Elastic Scaling for Data Stream Processing

Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu

Abstract—This article addresses the profitability problem associated with auto-parallelization of general-purpose distributed data stream processing applications. Auto-parallelization involves locating regions in the application's data flow graph that can be replicated at run-time to apply data partitioning, in order to achieve scale. In order to make auto-parallelization effective in practice, the profitability question needs to be answered: *How many parallel channels provide the best throughput*? The answer to this question changes depending on the workload dynamics and resource availability at run-time. In this article, we propose an *elastic* auto-parallelization solution that can dynamically adjust the number of channels used to achieve high throughput without unnecessarily wasting resources. Most importantly, our solution can handle *partitioned stateful* operators via run-time state migration, which is fully transparent to the application developers. We provide an implementation and evaluation of the system on an industrial-strength data stream processing platform to validate our solution.

Index Terms—Data stream processing, parallelization, elasticity

1 Introduction

As the world becomes more interconnected and instrumented, there is a deluge of data coming from various software and hardware sensors in the form of continuous streams. Examples can be found in several domains, such as financial markets, telecommunications, manufacturing, and healthcare. In all of these domains there is an increasing need to gather, process, and analyze these data streams to extract insights as well as to detect emerging patterns and outliers. Most importantly, this analysis often needs to be performed in near real-time.

Stream computing is a computational paradigm that enables carrying out analytical tasks in an efficient and scalable manner. By taking the incoming data streams through a network of operators placed on a set of distributed hosts, stream computing provides an on-the-fly model of processing. Since the data is not directly stored on disk, stream computing avoids the performance problems faced by the more traditional store-and-process model of data management. The emergence of commercial stream processing systems, such as StreamBase [28] and InfoSphere Streams [16], open source systems such as S4 [34] and Storm [27], as well as existing academic systems such as STREAM [5], Borealis [1], and System S [18], is evidence for the future growth and past success of the stream computing paradigm.

The frequent need for handling large volumes of live data in short periods of time is a major characteristic of stream processing applications [26]. Thus, supporting high throughput processing is a critical requirement for streaming systems [14]. It necessitates taking advantage of multiple host machines to achieve scalability [4]. This requirement will become even more prominent with the ever increasing amounts of live data available for processing. The increased affordability of distributed and parallel computing, thanks to advances in cloud computing and multi-core chip design, has made this problem tractable. This creates a demand for language and system level techniques that can effectively locate and efficiently exploit parallelization opportunities in stream processing applications. This latter aspect is the focus of this article.

Streaming applications are structured as directed graphs where vertices are operators and edges are data streams. To scale such applications, the stream processing system is free to decide how the application graph will be mapped to the set of available hosts. *Auto-parallelization* [24] is an effective technique that can be used to scale stream processing applications in a *transparent* manner. It involves detecting *parallel regions* in the application graph that can be replicated on multiple hosts, such that each instance of the replicated region (which we refer to as a *channel*) handles a subset of the data flow in order to increase the throughput. This form of parallelism is known as *data parallelism*. Transparent data parallelization involves detecting parallel regions without direct involvement of the application developer and applying runtime mechanisms to ensure *safety*: the parallelized application produces the same results as the sequential one.

While safety ensures correctness, it does not ensure improved performance. Transparent auto-parallelization that improves performance must have some *profitability* mechanism. In a streaming data-parallel region, profitability involves determining the right degree of parallelism, which is the number of parallel channels to be used, without explicit involvement of the application developer. *Elastic* auto-parallelization takes this one step further by making the profitability decisions adaptive to the runtime dynamics such as changes in the workload and the availability of resources. In this article, we propose novel

• *B. Gedik is with the Computer Engineering Deparment, Bilkent University, Ankara 06800, Turkey. E-mail: bgedik@cs.bilkent.edu.tr.*
• *S. Schneider, M. Hirzel, and K.-L. Wu are with the IBM Thomas. J. Watson Research Center, Yorktown Heights, NY 10598 USA. E-mail: {scott.a.s, hirzel, klwu}@us.ibm.com.*

Manuscript received 27 Feb. 2013; revised 9 Nov. 2013; accepted 20 Nov. 2013. Date of publication 4 Dec. 2013; date of current version 16 May 2014. Recommended for acceptance by M. Kandemir.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.295

techniques that provide effective elastic auto-parallelization for stream processing applications.

There are two important requirements in achieving elastic auto-parallelization.

(*R1*) *Elasticity in the presence of stateful operators requires state migration*. This brings about two major challenges. First, in order to maintain the transparent nature of auto-parallelization, state migration needs to be performed in the presence of, but without interfering with, application logic. Second, we need to minimize the amount of migrated state. By minimizing migrated state, we minimize the time and space overhead which can disturb the flow of data. This will, in turn, enable more frequent adaptation.

(*R2*) *Elasticity in the presence of runtime dynamics requires control algorithms*. This brings about two major challenges. First, general purpose stream processing applications contain a large number of user-defined operators. We cannot easily model the reactions of such operators. Addressing this challenge requires making use of runtime metrics to guide a control system, rather than relying on traditional cost based optimization. Second, we need to make sure that the control algorithm is able to provide SASO properties [12]. That is, it exhibits *stability* (does not oscillate the number of channels used), achieves good *accuracy* (finds the number of channels that maximizes the throughput), has short *settling time* (reaches a stable number of channels quickly), and finally, avoids *overshoot* (does not use more channels than necessary).

We address the challenge of transparent migration by developing a key-value store based state API that is designed to support the implementation of *partitioned stateful* operators. Partitioned stateful operators store independent state for each sub-stream identified by a partitioning attribute [24]. Such operators are very common in stream processing applications (network traces partitioned by IP numbers, financial streams partitioned by stock tickers, etc.). We develop compile-time rewrite techniques to convert high-level user code into an equivalent version that uses the state API, so as to shield application developers from the details of state migration.

We address the challenge of low-cost migration by developing an incremental migration protocol and an associated splitting strategy based on consistent hashing [19], which together minimizes the amount of migrated state.

We address the challenge of runtime control by relying on two local metrics computed at the splitter: the *congestion index* (a measure of blocking time at the splitter) and the *throughput*. The splitter is a run-time component that is colocated with the operator generating the stream to be split for parallel processing. We develop a local control algorithm that works at the splitter and uses these metrics to adjust the number of channels to be used for processing the flow.

We address the challenge of providing SASO properties by incorporating several techniques in our control algorithm. These include peeking up and down in terms of the number of channels used based on changes in observed metrics to address accuracy and overshoot; remembering the past performances achieved at different operating points to address stability; and *rapid scaling* to address settling time.

This article makes the following major contributions:

- To the best of our knowledge, it provides the first elastic auto-parallelization scheme that can handle stateful operators, works across multiple hosts, and is designed for general purpose stream processing applications.
- It proposes a state management API, compile-time rewrite techniques, and a run-time migration protocol to perform transparent state migration with minimal state movement.
- It proposes a control algorithm that uses local information and local control to achieve SASO properties, in order to find the best operating point to solve the profitability problem in a workload and resource adaptive manner.
- It provides an implementation and an evaluation on an industrial-strength stream processing system.

The techniques and algorithms we introduce for achieving elasticity, such as the control algorithm, the state management techniques, and the migration protocol, all have general applicability and can be implemented in any stream processing system. We exemplify the state management APIs using a prototype version of IBM InfoSphere Streams, Streams for short, and its programming language SPL [13].

In summary, this article shows how to solve the profitability problem for distributed stateful stream processing, in a transparent and elastic manner. The rest of the article is organized as follows. Section 2 provides background on Streams and SPL [13], and overviews the safety aspects of auto-parallelization based on previous work [24]. Section 3 provides an overview of our elastic auto-parallelization solution. Section 4 describes the control algorithm and how it achieves the SASO properties. Section 5 describes the key-value based state APIs and the compile-time rewrite techniques. Section 6 elaborates on the migration protocol. Section 7 reports experimental results. Section 8 discusses related work, and Section 9 concludes the article.

2 Background

This section briefly discusses the SPL language and the Streams middleware, and then gives background on the safety aspects of auto-parallelization.

