or fission [11]), the list of operators scheduled by Haren changes.

![Figure 3: APIs coupling Haren, the user and the SPE instance.](image)

### 5 EXECUTION TASK \( T_E \)

In this section, we provide a detailed description of the actions performed by PTs during \( T_E \). More concretely, we discuss (i) how each PT chooses the next operator to run, (ii) how it backs-off if there is no operator to be scheduled, and (iii) how it takes care of running operators for which execution-intrinsic features (Definition 3.4) have not been updated for more than the user-defined \( d \) time units.

The different variables accessed by each PT are presented in Table 2 while the main loop is shown in Algorithm 1. List \( A \) contains the operators assigned to each PT at the end of the previous \( T_S \) task. Before the first execution of \( T_S \), all operators are given the same priority and assigned randomly to PTs. This also applies for operators added, removed or changed at runtime due to adaptive reconfigurations triggered by SPE instances (their assignment to threads and priorities are then updated during the first \( T_S \) following the reconfiguration).

Each PT traverses \( A \) until it finds the operator with the highest priority that can run, or until it reaches the end of \( A \) (lines 5-13). Here, we remind the reader that an operator can generally run (i) if

**Table 2: Variables used during \( T_E \).**

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>List with the operators assigned to the PT.</td>
</tr>
<tr>
<td>E</td>
<td>Set that contains the operators that were executed by the PT at least once during the last ( T_E ).</td>
</tr>
<tr>
<td>LU</td>
<td>Array of size</td>
</tr>
</tbody>
</table>
it has tuples in its input queues as well as (ii) free space in its output queues (function `SPE.canRun`, line 9). If such an operator is found, it is executed and allowed to process at most \( b \) tuples, where \( b \) is one of the user-defined parameters (§ 4.2) which we refer to as batch size. Subsequently, the next operator in \( A \) is scheduled only if (i) it has the same priority of the previously run operator (and it can run)\(^4\) and (ii) if the elapsed time is less than the scheduling period \( P \). Intuitively, \( b \) is defined to limit the execution time of a given operator, allowing other operators to be scheduled too. Although Haren does not interrupt operators during the processing of a tuple, it can enforce preemptive scheduling policies, with the batch size \( b \) defining the preemption granularity. Smaller values of \( b \) allow for more frequent preemption of the scheduled operators, at the price of higher context-switching overhead.

If the PT reaches the end of the operator list \( A \) and does not find any operator that can run, it invokes a back-off function to avoid spinning (lines 14-15). PTs sleep using a simple exponential back-off algorithm. More specifically, they start with a very small sleep duration and double it at every invocation. The back-off time never exceeds the remaining duration of \( T_E \) and is reset every time a PT enters this task. Afterward (line 16), the PT checks if the time spent in the loop has surpassed the user-defined scheduling period \( P \) and if so, it enters \( T_S \) (line 17, later described in § 6).

As discussed in § 4, if any execution-intrinsic feature of operator \( \text{op}_j^i \) is used by \( f \) and \( g \), Haren needs to run \( \text{op}_j^i \) if the latter has not been scheduled for more than \( d \) time units in order (i) for its feature to be up-to-date, and (ii) for the scheduling policies to be enforced correctly. Because of this, PTs record the last execution time for each operator they schedule (line 10). Moreover, when task \( T_S \) is completed, each PT checks if there are any operators in \( A \) that have not been scheduled for more than \( d \) time units and runs them if that is the case (lines 19-23). Lastly, observe that when PTs run an operator, they also adjust that operator to their set of executed operators \( E \) (lines 11, 23). This allows PTs to selectively update only features of specific operators during the next \( T_S \) task, based on the ones executed during \( T_E \), as described in the following section.

6 SCHEDULING TASK (\( T_S \))

The purpose of the scheduling task is to produce, for each PT, a list \( A \) of operators sorted by their priority vectors \( P_i = (p_{i1}, p_{i2}, \ldots, p_{iD}) \). This list of operators will then be used by each PT during the following \( T_E \) to pick operators for execution.