2.1 SPL and Streams

SPL [13] is a programming language used to develop stream processing applications. SPL applications are composed of *operator instances* connected to each other via *stream connections*. An operator instance is a vertex in the application's data flow graph. An operator instance is a realization of an *operator definition*. For example, the operator instance shown on the left of the top graph in Fig. 1 is an instance of the `TCPSource` operator. In general, operators can have many different instantiations, each using different stream types, parameters, or other configurations such as windows. Operator instances can

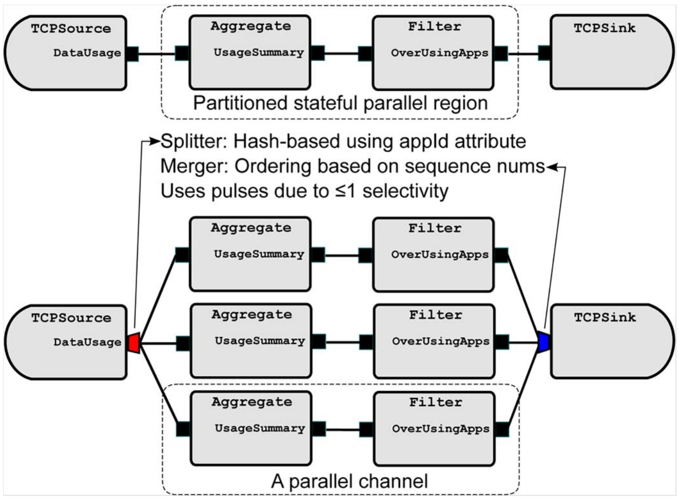

TCPSource
DataUsage
Aggregate
UsageSummary
Filter
OverUsingApps
TCPSink
Partitioned stateful parallel region
Splitter: Hash-based using appId attribute
Merger: Ordering based on sequence nums
Uses pulses due to ≤1 selectivity
A parallel channel

Fig. 1. Auto-parallelization of the OpMon application.

```
composite OpMon {
  param
    expression<float32> $threshold :
        (float32) getSubmissionTimeValue("threshold");
  type
    UsageDetailInfo = tuple<int32 srcIndex, timestamp time,
        int32 trafficSrc, int32 trafficDst, int32 appId,
        int32 appVersion, float32 flowAmount>;
    UsageSummaryInfo = tuple<int32 appId, float32 flowAmount,
        timestamp time>;
  graph
    stream<UsageDetailInfo> DataUsage = TCPSource() {
      param
        role: server; port: 40000; format: csv;
    }
    stream<UsageSummaryInfo> UsageSummary = Aggregate(DataUsage) {
      window
        DataUsage: tumbling, delta(time, 60.0), partitioned;
      param
        partitionBy: appId;
      output
        UsageSummary: flowAmount = Sum(flowAmount);
    }
    stream<UsageSummaryInfo> OverusingApps = Filter() {
      param
        filter: flowAmount > $threshold;
    }
    () as Sink = TCPSink(OverusingApps) {
      param
        role: client; address: "10.0.0.2"; port: 40001;
    }
}
```

Listing 1. OpMon, a simple operational monitoring app. in SPL.

have zero or more input and output *ports*. Each output port generates a uniquely named *stream*, which is a sequence of tuples. Connecting an output port to the input of an operator establishes a stream connection. A stream connection is an edge in the application's data flow graph.

Operators are implemented either directly in SPL (via Custom operators) or in general purpose programming languages. In both cases, the operator implementations rely on an event driven interface—they react to tuples arriving on operator input ports. Tuple processing generally involves updating some operator-local state and producing result tuples that are sent out on the output ports.

Streams [16] is a distributed stream processing engine that can execute SPL applications using a set of distributed hosts. It performs various runtime tasks, such as data transport, scheduling, fault-tolerance, and security.

2.2 Auto-Parallelization

Auto-parallelization is the process of automatically discovering data-parallel regions in an application's flow graph which can be exploited at runtime. In addition to discovering these parallel regions, the compiler must also establish certain properties required to activate appropriate runtime mechanisms that will ensure safety of the auto-parallelization. For instance, if a parallel region is stateless, the runtime data splitting mechanism to be applied can be round-robin, whereas if the region is partitioned stateful, the data splitting has to be performed using a hash-based scheme.

We illustrate an example auto-parallelization process using the SPL code sample given in Listing 1. Here, we see a sample operational monitoring application called OpMon. An instance of the TCPSource operator is used to receive a stream that contains information about network usage of different applications. This is followed by an Aggregate operator instance, which computes minute-by-minute data usage information for each application, using the appId as the partitioning key. The aggregated results are taken through a Filter operator to retain applications whose network usage is above a threshold. Finally, the end results are sent to a TCPSink operator instance.

Fig. 1 shows a visualization of the data flow graph for the OpMon application (at the top), as well as an auto-parallelized version of it (at the bottom). In this example, the parallel region consists of the Aggregate and the Filter operators. It is a partitioned stateful parallel region, as the Aggregate operator maintains state on a per-partition basis. The splitter resides on the output port of the operator preceding the parallel region, and is responsible for routing tuples to parallel channels. Once a tuple is routed to a particular channel, then all future tuples that share their appId attribute value with it must be routed to the same channel. This is achieved at the splitter by applying a hash function on the appId attribute.

In this example, there is an additional operator that follows the parallel region (the TCPSink). Furthermore, there is no indication that this operator can tolerate out of order results. As such, this particular parallel region needs to maintain the order of tuples at its output. This is achieved at the merger, which resides on the input port of the operator succeeding the parallel region. The merger performs a re-ordering operation using sequence numbers which were assigned at the splitter and carried through the parallel region.

Finally, this parallel region contains a Filter operator that can drop some of the tuples. This results in a selectivity value of at most 1. Therefore, the merger may block for long periods of time, if the tuples for a given channel happen to get dropped with a higher frequency than others. This is because during times of no tuple arrival, the merger cannot differentiate between tuples that take a long time to arrive and tuples that will never arrive (dropped). To solve this problem, this particular parallel region uses *pulses*, which are special markers periodically sent by the splitter and used by the merger to avoid lengthy stalls.

Complete details of the safety aspects of auto-parallelization can be found in [24]. In summary, the following properties [33] of parallel regions play a central role in the runtime mechanisms used to ensure safety:

- **Statefulness** determines whether a region can be parallelized, as only stateless and partitioned stateful operators are amenable to data parallelism. It also determines the data partitioning scheme used at the splitter.
- **Ordering** requirements of the downstream operators determine whether a parallel region requires an ordering step during the merge or not.
- **Selectivity** of a parallel region determines whether pulses are required to avoid lengthy stalls or not.

In the rest of the article, we look at how the profitability aspect of auto-parallelization can be addressed through run-time adaptation.

3 Solution Overview

In this section we give an overview of our solution, which is based on run-time elasticity.

The key idea of our approach is to leave the profitability decision to run-time, where we can infer workload and resource availability. When an application starts its execution, the number of parallel channels is set to 1. A control algorithm placed at the splitter periodically re-evaluates the number of channels to be used based on local run-time metrics it maintains. The control algorithm can decide to increase or decrease the number of channels used or take no action at any decision point. When the number of channels to use changes, then a state migration protocol may need to be executed if the parallel region is stateful.

It is important to note that we are not addressing the placement problem in this work. In particular, when a new parallel channel is requested by our algorithm, we assume that it will be placed on available hosts/cores in the system.

For parallel regions that are partitioned stateful, changing the number of parallel channels necessitates *partial* relocation of state. For instance, if the number of parallel channels increases, then the assignment of some of the partitions needs to move from the existing parallel channels to the new parallel channels. Whenever such change of assignment happens at the splitter, the state associated with the moved partitions has to be relocated as well. In particular, the newly added parallel channels need collect the state of the partitions assigned to them from the existing parallel channels. Similarly, when existing channels are removed, the state associated with the partitions they were handling has to be redistributed to the existing parallel channels.

As a system invariant, each partition is owned by only a single parallel channel at any given point in time. We perform the assignment of partitions to parallel channels using consistent hashing in order to minimize the amount of state moved during migration.

In order for run-time migration to be performed transparently, the stream processing middleware has to reason about the state maintained by operators. In a general purpose streaming system where user-defined operators are commonly used, this requires special machinery. To address this problem, our solution includes a state management API in the form of a local key-value store. The SPL compiler rewrites code present in `Custom` operators such that the state is converted to use this API, enabling the runtime to reason about such state and perform transparent migration.

4 Control Algorithm

The control algorithm is run periodically to update the number of parallel channels. It relies on the following two locally computed metrics:

Congestion is an indication of whether the splitter observes an undue delay when sending tuples on a connection. It is a useful metric in two respects: 1) Presence of congestion is an indication that we need more channels to handle the current load, and similarly lack of congestion is an indication that we may be using more channels than necessary. 2) Temporal changes in the congestion value can indicate changes in the workload availability. We compute the congestion as a boolean value by applying a threshold on the *congestion index*, which is a measure of the fraction of time the tuple transport at the splitter is blocked due to backpressure.

To compute the congestion index, we use non-blocking I/O for transferring tuples. If the send call notifies us that the call would block, then we block until room is available and measure the amount of blocking involved. Overall, the congestion index measures the fraction of time spent blocking. We average this value over all channels. The congestion index is a value in the range [0, 1]. We further discuss the congestion index threshold in Section 4.3.

Throughput is the number of tuples processed per second over the last adaptation period. Throughput is useful in two respects: 1) When we move to a new operating point in terms of the number of channels, it tells us if the situation has improved or not—after all, the goal is to optimize the throughput. 2) Temporal changes in the throughput can indicate changes in the workload.

The control algorithm operates based on the following two fundamental principles:

(P1) *Expand*: If there is congestion, *go up* (increase the number of channels) unless you have been there before and have not observed improved throughput.

(P2) *Contract*: If there is no congestion, *go down* (decrease the number of channels) unless you have been there before and have observed congestion.

Here, (P1) provides the accuracy property in SASO: we get good accuracy, since the number of channels is increased until congestion is removed. (P2) provides the overshoot property in SASO: we avoid using more channels than necessary, since the number of channels is decreased unless congestion appears below. The 'unless' clauses in (P1) and (P2) provide the stability property in SASO: we do not oscillate between operating points, since what happened in the past is remembered and is not repeated. However, these two principles are not sufficient

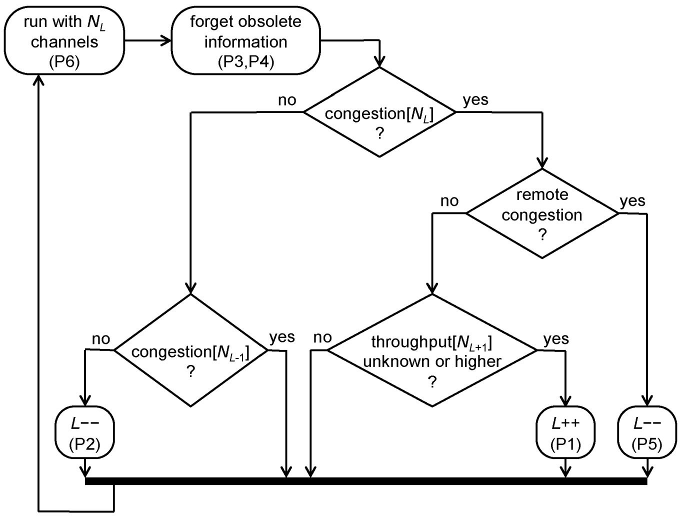


Fig. 2. High-level description of the control algorithm.

when the workload fluctuates. When the workload availability changes, we need to *forget* part of what happened in the past. Thus, we introduce the following adjustments in order to adapt to workload changes:

(P3) *Congestion Adapt*: If a change is observed in congestion that is indicative of workload increase (decrease), forget about past observations regarding upper (lower) channels.

(P4) *Throughput Adapt*: If a change is observed in throughput that is indicative of workload increase (decrease), forget about past observations regarding upper (lower) channels.

Here, (P3) and (P4) enable the control algorithm to adapt to workload changes. Note that the control algorithm has the following critical feature: It will settle upon a number of channels such that there is no congestion, yet any smaller number of channels will result in congestion. When in this stable state, if there is an increase in the workload, it will start observing congestion and go up (due to (P1)). When there is a decrease in the workload, it will observe throughput decrease and forget about the fact that one channel below was resulting in congestion (due to (P4)) and go down (due to (P2)).

There are two additional minor issues with this version of the control algorithm. The first one is about the *nature* of congestion. The algorithm is designed to interpret presence of congestion as an indication of the need for more channels to handle the load. In the case that the congestion is *not* due to the cost of the parallel region but instead due to the cost of the flow that is downstream of the parallel region, increasing the number of channels will not result in any improvement, but instead will cause overshoot. We call this kind of congestion *remote congestion* and avoid the potential pitfall using the following additional principle:

(P5) *Remote Congestion*: If the congestion continues after increasing the number of channels, yet the throughput has not significantly increased, go down.

Here, (P5) avoids the case where the number of channels is continuously increased due to the continued presence of congestion, yet the throughput does not improve. Without (P5), this can happen in the presence of remote congestion. It is important to note that most streaming applications eventually hit a scalability limit when their original bottlenecks are removed via parallelization. This is because the bottleneck moves to a non-parallelizable portion of the application, which in most cases is the source, the sink, or some stateful operator. In all cases where an operator downstream of a parallel region becomes the bottleneck, remote congestion will occur.

The second problem is that, in cases where the available resources (execution contexts such as hosts and cores) and the cost of the parallel region are both high, the optimal number of channels can be high as well. Thus, it can take a long time for the control algorithm to reach this number. This is due to the one-channel-at-a-time nature of the algorithm and can negatively impact the settling time property in SASO. We address this problem by introducing an option to our algorithm called *rapid scaling*. It is summarized a follows:

(P6) *Rapid Scaling*: Rather than operating one-channel-at-a-time, operate one-level-at-a-time and define a super-linear mapping between the number of levels and channels.

Here, (P6) preserves the main operation mode of the algorithm by still making changes on the operating point one-step-at-time. But rather than using number of channels directly, it uses a *level*, which is mapped to the number of channels via a function. In particular, we use the following function:

$$N_L = \left\lfloor 0.5 + 2^{0.5*(L+1)} \right\rfloor.$$

For increasing level values starting at 0, this results in the following series of number of channels: {1, 2, 3, 4, 6, 8, 11, 16, 23, 32, ...}. It is possible to use other functions that follow a more steep or less steep curve depending on the maximum number of channels and the settling time requirements.

Fig. 2 gives a high-level description of the control algorithm, illustrating the principles P1 through P6.

4.1 Algorithm Implementation

The control algorithm keeps three state variables. The first one is the current adaptation period, denoted by P. The second one represents the current level, denoted by L. The third one is an array that keeps the following information for each level: the last adaptation period during which the algorithm was at this level, denoted by P_i; whether congestion was observed the last time the algorithm was at this level, denoted by C_i; the throughput observed the last time the algorithm was at this level, denoted by $T_i^{\dashv}$; and the throughput observed during the first of the periods the last time the algorithm stayed consecutive periods at this level, denoted by $T_i^{\vdash}$. We use L^* to denote the maximum number of levels.

The control algorithm has a global parameter called *change sensitivity*, denoted by α, which determines what *significant change* means. It takes a value in the range [0,1]. A value of 1 means the algorithm is very sensitive to small changes in the throughput. For instance, a minor improvement in the throughput will be sufficient to go up if the sensitivity is high. All changes in throughput are normalized against the ideal throughput for a single channel in a linearly scaling system.

Algorithm 1: Initialization of the state variables.

```
procedure $init()$
  $P \leftarrow 1$; $L \leftarrow 0$;
  $\forall_{i\in[0\ldots L^*)}(T_i^{\vdash} \leftarrow nan; T_i^{\dashv} \leftarrow \infty)$
  $\forall_{i\in[0\ldots L^*)}(C_i \leftarrow true; P_i \leftarrow -1)$
  $s \leftarrow 0.1 + (1.0 - \alpha) * 0.9$
```

Algorithm 2: Update of the number of channels.

```
Require: $T$: current throughput, $C$: current congestion
procedure $getNumberOfChannels(T, C)$
  /*(P3) and (P4): congestion and throughput adapt*/
  $l_c \leftarrow checkLoadChangeViaCongestion(C)$
  $l_t \leftarrow checkLoadChangeViaThroughput\ (T)$
  if $l_c = LessLoad$ or $l_t = LessLoad$
    $\forall_{i\in[0\ldots L)} C_i \leftarrow false$; $T_i^{\dashv} \leftarrow 0$
  if $l_c = MoreLoad$ or $l_t = MoreLoad$
    $\forall_{i\in(L\ldots L^*)} C_i \leftarrow true$; $T_i^{\dashv} \leftarrow \infty$
  /*update info on current level */
  $P_L \leftarrow P$; $P \leftarrow P + 1$
  $T_L^{\dashv} \leftarrow T$; $C_L \leftarrow C$
  if $T_L^{\vdash} = nan$ then $T_L^{\vdash} \leftarrow T$
  /* update the current level */
  $r \leftarrow (P_{L-1} = P_L - 1)$ and $C_{L-1}$ and $C_L$ and $T_L^{\dashv} \le T_{L-1}^{\dashv}$
  if $r$ /* (P5): remote congestion*/
    $T_{L-1}^{\vdash} \leftarrow nan$; $L \leftarrow L - 1$
  else if $C$ /* (P1): expand */
    if $L < L^* - 1$ and $T_{L+1}^{\dashv} \ge T$
      $T_{L+1}^{\vdash} \leftarrow nan$; $L \leftarrow L + 1$
  else /*(P2): contract */
    if $L > 0$ and $\neg C_{L-1}$
      $T_{L-1}^{\vdash} \leftarrow nan$; $L \leftarrow L - 1$
  return $N_L$ /* (P6): rapid scaling */
```

The $init()$ routine in Algorithm 1 provides the initialization logic for the state variables. The core of the algorithm is given in the $getNumberOfChannels()$ routine provided by Algorithm 2, which takes as parameters the current throughput (denoted by T) and the current congestion status (denoted by C). As the first step, principles (P3) and (P4) are applied to see if there is a change in the load. If there is a load increase (decrease), the information kept about the levels above (below) are reset. The second step updates the information kept about the current level, last throughput, last congestion status, and the first throughput (unless the algorithm was at this level the last time). The third step adjusts the current level. First, principle (P5) is applied to see if there is remote congestion (were at this level the last time ($P_{L-1} = P_L - 1$), observed continued congestion (C_{L-1} and C_L), and the throughput did not improve ($T_L^{\dashv} \le T_{L-1}^{\dashv}$)). If so, we go back to the previous level. Otherwise, principle (P1) is applied: We check if there is congestion and if so go up one level unless the algorithm has been there before but the throughput was worse. Finally, principle (P2) is applied, that is if there is no congestion, we go one level down unless the algorithm has been there before and observed congestion. Once the current level is adjusted, principle (P6) is applied to return the channel count corresponding to the current level.