As mentioned in § 4, Haren tries to minimize the scheduling overhead by parallelizing the costly steps of this task and splitting the work between all PTs. However, as we discussed before (and further elaborate in this section), \( T_S \) also defines a sequential portion executed by exactly one (randomly selected) PT, which we denote as \( t^* \). The sequential portion acts as a logical meeting point for PTs to synchronize their parallel work. In particular, the mechanics of \( T_S \) can be broken down into four main steps:

1. Computing the up-to-date features, done partly in parallel (for independent features) and partly sequentially by \( t^* \) (for dependent features).

\(^4\)The actual implementation does not invoke \( g \) but uses the priority value computed during the previous \( T_S \). In the algorithm \( g \) is used for compact notation.

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global constants</strong></td>
<td></td>
</tr>
<tr>
<td>( F )</td>
<td>Set of all the features used by the user-defined scheduling functions ( f ) and ( g ).</td>
</tr>
<tr>
<td>( F_D )</td>
<td>Set of dependent features.</td>
</tr>
<tr>
<td>( F_C )</td>
<td>Set of constant features.</td>
</tr>
<tr>
<td>( D )</td>
<td>Number of dimensions of the scheduling function ( g ).</td>
</tr>
<tr>
<td>( K )</td>
<td>Number of PTs.</td>
</tr>
<tr>
<td>( PT )</td>
<td>Array of dimension ( K ), of all the available PTs.</td>
</tr>
<tr>
<td><strong>Thread-local variables</strong></td>
<td></td>
</tr>
<tr>
<td>( P )</td>
<td>Matrix of size (</td>
</tr>
<tr>
<td>( D ) Bitmap of size (</td>
<td>O</td>
</tr>
<tr>
<td><strong>Shared variables</strong></td>
<td></td>
</tr>
<tr>
<td>( O )</td>
<td>Array of all the operators deployed to the SPE instance.</td>
</tr>
<tr>
<td>( P )</td>
<td>Matrix of size (</td>
</tr>
<tr>
<td>( U )</td>
<td>Bit array of length (</td>
</tr>
<tr>
<td>( t^* )</td>
<td>The PT that runs the sequential part of ( T_S ).</td>
</tr>
</tbody>
</table>

Table 3: Additional variables used during \( T_S \).
This is because the values of an operator’s independent features might change based on the scheduling decisions of more than one operator will be final (for this $T$ can be certain that any update to an independent feature of any $T$ updates the independent features of each operator in its (local) $PT$s and the second executed sequentially by one $PT$ at each combination whose value can have potentially changed. Updating $Haren$ avoids concurrent attempts to update the value of the same feature for the same operator by multiple $PT$s, since deciding on a correct ordering of these concurrent updates would require the use of a synchronization protocol and thus add overhead to the system (e.g., $SPE.getFeature()$ would need to return the value of the feature and the timestamp of its invocation atomically). At the same time, $PT$s would waste CPU cycles, since all updates to the same position of $F$, except the last one, would be replaced. For these reasons, the second step of updating the features in $Haren$ is done sequentially by $t^*$. This procedure is shown in Algorithm 3, lines 3-5. $t^*$ decides which operators need to have their dependent features updated, using the shared bit array $U$. This array is initialized during the previous, parallel step, with each PT marking (i) the operators they executed (Algorithm 2, line 3), as well as (ii) the feature-dependent operators of the executed operators (lines 4-5). For (ii), each PT needs to know which operator(s) are feature-dependent on each operator they executed. This knowledge is encoded in bitmap $D$, which is initialized once, at the beginning of the execution, based on the structure of the streaming queries and the dependent features used in the scheduling policy. For this phase to work correctly, all updates to $U$ by $PT$s must have finished and be visible to $t^*$. To ensure this, $PT$s are forced to wait at the $entryBarrier$ (Algorithm 3, line 1) before the update of the dependent features can begin. When all $PT$s arrive at that barrier, they are allowed to pass it, and immediately afterward, all $PT$s, except $t^*$, are blocked at the $exitBarrier$ (line 11). The $PT$s will wait there until the dependent features have been updated by $t^*$. The need for the second barrier is explained in the next paragraph. Since the only requirement for choosing a $PT$ to become $t^*$ is that there can be only one at every $T_S$, the $PT$ is chosen arbitrarily: the last $PT$ that leaves the $entryBarrier$ is appointed to be $t^*$ (line 2).

**Assigning operators to $PT$s.** After the features have been updated, $t^*$ assigns operators to $PT$s, using the inter-thread scheduling function $f$, as seen in Algorithm 3 lines 6-8. Care is needed in this step so that the updated mapping of operators to $PT$s takes effect at the same time for all $PT$s. This ensures that $Haren$ avoids situations where, for example, the same operator (which could be mapped to two distinct $PT$s in two distinct $T_S$s) is executed concurrently by two different $PT$s. Such situations are avoided having all $PT$s except one block at $exitBarrier$ (introduced above). Only when $t^*$ has finished its work (and has called $await()$ at the barrier), can all $PT$s move forward to update the priorities. If a reconfiguration was triggered by the $SPE$ during the previous $T_S$, $Haren$’s data structures will be resized during this step, and any new operators will be randomly assigned to $PT$s. After the operator assignment, the two final parts of $T_S$, namely the calculation of the new priority vectors and the sorting of operators, are done in parallel by all $PT$s.

**Priority Update.** As discussed in § 4.1, the intra-thread scheduling function $g$ can use features of any deployed operator to compute the priority vector $P_i = \langle P_{i1}, P_{i2}, \ldots, P_{iD} \rangle$ of operator $op_i$. To maintain simplicity without sacrificing performance, $Haren$ performs the feature and priority updates separately, but executes them both in parallel in all $PT$s. To do the priority update, each $PT$ applies the intra-thread scheduling function $g$ to all operators in its operator list $A$ (Algorithm 2, lines 7-8). This process begins immediately after the assignment of operators to $PT$s. The resulting priority vectors are stored in a thread-local $|O| \times D$ matrix denoted by $P$.  

---

**Algorithm 2: Haren.update( ) -- $T_S$ (Parallel Steps)**

// Update independent features
1
2 for $op_i \in \mathbb{E}$ do
3 $F_i \leftarrow SPE.getFeatures(i, j, F_D \cup P_C)$
4 $U_i \leftarrow 1$
5 for $op_i \in \mathbb{O} \mid D_{i,k} = 1$ do
6 $U_k \leftarrow 1$
7 $Haren.coordinate()$ // Algorithm 3
8 // Compute priorities
9 for $op_i \in \mathbb{A}$ do
10 $P_i \leftarrow g(i, j, P)$
11 // Sort based on priorities
12 $sortOperators(A, P)$

---

**Algorithm 3: Haren.coordinate( ) -- $T_S$ (Coordination Step)**

1 $entryBarrier.await()$
2 if last then
3 $// Start sequential (only $t^*$ enters)$
4 $// Update dependent features$
5 for $op_i \in \mathbb{O} \cup U_i = 1$ do
6 $F_i \leftarrow SPE.getFeatures(i, j, F_D)$
7 $U_i \leftarrow 0$
8 $// Inter-thread scheduling$
9 for $op_i \in \mathbb{O}$ do
10 $t \leftarrow f(i, j, P)$
11 $A_t.append(op)$
12 $// End sequential$
13 $exitBarrier.await()$
14 else
15 $exitBarrier.await()$
We evaluate \( O \) without the need to synchronize with the other PTs. The source data is artificially generated. Each chain has \( \mathbb{C} \) signed a cost class \( \mathbb{E} \) expressed the number of egress tuples produced for every ingress tuple, which once again is done in parallel by all PTs (Algorithm 2, line 9). The operators in \( \mathbb{A} \) are lexicographically sorted according to the values of their priority vectors. More precisely, an operator \( \mathbb{p} \) is considered to have higher priority than \( \mathbb{p} \) if
\[
(\forall k < l : p_{mk} = p_{nk}) \land (p_{ml} > p_{nl})
\]
After the sorting is complete, each PT can immediately enter \( T_E \) without the need to synchronize with the other PTs.