4.2 Detecting Workload Changes

The logic used to detect changes in the workload is given in Algorithm 3. The $checkLoadChangeViaCongestion()$ routine uses the congestion status to detect load changes. If the current level and the last level are the same, yet the congestion status has changed, this is taken as an indication of load change (load increase if there is congestion currently, load decrease otherwise). If the current level is lower than the last one, yet the congestion has disappeared, this is taken as load decrease. And finally, if the current level is higher than the last one, yet the congestion has appeared, this is taken as load increase.

The $checkLoadChangeViaThroughput()$ routine uses the throughput to detect load changes. If the current level and the last level are the same, yet there is a significant change in the throughput, this is taken as an indication of load change (load increase if the current throughput value is higher, load decrease otherwise). Change sensitivity is used to detect significant change relative to the ideal change in a linearly scaling system. If the current level is lower than the last one, yet the throughput has increased, this is taken as load increase. And finally, if the current level is higher than the last one, yet the throughput has decreased, this is taken as load decrease.

Algorithm 3: Detecting workload changes.

```
Require: $C$: current congestion
procedure $checkLoadChangeViaCongestion(C)$
  if $P_L = P - 1$ and $C_L \neq C$
    return $C$ ? MoreLoad: LessLoad
  if $P_{L+1} = P - 1$ and $C_{L+1}$ and $\neg C$
    return LessLoad
  if $P_{L-1} = P - 1$ and $\neg C_{L-1}$ and $C$
    return MoreLoad
  return Unknown
Require: $T$: current throughput
procedure $checkLoadChangeViaThroughput(T)$
  if $P_L = P - 1$
    if $T < T_L^{\vdash}$
      if $(T_L^{\vdash} - T) > s * (N_L - N_{L-1}) * (T_L^{\vdash}/N_L)$
        return LessLoad
    else
      if $(T - T_L^{\vdash}) > s * (N_{L+1} - N_L) * (T_L^{\vdash}/N_L)$
        return MoreLoad
  if $P_{L+1} = P - 1$ and $T > T_{L+1}^{\dashv}$
    return MoreLoad
  if $P_{L-1} = P - 1$ and $T < T_{L-1}^{\dashv}$
    return LessLoad
```

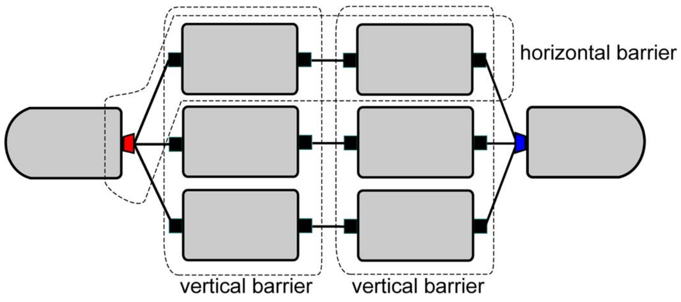


Fig. 3. Vertical and horizontal barriers during migration.

4.3 Discussion of Parameters

The control algorithm has three configurable parameters. Out of these, the congestion index threshold is the only one that requires careful tuning. We study its setting empirically in the experimental evaluation section and show that any threshold in the range [0.01,0.3] provides a robust setting.

Rapid scaling is a feature that can be turned on to reduce the settling time. It adjusts the tradeoff between quick adaptation and the ability to fine-tune the number of parallel channels.

The change sensitivity adjusts the sensitivity of the system to workload changes. In systems where the cost of migration is low, a small value for the change sensitivity parameter (higher sensitivity) is appropriate. For systems where migration is a costly operation, it is best to wait for significant change in the observed throughput before taking an adaptation step. For instance, if the throughput handled by a channel has dropped by 10 percent, reacting to this by going one level down may be too aggressive for a system with high overhead migrations.

Our algorithm satisfies all SASO properties even if the user does not tune rapid scaling or change selectivity. But by offering these parameters, we enable power users to adjust the relative tradeoffs between the SASO properties, if desired.

5 State Management

Operators that participate in parallel regions are either stateless or partitioned stateful, as outlined in Section 2.2. Partitioned stateful operators maintain independent state on a per-partition basis based on a partitioning attribute. Such operators require special machinery to support transparent elastic parallelization. In particular, the runtime system needs to migrate (across hosts) state associated with a subset of the partitions. This requires the runtime to understand the state managed by partitioned stateful operators.

To address this problem, we developed a state management API and an associated state management service. Furthermore, we provide language-level mechanisms to enable newly developed operators to take advantage of managed state. Details about the managed state APIs as well as their transparent use from within SPL applications is covered in Appendix A.

6 State Migration

The migration protocol is executed for a parallel region in response to the decisions made at the splitter by the control algorithm. When the control algorithm updates the number of channels, it also updates the data partitioning function it uses to distribute the partitions among the parallel channels and initiates the migration protocol. The migration is only needed for the case of partitioned stateful parallel regions.

The migration protocol is initiated by sending a *migration pulse* from the splitter to all parallel channels. When an operator in a parallel channel receives a migration pulse, it first forwards the pulse downstream and then starts executing the per-operator migration protocol. This makes it possible to execute migration of state between replicas of multiple operators in parallel, in case the parallel region contains more than one partitioned stateful operator.

We first describe the migration protocol for an operator and then discuss the implications of using different data partitioning functions on system performance.

6.1 Migration Protocol

The $migrate$ routine given in Algorithm 4 provides the pseudo-code for the migration protocol executed by an operator. There are four parameters to the routine. The first is the index of the operator's parallel channel, denoted by i. The second is the new operating point in terms of the number of channels, denoted by N. The third is the state kept locally at this operator, which consists of a list of managed stores, denoted by S_i where $s_i^k \in S_i$ denotes one of the stores. The last is the data partitioning function generator, which generates a data partitioning function given the number of parallel channels, denoted by $\mathcal{H}$.

The protocol has two phases, namely the *donate* phase and the *collect* phase. In the donate phase, the items that do not belong to the current operator—after the data partitioning function has been updated based on the new number of channels—are collected into a *package*. Package $\Delta_{i \to j}^k$ represents the set of data items in s_i^k that needs to migrate from the operator replica running on the ith channel to the replica running on the jth channel. These items are removed from the in-memory store s_i^k. The resulting packages are stored on a backing store and then a *vertical barrier* is performed across replicas of the operator. This ensures that all replicas complete the donate phase before the collect phase starts.

In the collect phase, packages in the backing store that are destined to the current operator replica are retrieved and the in-memory stores are updated. For instance, items in package $\Delta_{j \to i}^k$ are added to the store s_i^k. A vertical barrier is performed to ensure all replicas have completed the collect phase. Once complete, a *horizontal barrier* is performed, in order to ensure that the splitter does not start sending tuples before the migration is complete. This barrier is performed across the *master* operator replicas (at index 0) and the splitter.

Consider a parallel region with 2 operators and 3 parallel channels, as shown in Fig. 3. During a vertical barrier the set of 3 replicated operators synchronize with each other, whereas during a horizontal barrier the 2 operators on channel 0 synchronize with the splitter.

Algorithm 4: Migration algorithm for an operator.

```
Require: $i$: index of this operator's parallel channel
Require: $N$: number of parallel channels to migrate to
Require: $S_i$: state kept locally at this operator instance
Require: $\mathcal{H}$: data partitioning function generator
procedure $migrate(i, N, S_i, \mathcal{H})$
  $H_N \leftarrow \mathcal{H}(N)$
  /* Donate phase */
  for each store $s_i^k \in S_i$ do
    $\forall_{j \neq i, j \in [0 \dots N)} \Delta_{i \to j}^k \leftarrow \{\tau \mid H_N(\tau) = j \wedge \tau \in s_i^k\}$
    $\forall_j s_i^k \leftarrow s_i^k \setminus \Delta_{i \to j}^k$
    Save $\Delta_{i \to j}^k$ to backing store
  $verticalBarrier()$
  /* Collect phase */
  for each store $s_i^k \in S_i$ do
    $\forall_{j \neq i, j \in [0 \dots N)}$ retrieve $\Delta_{j \to i}^k$ from backing store
    $\forall_j s_i^k \leftarrow s_i^k \cup \Delta_{j \to i}^k$
  $verticalBarrier()$
  if $i = 0$
    $horizontalBarrier()$
```

Our implementation of the migration protocol works across multiple machines and does not rely on shared memory. It makes use of a back-end database for state movement and synchronization. Alternative implementations are possible (e.g., sockets or MPI). Using a database has advantages, such as periodic checkpointing of managed state for fault-tolerance, which is beyond this article's scope.