7 EVALUATION

We evaluate \( \mathbb{H} \) by integrating it with a real-world SPE, implementing several scheduling policies of different complexities and studying their behavior and performance. We utilize small, low-end devices usually found at the edge of cyber-physical systems. We chose them because, while \( \mathbb{H} \) can provide custom scheduling facilities to SPEs running in any kind of computing node, scheduling decisions can have a higher impact on performance when processing resources are limited. We first describe the experimental setup, then cover the various scheduling policies we use and present results for different complexities of the latter.

7.1 Experiments setup

Hardware/software. We use Odroid-XU4 [17] devices (or simply Odroid) with Samsung Exynos5422 Cortex-A15 2Ghz and Cortex-A7 Octa core CPUs and 2 GB of RAM, running Ubuntu 18.04.2 LTS and Java HotSpot(TM) Client VM 1.8.0_201-b09. \( \mathbb{H} \)'s PTs run on the four big cores (i.e., \( K = 4 \)). CPU consumption is measured with ps and memory usage is retrieved from the JVM Runtime.

\( \mathbb{H} \) implementation. We evaluate a fully-featured version of \( \mathbb{H} \), implemented in Java, and integrated with Liebre, a lightweight SPE for edge-computing [14]. The integration builds on \( \mathbb{H} \)'s API (§ 4) with few changes in the SPE’s implementation.

Queries. We evaluate \( \mathbb{H} \) using synthetic queries, each consisting of a chain of operators with custom cost and selectivity (see Table 1). The source data is artificially generated. Each chain has one Ingress operator that retrieves data from a Data Source, which runs independently of the SPE. All chains have the same length \( L \). The selectivity and cost values of operators are chosen using a strategy inspired by [23]. More specifically, selectivity and cost are chosen at two levels: query-level and operator-level. Regarding selectivity, each query \( j \) is assigned a selectivity value \( s_j \), which expresses the number of egress tuples produced for every ingress tuple, chosen uniformly at random from \([0.01, 1]\). Then, to satisfy the query selectivity, each operator of the query gets a selectivity equal to \( e^{\log_{10} s_j / L} \approx 10\% \). For the cost selection, each query \( j \) is assigned a cost class \( z \in [0, 4] \) and then the query’s cost is computed as \( c_j = B \times 2^z \). The cost of a query is proportional to the minimum time required for an ingress tuple to be processed by all query’s operators. The cost of the operators is then set to \( c_j \leq 10\% \). \( B \) is the base cost parameter that allows us to vary the load and thus the utilization of the system. We use operator chains in our evaluation to stress-test \( \mathbb{H} \) in a tractable manner by simulating an SPE with a heterogeneous load. However, it should be noted that \( \mathbb{H} \)'s model can handle complex query graphs that contain forks or joins, without any alteration. The correct handling of such cases depends only on the implementation of scheduling functions \( f \) and \( g \).

7.2 Scheduling Policies

As described in § 1, \( \mathbb{H} \) is a general scheduling framework that can implement most scheduling policies defined in the literature. To give evidence of this, we evaluate three scheduling policies from the literature, each of which optimizes a different performance metric. We also study a custom policy that we define in § 7.4. Apart from these policies, we also evaluate the performance when the SPE runs without \( \mathbb{H} \), executing each operator in a dedicated thread instead. An overview of the features, dimensions and goals of each policy is given in the following (discussing how they define function \( g \)) and also in Table 4. In all experiments, the inter-thread scheduling function \( f \) randomly distributes operators to all PTs. The queries are chains of operators with \( L = 12 \). The scheduling period \( P \) is 100ms and the batch size \( b \) is 10 tuples. Each experiment runs for at least five minutes and is repeated at least five times.