6.2 Data Partitioning

Data partitioning is performed at the splitter for partitioned stateful parallel regions. The partitioning function needs to be updated when the number of parallel channels changes. The choice of the partitioning function impacts the cost of the migration, as it changes the amount of migrated state.

Consider a scenario where a simple data partitioning function is used, which applies a hash function on the partitioning attributes and mods the result based on the number of channels. This data partitioning function can result in massive state migrations. Worse, it results in moving some partitions across channels that are present both before and after the migration. In general, we need a data partitioning function that provides good *balance* and *monotonicity*. Balance ensures that the partitions are uniformly distributed across channels, achieving good load balance. Monotonicity ensures that partitions are not moved across channels that are present before and after the migration. Consistent hashing [19] is a technique that provides these properties.

Consistent hashing maps each data item to a point on a 128-bit ring in uniformly random fashion. Similarly, each channel is also mapped to the same ring, but rather than to a single point, each channel is mapped to multiple points on the ring (using multiple hash functions). A data item is assigned to the channel that is closest to it on the ring. As a result of this scheme, when a new channel is inserted, it collects data items from multiple of the existing channels. Similarly, when a channel is removed, its data items are distributed over multiple of the existing channels. Consistent hashing ensures that on average M/N partitions are moved when the Nth channel is inserted or removed from a system with M partitions.

Consistent hashing can be implemented in $O(1)$ time (by dividing the ring into segments [19]), yet it is slightly more costly to compute compared to a simple hashing scheme. However, it minimizes the amount of state to be moved during migration. Our migration algorithm given in Algorithm 4 can work with any data partitioning function.

7 Experimental Results

This section evaluates the effectiveness of our solution based on experimental results. Two kinds of results are presented. First, micro-benchmarks are used to evaluate the scalability, adaptation, congestion index threshold sensitivity, and migration time properties. These results are presented in Appendix B. Second, real-world application kernels are used to compare the throughput achieved by our elastic auto-parallelization scheme with the optimal throughput.

7.1 Experimental Setup

We implemented our elastic auto-parallelization scheme in C++, as part of the SPL runtime within Streams. The backing store used for migration is a DB2 database.

Our experiments were performed on 4 machines, each with 2 3 GHz Intel Xeon processors containing 4 cores (8 cores per machine). Each machine runs Linux with a 2.6 kernel version and has 64 GB of memory. In all experiments, the adaptation period is 5 seconds and the change sensitivity is 0.5. The congestion index threshold is 0.2 (see Section B.3). The error bars are plotted based on 3 repeated runs.

7.2 Application Benchmarks

In this section we look at the performance achieved by the elastic auto-parallelization scheme under several application kernels, some of which has been used in previous work [24]. We compare the results obtained from elastic scaling against the performance of a fixed number of channels, by experimenting with different numbers of channels for the latter. Note that for applications with more than one parallel region, every region settles independently on its own.

The application kernels, shown in Fig. 4, are:

Finance: This application computes the volume weighted average price of trades on a per stock ticker basis and compares the results to the quote values in order to detect bargains. As the final result, it computes a bargain index for the quotes that are considered profitable. It uses real-world stock market data as input. There are three parallel regions in this application, two of which are stateless and one is partitioned stateful.

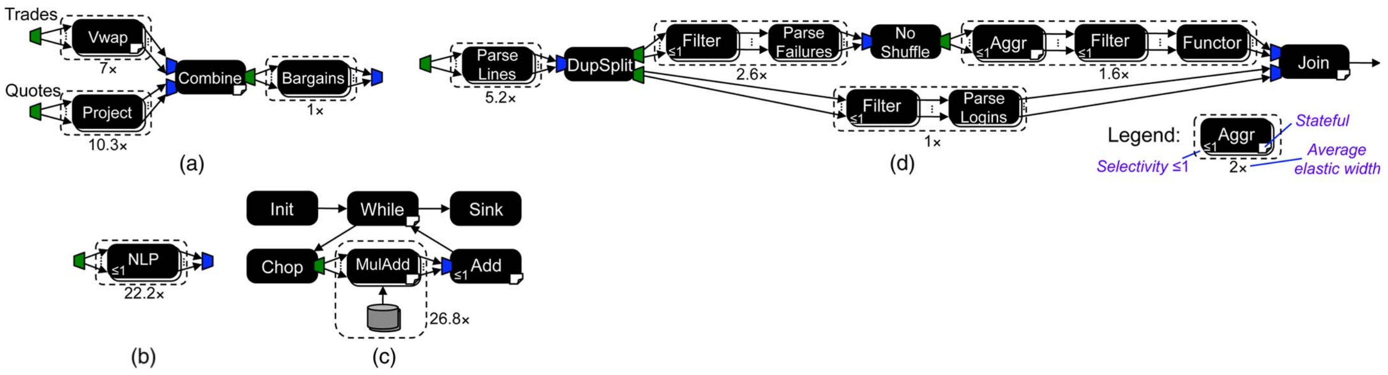


Fig. 4. Stream graphs for the application kernels. (a) Fiannce. (b) Twitter NLP. (c) PageRank. (d) Network monitoring.

Twitter: This application applies basic text analytics on Twitter messages (aka tweets) in order to identify key words used and basic message statistics. The application uses real-world tweet data as input. It contains a single stateless parallel region.

PageRank: This application uses a feedback loop to iteratively rank pages in a web graph [8]. This kind of application is typically associated with MapReduce [9], but is also easy to express in a streaming language. The input to the application consists of a synthetic graph of 2 million vertices and 4 billion uniformly distributed edges (sparsity of 0.001). It contains a single stateful parallel region.

Network Monitoring: This application monitors Linux log files, looking for successful logins that were preceded by many failed logins from the same host. Such logins are flagged as breakins. There are four parallel regions, one of which is stateful. The input is synthetic data based on real data collected from a public-facing server which experienced a breakin attempt about every 2 seconds for a 12 hour period. The real data has been modified with several fake breakins, and the 12 hour period is cycled through 2000 times.

We present the results for these application kernels by plotting the speedup in throughput as a function of the number of channels used. We plot the speedup for the case of a fixed number of channels as well as for the ideal case of linear scalability. We also plot the speedup for the elastic auto-parallelism, which adjusts the number of channels automatically. The speedup reported for the elastic scenario is measured after the control algorithm has settled down on a number of channels for the parallel regions. Each experiment is run at least three times. In all of the applications the speedup achieved shows a linear trend only up to a certain number of parallel channels, after which point additional speedup is not possible. This is because the parallel regions stop being the bottleneck of their application once sufficient parallelism is introduced. The sequential parts of the applications, especially the I/O bound sources and sinks, become the bottleneck. One of the major strengths of the elastic approach is to automatically find the point after which additional parallelism does not help.

Finance: Fig. 5 plots the speedup for the Finance application. The best speedup with a fixed number of channels is 3.1×, and elastic scalability achieves slightly higher throughput around 3.6×. This particular application has three parallel regions, yet the fixed approach uses the same number of channels for all of them. The elastic approach has the flexibility to adjust the number of channels for each region independently. In this case, one of the parallel regions is not profitable.

Twitter: Fig. 6 plots the speedup for the Twitter application. The speedup achieved by the elastic approach is almost as good as the fixed one (around 12× vs. around 10×). The elastic approach uses approximately 22 channels, whereas the fixed approach uses 16 channels when it achieves the highest speedup. This is due to the use of rapid scaling, which means not all numbers of channels are available to the elastic approach. A better result is achievable without rapid scaling, at the cost of longer settling time.

PageRank: Fig. 7 plots the speedup for the PageRank application. In this experiment we used 8 machines instead of 4, in order to have more memory, as the web graph is distributed and cached in memory. The speedup achieved by the elastic auto-parallelism is around 5×, whereas the fixed approach achieves a speedup value of around 6.5×. The best speedup is achieved with 16 channels for the fixed approach vs. around 27 channels for the elastic approach.

Network Monitoring: Fig. 8 plots the speedup for the Network Monitoring application. This experiment uses 5 machines, giving each parallel region a node to itself, and placing the other operators on a separate node. The elastic approach achieves 2.8× speedup, compared to the best

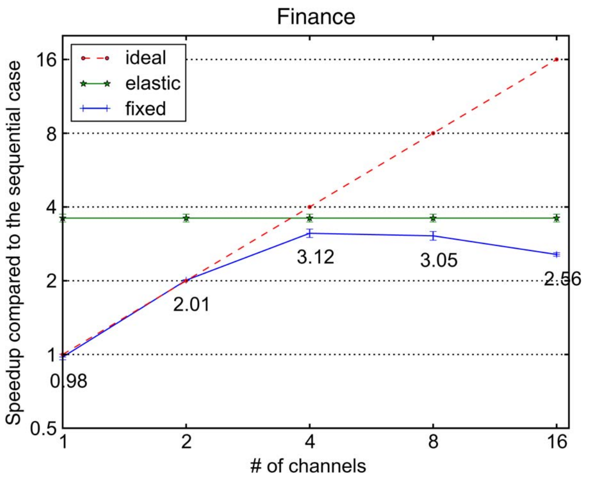


Fig. 5. Performance for the Finance application.

speedup of 3.8× given a static configuration. There are four independent parallel regions in this application, as shown in Fig. 4d. Without auto-parallelization, the *ParseLines* operator is the bottleneck of the application. The elastic algorithm dynamically discovers this bottleneck and settles on 4-8 channels for that region. As more resources are given to *ParseLines*, the *ParseFailures* operator becomes the bottleneck of the application. Its elastic algorithm responds by allocating it 2-4 channels, until the bottleneck of the application shifts to be operators that cannot be parallelized (*DupSplit* and *NoShuffle*). Note that there are independent control algorithms controlling each parallel region; there is no master algorithm that decides what number of channels to use for all regions. Instead, as the elastic algorithm controlling *ParseLines* uses more channels, its throughput increases, which increases the pressure on *ParseFailures*. The elastic algorithm controlling *ParseFailures* responds to the increased workload by using more channels, thus enabling *ParseLines* to use even more channels. This feedback loop continues until the bottleneck is not controlled by an elastic algorithm. The *Logins* path receives very few tuples, and the elastic algorithm correctly uses only a single channel for it.

7.3 Evaluation Summary

In summary, the evaluation presented in Appendix B and in this section shows that our elastic auto-parallelization scheme:

- performs parallelization with small overhead, especially for the partitioned stateful cases when additional processing performed on a per tuple basis is non-trivial;
- adapts to changes in the workload, both when the workload change is smooth and when it is sharp;
- locates the channel count that gives close to optimal performance without causing overshoot, in real-world settings.

8 Related Work

Compared to prior work, our elastic parallelization scheme is the first that meets all three of the following criteria: adjusts the level of parallelism at runtime; adapts to workload changes; and works in the presence of stateful operators. Our earlier work [24] has addressed the last part. It provides language and runtime support for auto-parallelization in the presence of stateful operators, which we classify as the *safety* problem. This work addresses the profitability problem, providing runtime degree-of-parallelism adaptation in the presence of workload changes.

Next, we review the related work in several areas where adaptive and parallel data processing is prevalent.

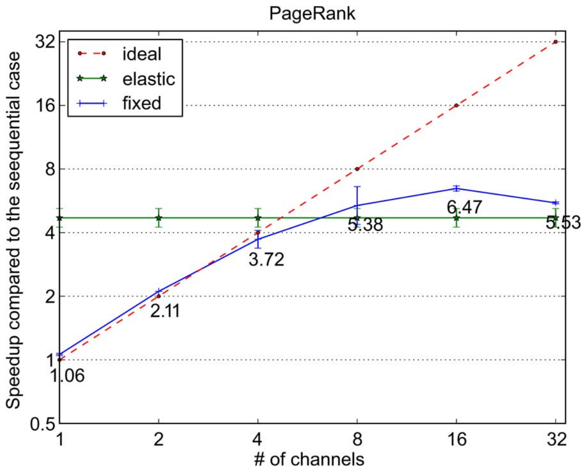


Fig. 7. Performance for the PageRank application.

8.1 Adaptive Query Processing (AQP)

AQP techniques [10] address the rigidity of the traditional optimize-then-execute model of relational database optimizations. The traditional model is unable to adapt to the dynamically changing data, runtime, and workload characteristics of the new breed of data management applications. To solve this, AQP techniques typically apply the *adaptivity loop*, which involves the steps of measure, analyze, plan, and activate. In this work, we follow this general approach as well. However, most of the work on AQP has dealt with the topic of adaptive operator reordering, such as selections [6] and joins [30]. In contrast to previous work on AQP, we address the problem of adaptive partitioned parallelism for general-purpose

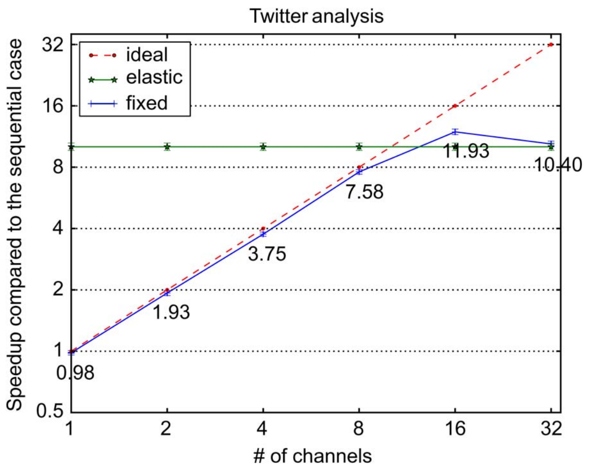


Fig. 6. Performance for the Twitter application.

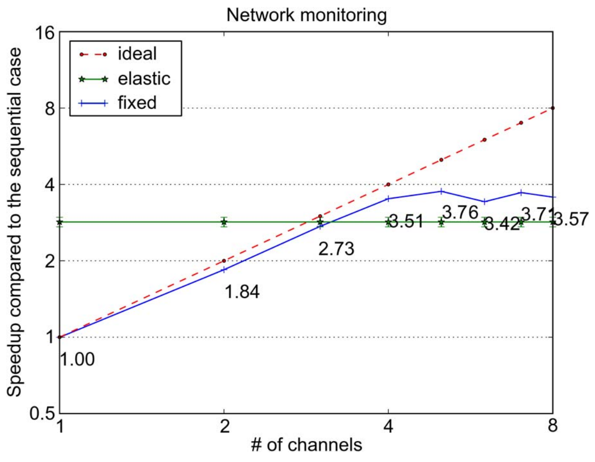


Fig. 8. Performance for the Network Monitoring app.

stream processing systems, where the operators are not limited to relational operators. Our work does not involve operator re-ordering and instead focuses on elastic setting of number of parallel channels to use based on workload availability.

The most relevant work from the AQP area is the Flux operator [25], which applies partitioned parallel processing in the context of stateful continuous queries. However, the focus is on dynamic load balancing and the level of parallelism is not dynamically adjusted. The load balancing problem is of particular interest for non-dedicated hosts, where resources can be used by external processes. Comparison of several different approaches for query parallelization under this assumption can be found in the literature [22]. Our work focuses on elastic scaling in a dedicated host setup.

8.2 Data Stream Processing Systems (DSPSs)

Elastic operators [23] address dynamic data parallelism in the context of DSPSs. However, the approach is limited to individual operators, does not handle stateful operators, and works at the thread level, limiting it to a single host. Similar to our work, run-time control mechanisms are applied to adjust the number of parallel operators in order to adapt to run-time dynamics.

Parallelization of stateful operators in DSPSs is addressed in [32]. A distributed shared state mechanism coupled with a split/process/merge model is proposed to facilitate parallelization. A theoretical model is provided to determine the right level of parallelism as well. Compared to our work, this approach does not provide transparent or elastic parallelization. It does not handle partitioned stateful parallelism and is targeted at the most general case of arbitrary state, which requires explicit use of synchronization and shared state.

Open source DSPSs like Storm [27] and S4 [34] can take advantage of partitioned stateful parallelism. However, Storm leaves the profitability decision to the developers and system administrators (the parallelism level is adjustable at runtime without shutting down the application). S4 creates as many parallel operator instances as there are unique values for the partitioning attribute, which is shown to be sub-optimal [4].

Elasticity related work on Map/Reduce systems and additional relevant research is covered in Appendix C.

9 Conclusion

We presented an auto-parallelization scheme that can provide elasticity to stream processing applications. It is able to adjust the number of parallel channels to use at runtime, depending on workload availability. Most importantly, by relying on migration of partitioned state, it is able to handle partitioned stateful operators that are commonly found in stream processing applications. We presented a control algorithm that is able to achieve good throughput, has short settling time, and avoids oscillation and overshoot. We described a state management API and a migration protocol, which together enable elastic parallelization that is transparent to the application developers. Furthermore, elasticity does not interfere with safety and given the same inputs the same outputs are produced in the same order, irrespective of whether there are migrations or not. Experimental results illustrate the effectiveness of our solution in finding an ideal operating point for parallel regions and adjusting this point in the presence of workload dynamics.

Appendix A
State Management

A.1 Managed State APIs

The state management API provides a key-value store interface to operator developers. The API contains a set of operations commonly found in key-value stores, such as `put` (insert a new item), `get` (retrieve an existing item), `has` (check for existence), `remove` (delete an item), and basic iteration constructs. Additional operations are provided to dynamically create and remove stores at run-time, such as `createStore` and `removeStore`. The store API fully supports the SPL type system, allowing all available types to be used as both keys and values. The runtime service that backs up this API has multiple implementations, specialized depending on the *scoping* and *persistence* policies.