Dedicated Threads (OS), the baseline policy, is the default for many SPEs. \( \mathbb{H} \) is not used and, instead, each operator runs in a dedicated thread. Threads are thus scheduled by the OS. Since the OS is agnostic to specific streaming-related metrics, the metric this policy optimizes depends on the OS scheduler.

First-Come-First-Serve (FCFS) has been shown to optimize the maximum latency of the queries [6]. Our implementation uses the inverse of the head clock time \( \mathbb{I} \) (defined in § 3) as the operator priority. To minimize the maximum latency of the system, operators with higher head latency (earlier \( \mathbb{I} \)) are given higher priorities.

Highest Rate (HR), presented in [21], aims at minimizing the average latency of the queries running in the system. The priority value of each operator is equal to its global output rate, which represents the number of egress tuples that would be produced per time unit if that operator and all its downstream operators were executed. This policy prioritizes operators that are more productive (higher selectivity) and less costly (lower cost). Since the priorities depend on the features of multiple operators, we expect this policy’s overhead to be higher than that of FCFS.

Chain policy [5] tries to minimize runtime memory usage. It groups operators based on how many tuples they discard and how quickly they do so and prioritizes operators that belong to the

<table>
<thead>
<tr>
<th>Policy</th>
<th>Features</th>
<th># Dims</th>
<th>Optimizes</th>
<th>Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS</td>
<td>( \mathbb{I} )</td>
<td>1</td>
<td>Max Latency</td>
<td>§ 7.3, § 7.4</td>
</tr>
<tr>
<td>HR</td>
<td>( c, s )</td>
<td>1</td>
<td>Mean Latency</td>
<td>§ 7.3, § 7.4</td>
</tr>
<tr>
<td>Chain</td>
<td>( c, s, \mathbb{I} )</td>
<td>2</td>
<td>Memory</td>
<td>§ 7.3</td>
</tr>
<tr>
<td>Multi-Class</td>
<td>( c, s, \mathbb{I} )</td>
<td>2</td>
<td>Custom</td>
<td>§ 7.4</td>
</tr>
</tbody>
</table>

Table 4: Scheduling policies studied in the evaluation.
groups that discard the most tuples for the least cost. If two operators have equal value of priority returned by the chain algorithm, the operator with the earliest head clock time is executed. Thus, the used intra-thread scheduling function $g$ has two dimensions: the priority of the chain algorithm and the head clock time.

*Multi-Class* is a combination of multiple of the previous policies, which are applied depending on the priority class of each query. It is described in detail and studied in § 7.4.

### 7.3 Single-Class Scheduling

In this first part of the evaluation, we study the behavior of intra-thread scheduling functions $g$ that assume that all the queries belong to the same priority class and only prioritize operators based on the value calculated by each specific policy.

*Performance Comparison.* Figure 4 compares the performance of scheduling using dedicated threads or custom scheduling with the FCFS, HR and Chain policies. We evaluate the mean throughput at the Ingress operators, the mean and maximum latency at the Egress operators and the total number of queued tuples. We also evaluate the maximum memory consumption and the average CPU utilization of the SPE process, including the scheduling overheads. The comparison is made for 5, 10, 15 and 20 queries running in parallel. When the processing load is much lower than the maximum capacity of the system (5 queries), OS scheduling can be optimal in throughput and latency, since there is no contention for resources. In such cases, the OS scheduler can respond faster than Haren’s PTs which use an exponential back-off to conserve resources (Algorithm 1). However, OS scheduling’s advantage diminishes as utilization and resource contention increase (>5 queries). For throughput, the Chain policy always performs better, which is expected since it prioritizes operators closer to the Ingress operators. Moreover, HR and FCFS policies optimize for mean and maximum latency respectively, as expected, outperforming OS scheduling. The Chain policy meets its goal of minimizing the total number of tuples in operator queues. Although FCFS results in more queued tuples, its memory consumption is usually lower or equal to Chain. We believe this is because different scheduling strategies result in different behaviors of the garbage collector. The CPU utilization is almost always lower for Haren than for OS scheduling.

*Scheduling Overhead.* Figure 5 shows a breakdown of the overheads introduced by the scheduling task $T_S$. More specifically, it shows the percentage of time spent (i) calculating priorities using the intra-thread scheduling function $g$ (Priority), (ii) sorting the operators based on their priorities (Sort), (iii) updating the independent features and marking operators that need dependent feature updates (Update), (iv) running coordinate (Algorithm 3) (Coord).
and (v) the total time spent in $T_S$ (Total). As shown, the total scheduling overhead remains very low, almost always less than 2%. The overhead of computing operator priorities is negligible for FCFS since its intra-thread scheduling function $g$ is simply the inverse value of one feature of a single operator. That overhead is higher for HR and Chain since they compute costly functions $g$ involving the features of many operators. The time to sort operators by priority and update the independent features is negligible (lower than 1%).

The highest overhead of scheduling in most experiments is the duration of Algorithm 3 (Coord). In that phase, $t^*$ runs the sequential part of $T_S$ (Algorithm 3, L3-8), while all other PTs block. Figure 6 shows a breakdown of the sequential part, illustrating (i) the percentage of time spent updating the dependent features ($Update$), (ii) computing the inter-thread scheduling function $f$ ($Assign$), and (iii) the total percentage of time spent in that part ($Total$). The figure shows that overheads usually increase with the number of queries. The HR policy has only an assignment overhead since it does not use any dependent features. On the other hand, FCFS and Chain also have an update overhead because they use the clock time, a dependent feature which needs updating. In all cases, the total time spent by $t^*$ in the sequential part is less than the total duration of Algorithm 3 (Coord in Figure 5). The time difference is due to the synchronization overhead of the entryBarrier and exitBarrier. This overhead is needed not only to coordinate the PTs entering the different scheduling phases together but also to ensure memory visibility of actions happening before and after the barriers.

Figure 7 shows the duration of Algorithm 3 for different #queries and values of the scheduling period $P$. It depicts executions of the same policy (FCFS); the size of the dots indicates the magnitude of $B$ (100-1600). In all cases, the overhead remains lower than 5%. Also, we observe an inverse relationship between the length of $P$ and the overhead, which is expected since shorter $P$ causes more frequent invocations of $T_S$. Additionally, the overhead increases with the #queries (and consequently, operators), since there is more data to update and synchronize. These results show a trade-off between the freshness of priorities and the overhead imposed by scheduling.

Depending on the scheduling policy, it might be beneficial to pay a higher overhead for more up-to-date priorities, because the gain in performance will counterbalance the loss due to the overhead.

### 7.4 Multi-Class Scheduling

In this section, we focus on a more complex scheduling scenario and (i) study scheduling queries that belong to different priority classes, giving higher priority to the queries of higher classes and (ii) apply different scheduling policies for the queries belonging to each priority class. Scheduling based on priority classes can be important in many use cases of stream processing. For example, in edge and fog cyber-physical systems, there are frequently many streaming queries with different levels of criticality deployed to a single processing node [18, 19]. A smart vehicle, for instance, can be running many different streaming queries. Some of the queries can be very urgent, such as a query that detects obstacles, while others can be less urgent, such as a query that checks if the fuel is running low. Motivated by the use-case above, we construct the following evaluation scenario: each query belongs to a user-defined priority class which is provided to Haren, and represents the criticality of the query. Several synthetic queries are deployed using Haren having one of two possible priority class values, HIGH or LOW.