Scoping policy determines the visibility of the managed state across different application components. For instance, state with global scope enables operators from different applications to share state, whereas state with job-level scope enables operators from the same application to share state. For the purpose of this work, we rely on operator-level scope, as no sharing is required. This results in fast access to managed state, as the state can be stored in local memory of each operator instance (recall that at any time each partition is owned by a single channel).

The persistence policy determines whether and how the state is saved. For instance, a policy of transient state means that the state is never backed up to disk and is lost upon failures or restarts. Alternatively, a policy of periodically checkpointed state provides transparent, asynchronous, and incremental saving of state to disk. This minimizes the amount of state lost upon failures or restarts. The persistence policy is orthogonal to migration.

A.2 Transparent Usage of the API

To employ the managed state API in a manner that is transparent to the application developers, several language-level techniques can be used. We look at three unique cases:

Case 1

New SPL operators: User-defined operators are written using `Custom` operators in SPL. For newly developed `Custom` operators, we provide two language constructs aimed at making the development of partitioned stateful operators easier for application developers. First, we introduce the `partitioned state` clause to specify the list of state variables to be maintained on a per-partition basis (otherwise done explicitly through `map` data structures). Second, we introduce the `partitionBy` parameter to specify the partitioning attribute to be used for the

```
stream<rstring id, float32 value> Src = MySource() {}
stream<rstring id, float32 value> Res = Custom(Src) {
 logic
   state:
     float32 threshold = 10.0;
   partitioned state:
     mutable float32 storedValue = minFloat32();
   onTuple Src: {
     float32 diff = abs(value-storedValue);
     if (diff > threshold) {
       storedValue = value;
       submit(Src, Res);
     }
   }
 param
   partitionBy: id;
}
```

Listing 2. Assisted state management in a `Custom` operator.

partitioned state. This removes the need to explicitly deal with the `map` data structures in user code, while at the same time simplifying the compiler's job in identifying partitioned stateful `Custom` operators. The compiler will use the state API when generating code for these operators, so as to enable elastic parallelism.

Listing 2 gives example SPL code that uses the new SPL language constructs. In this example, we have a sensor stream carrying tuples with two attributes: a sensor id and a sensor value. The example shows a `Custom` operator that performs basic thresholding. The operator forwards a tuple if and only if its value is more than a threshold higher than that of the last tuple forwarded with the same id. For this purpose it stores the last value submitted inside the variable `storedValue`. Since the variable is declared within the partitioned state section, at run-time one instance of it for each unique id value is created. The `partitionBy` parameter is used to specify that the partitioning is done using the id attribute.

We note that the code is simpler compared to Listing 3 corresponding to Case 2 (described below), as it avoids explicit management of `map` data structures.

Case 2

Existing SPL operators: In legacy code (Case 1 not applicable), partitioned stateful `Custom` operators are created by explicitly using the `map` data structure. However, such operators neither specify a partitioning attribute nor use the managed state API we outlined earlier. To address this problem, compile-time static analysis can be used to identify whether a `Custom` operator is partitioned stateful, and if so locate the partitioning attribute. In order for an operator to be partitioned stateful, all mutable state needs to be using a `map` data structure and all accesses to this structure need to be based on a common index expression that depends on one or more stream attributes (aka the partitioning attributes).

Once a `Custom` operator is identified as partitioned stateful, the map data structure and the accesses to it can be rewritten to use the managed state API, without any involvement by the application developer. Our current prototype does not fully implement this case.

Listing 3 gives SPL code illustrating a `Custom` operator that can be determined to be partitioned stateful after applying static analysis. The code implements the same application from Listing 2. The operator keeps a mutable state variable (`storedValues`), which is a `map` and is

```
stream<rstring id, float32 value> Src = SensorSource() {}
stream<rstring id, float32 value> Res = Custom(Src) {
 logic
   state: {
     float32 threshold = 10.0;
     mutable map<rstring, float32> storedValues = {};
   }
   onTuple Src: {
     if (id in storedValues) {
       float32 diff = abs(value-storedValues[id]);
       if (diff > threshold) {
         storedValues[id] = value;
         submit(Src, Res);
       }
     } else {
       storedValues[id] = value;
       submit(Src, Res);
     }
   }
}
```

Listing 3. Manual state management in a `Custom` operator.

always accessed via the `id` attribute of the incoming stream. Thus, this operator is partitioned stateful, with a partitioning attribute of `id`.

It is worth noting that Case 1 is superior to Case 2 not only from the perspective of system implementation (easier compiler analysis) but also from the perspective of application development (cleaner and shorter code).

Case 3

Native operators: For operators that are developed in general purpose programming languages, such as C++ and Java, the managed state API is provided as native interfaces. Operator developers are responsible for using these APIs in their native code to manage partitioned state, providing a `partitionBy` parameter, and specifying meta information about their operator (via the operator model [13]) to indicate that it is partitioned stateful. Note that the application developers do not need to know about these details.

APPENDIX B
MICRO-BENCHMARKS

For the micro-benchmarks, we use a synthetic application that has a single parallel region with a single operator. This operator can be configured as stateless or partitioned stateful. For the latter case the number of unique partitioning attribute values present in the workload is set to 1000 unless stated otherwise. The operator performs multiple integer multiplications per tuple. We scale the number of multiplications to adjust the amount of work performed per tuple, in order to explore both ends of the spectrum. When there is little work per tuple, scalability is more difficult to achieve because the parallelization overhead becomes significant compared to the actual work. When there is more work per tuple, it is easier to achieve scalability as the additional runtime work performed on a per tuple basis becomes insignificant relative to the cost of the application logic.

B.1 Scalability: Accuracy and Overshoot

Fig. 9 plots the speedup compared to the sequential case as a function of the per tuple processing cost, for different numbers of parallel channels as well as for elastic parallelism. For this experiment, the parallel region is configured to be stateless. We make a few observations from the figure. First, for high per tuple processing costs, linear scalability is achieved for up to 16 channels. For 32 channels, the speedup achieved is less than the ideal (around 21×). This is because the 4 machines are fully utilized at this time (8 cores per host). Second, we observe that for high per tuple processing costs, elastic auto-parallelization is able to achieve speedup that is close to the best achieved by the fixed number of channels. Finally, we observe that for low per tuple processing costs, there is overhead compared to the sequential case, both for fixed number of channels and for elastic auto-parallelization. The overhead of elastic auto-parallelization is less than that of 32 channels but more than that of 16 channels or less. Naturally, such overheads are more pronounced for lower per tuple processing costs.

Fig. 10 plots the number of channels the control algorithm settles down to as a function of the per tuple processing cost. As expected, the number of channels stays flat until the per tuple processing cost is high enough to make additional parallelism beneficial. After that point, it increases as the per tuple cost increases, and flattens again when the maximum number of channels is reached.

Fig. 11 again plots the speedup compared to the sequential case as a function of the per tuple processing cost, for different numbers of parallel channels as well as for elastic parallelism. This time, the parallel region is configured to be stateful. Overall, we observe similar trends as in Fig. 9. However, there are a number of differences. First, the speedup achieved with more than one channel as well as elastic parallelization are both higher than 1× for all per tuple processing costs. This is because a stateful parallel region is more costly to process even for the sequential case, since for each tuple several operations are performed to locate and update its state from the `map` data structure. Second, we observe that the speedup obtained by the elastic approach is slightly worse than the speedup achieved by the most effective fixed approach for a given configuration. This is due to the additional overheads of stateful elastic parallelism, such as the cost of consistent hashing at the splitter and the overhead of managed state APIs.

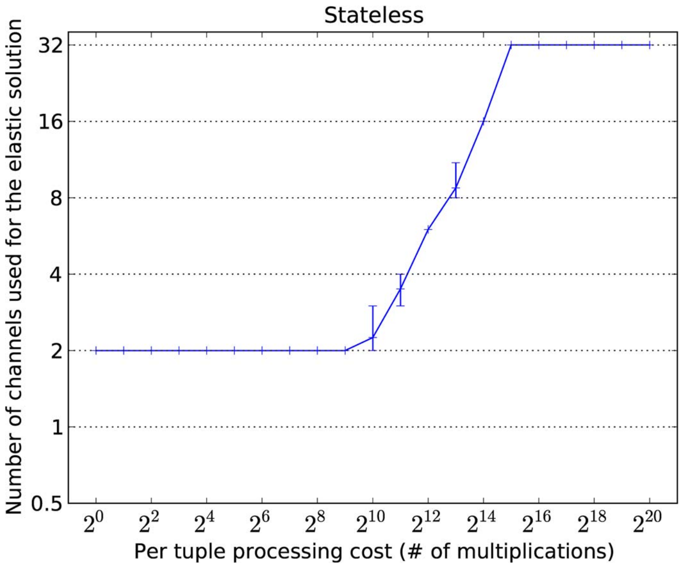


Fig. 10. Number of channels with increasing cost.