In our Multi-Class scheduling policy, queries of HIGH priority are always scheduled before LOW priority ones (objective 1). HIGH priority queries are scheduled using the FCFS policy that minimizes the maximum latency (objective 2) while LOW priority queries are scheduled with the HR policy, to minimize the average latency (objective 3). We run 3 HIGH and 10 LOW queries with different loads, comparing the behavior and ability of OS scheduling and Haren to meet the scheduling objectives. The base cost $B$ is 600.

**Scenario 1 (steady state).** In this experiment, there are adequate processing resources, and the SPE is at a steady state. The HIGH

![Figure 8: Multi-Class Scenario 1 (steady state)](image)

![Figure 9: Multi-Class Scenario 2 (dynamic – high load)](image)
and LOW data sources emit at a constant rate of 500 t/s and 1000 t/s respectively. Figure 8 shows the throughput, mean and max latency for the two query classes. Both scheduling techniques match the throughput of the data sources. However, Haren achieves a much lower max and mean latency for the HIGH queries (objective 2), while keeping the mean latency of the LOW queries at similar levels as the OS (objective 3). The overall performance of HIGH queries is higher than that of the LOW queries (objective 1). Since the policy does not optimize for the maximum latency of LOW queries, this metric shows a higher increase.

Scenario 2 (dynamic — high load). In this scenario, the source rate fluctuates and the system is in an overloaded state. The data sources of HIGH queries emit tuples at a rate of 5000 t/s for 5 seconds and then stop emitting for another 15 seconds. The data sources of the LOW queries emit at a constant rate of 1000 t/s, as before. Figure 9 shows the same performance metrics of the HIGH and LOW queries. Similarly to scenario 1, Haren prioritizes HIGH queries compared to the LOW ones, in contrast with OS scheduling (objective 1). More specifically, Haren achieves better throughput than OS for the HIGH queries, while it is slightly worse for the LOW ones. The figure shows that, for OS scheduling, the maximum latency of all queries keeps increasing. On the other hand, Haren dramatically reduces the maximum latency of HIGH queries (-17.4s) and at the same time keeps it at a near-constant level during the whole execution (0.1s), achieving objective 2. Moreover, the mean latency of the LOW priority queries increases but remains stable and at lower values (2.6s) than those achieved for HIGH queries by OS scheduling (3.5s), thus achieving objective 3. The max latency of LOW queries increases faster, which is expected since Haren’s custom policy does not have this scheduling objective. The results highlight that, especially in the presence of resource contention, Haren’s application-level scheduling allows the users to choose which metric (of which queries) they want to prioritize, until the load decreases or more resources become available.

8 RELATED WORK

Scheduling in data streaming can refer to resource scheduling (how to deploy operators, from one or more queries, to SPE instances within and across computational nodes [3, 13, 27–29]) and thread scheduling (how to allocate threads to operators within each SPE instance). These complementary views can be joint to meet performance metrics (e.g., latency) from both a top-down (e.g., to decide which node should run a certain query or operator) and a bottom-up perspective (e.g., to customize CPU threads allocation to operators). Since we focus on thread scheduling, Haren’s approach is orthogonal to resource scheduling (see § 1) and can work in synergy with it. For a given resource allocation, Haren can take care of thread scheduling at each SPE instance (e.g., Flink TaskManager or Storm Worker [7, 24]) and run operators (that would otherwise be run by dedicated task/executor threads) based on the scheduling policies. The features can be retrieved either from the SPE’s API or from secondary monitoring components (e.g., Flink’s metric system).

Haren is mainly orthogonal to existing work, since it does not rely on any hard-coded policy but rather distills the functionality required from a scheduler to implement general, user-defined scheduling policies. We believe ours is the first work proposing and evaluating a concrete implementation of such a scheduler.

Many scheduling policies and metrics proposed in the literature aim at meeting the growing requirements that users have for streaming applications. The First-Come-First-Serve (FCFS) policy was first proposed in [6] to optimize for the maximum latency of streams of continuous requests, in the context of database and web servers, and has been further studied in the context of stream processing [22, 23]. The Rate-Based (RB) policy optimizes for the average latency of a single streaming query and was described in [25]. In [21], Sharaf et al. present an extension of the Rate-Based policy called Highest Rate (HR) that extends the former to multiple queries. Chandramouli et al. [10] introduce a metric called Mace (Maximum cumulative excess) and describe a scheduling framework for the StreamInsight SPE that uses this metric to accurately estimate the latency imposed by the stream processing pipeline. The Chain scheduling policy, described in [5], tries to minimize the runtime memory usage of multiple queries at the same time. It is proven to be near-optimal for many types of single-stream queries and also acceptable for multi-stream queries; it is also extended in [4] to take maximum latency into account. Aurora, a pioneer SPE, provided a detailed description of its scheduling policy [8, 9] based on two schedulers with different functionalities and goals. The first two-level scheduler schedules queries (superboxes) using Round-Robin, whereas operators (boxes) are scheduled with one of three policies that either optimize for average throughput (Min-Cost), average latency (Min-Latency, which is very similar to the Rate-Based policy) or available memory (Min-Memory). The second scheduler of Aurora aims at optimizing the QoS of the system by utilizing user-provided graphs that correlate the latency with the QoS of queries. The work explores various optimizations to minimize the scheduling overhead while matching user-defined goals.

When many heterogeneous queries run in the same node, it can be crucial to achieve fairness, i.e., balance the degree of slowdown experienced by co-scheduled queries. One way to express this notion is the slowdown or stretch [1, 16] metric. The Longest Stretch First (LSF) metric has been shown to optimize the maximum slowdown [1]. Sharaf et al. propose operator scheduling policies to optimize for latency or slowdown or to balance both of these metrics, either in the average or in the worst case [22, 23]. Heterogeneous queries deployed in the same system can exhibit different QoS requirements. Scheduling queries based on different priority classes is explored in [15], with the Continuous Query Class (CQC) scheduler, a two-level scheduler relying on Weighted Round Robin and Highest Rate schedulers [23]. CQC aims to minimize the latency of high-priority queries and maintain reasonable latency values for the low-priority ones. Pham et al. extend this work and explore the relationship between scheduling and load management in [19, 20]. Their scheduler and load manager work in synergy, exchanging runtime information to consistently honor the user-defined priorities of the queries while increasing the system’s utilization.

9 CONCLUSIONS AND FUTURE WORK

We study the problem of thread scheduling in stream processing, searching for a solution that is not bound to a specific SPE implementation nor scheduling policy. As a result, we propose Haren, an
all-purpose scheduling framework that can be integrated into an SPE through a well-defined API and that allows users to define ad-hoc scheduling policies with minimal programming effort. Haren implements such policies efficiently by parallelizing the work to multiple processing threads in a transparent fashion. We thoroughly evaluate Haren and observe that its expressiveness and efficiency not only allow to define many of the scheduling policies in the literature but also outperform widely-adopted approaches in which SPEs rely on the Operating System scheduler.

Interesting future work studies include the possibilities given by the inter-thread deployment function of Haren to elastically adjust threads of an SPE and boost Haren’s adaptivity by means of autonomous adjustments of its configuration parameters (e.g., the scheduling period $P$). Other interesting directions include a more in-depth exploration of Haren’s behavior for complex queries that liebrevolve parallel branches [26] as well as for runtime changes of the queries or the policies used to schedule their operators.

ACKNOWLEDGMENTS

We thank the shepherd, Ruben Mayer, and the anonymous reviewers for their insightful comments and suggestions. The work was supported by the Swedish Foundation for Strategic Research, proj. "FiC" grant nr. GMT14-0032, by the Chalmers Energy AoA framework proj. INDEED and STAMINA and by the Swedish Research Council (Vetenskapsrådet) proj. “HARE” grant nr. 2016-03800.

REFERENCES