Fig. 12 plots the number of channels the control algorithm settles down to as a function of the per tuple processing cost. The results are similar to those in Fig. 10, except that the number of channels starts from a higher value for the smallest per tuple processing cost, as partitioned stateful parallel regions have additional per tuple processing costs.

B.2 Adaptation: Settling and Stability

Fig. 13 plots the available load (using the left y-axis) and the number of channels used by the elastic auto-parallel scheme (using the right y-axis) as a function of time. For this experiment the available load follows a sine wave with a mean value of 5000 tuples/second, amplitude of 2500 tuples/second, and a period of 500 seconds. From Fig. 13, we observe how the control algorithm reacts to change in

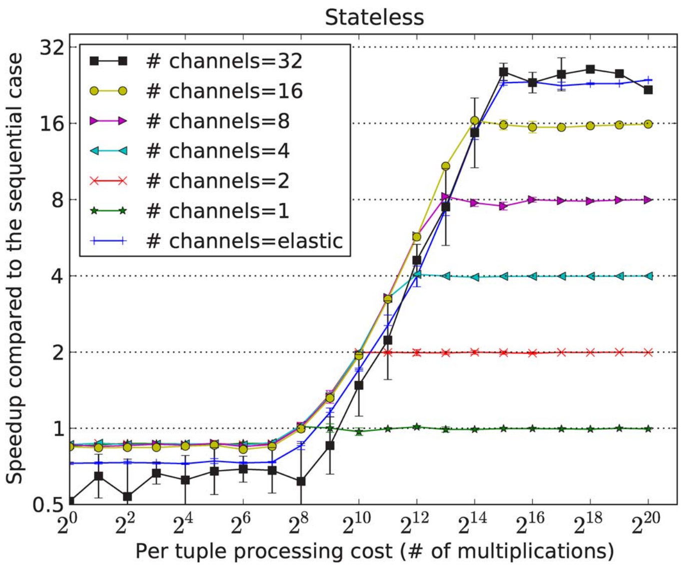


Fig. 9. Scalability with increasing tuple cost.

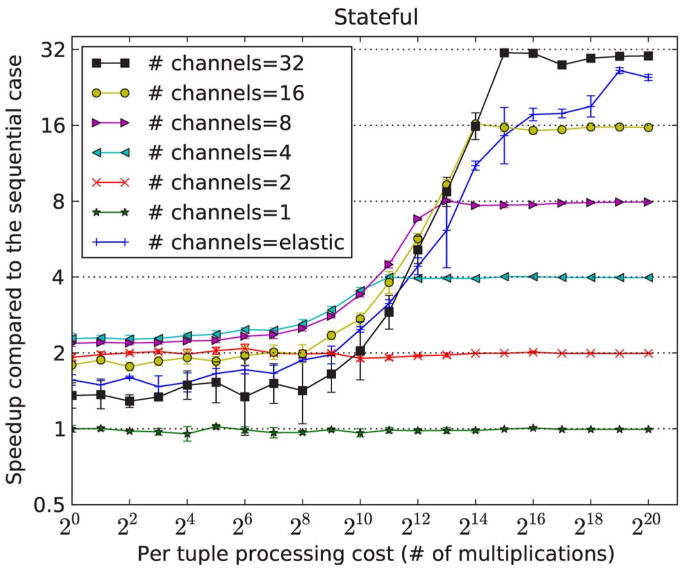


Fig. 11. Scalability with increasing tuple cost.

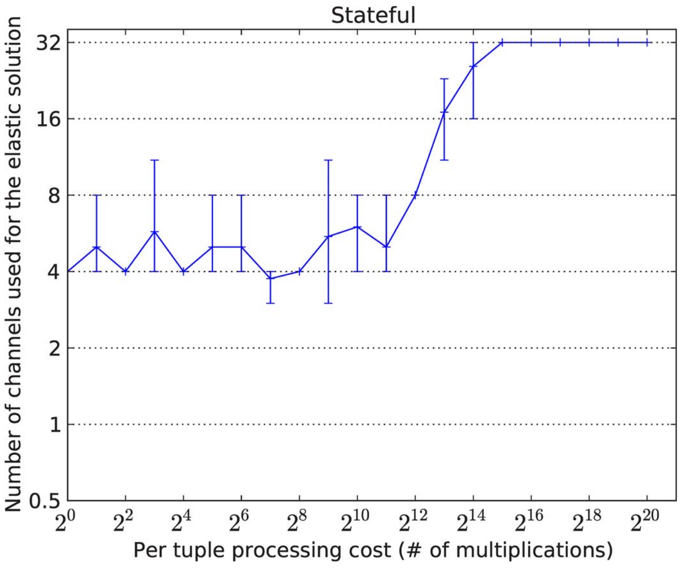


Fig. 12. Number of channels with increasing cost.

the load. The number of channels used follows the load availability, avoiding overshoot by decreasing the number of channels when load availability decreases and achieving good accuracy by increasing the number of channels when the load availability increases. Fig. 14 plots the available load and the handled load as a function of time, for the same experiment. This figure helps in understanding the effectiveness of the solution with respect to throughput. When the lines corresponding to the available load and handled load overlap, it indicates that the system is able to handle all of the load available. Yet, we need to check Fig. 13 to ensure that there is no overshoot, as all of the available load can be handled with more than the ideal number of channels (which cannot be told from Fig. 14 alone). Fig. 14 shows that most of the time we are able to handle all of the available load. During startup (which uses rapid scaling) as well as during times of adaptation, there are small periods of time during which some of the load is not handled. Nevertheless, the system quickly locates an effective operating point.

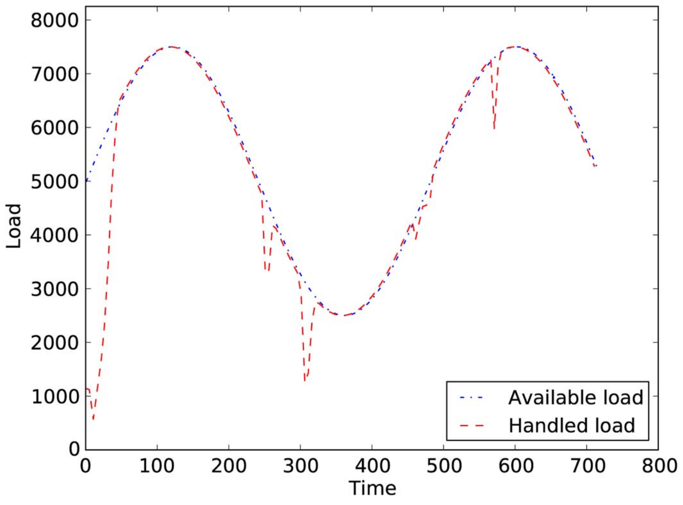


Fig. 14. Handled load vs. available load.

Figs. 15 and 16 plot similar results, but this time for a workload that follows a step function. The available workload starts with 5000 tuples/sec, after 250 seconds goes up to 10000 tuples/sec, after 100 seconds goes down to 2000 tuples/sec, and after 150 seconds goes back up to 5000 tuples/sec and repeats itself. Different than the sine workload, which has smooth changes in the workload, the step workload has sharp changes. Still, the control algorithm is able to adjust the number of channels based on workload availability and handle all of the load most of the time. Overall, the control algorithm is resilient to the workload characteristics.

B.3 The Congestion Index Threshold

The congestion index threshold impacts the adaptation behavior of the system. If the threshold is too high, then the system will not increase the number of channels even when additional workload can be handled with more channels. As a result, the accuracy will suffer. In general, smaller values for the threshold are safer. When the threshold is lower than ideal, then the system will overreact by increasing the number of channels when it is not really

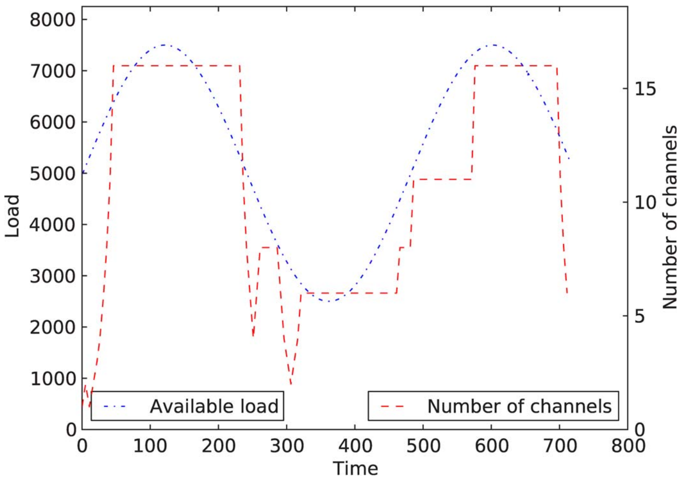


Fig. 13. Number of channels with varying load.

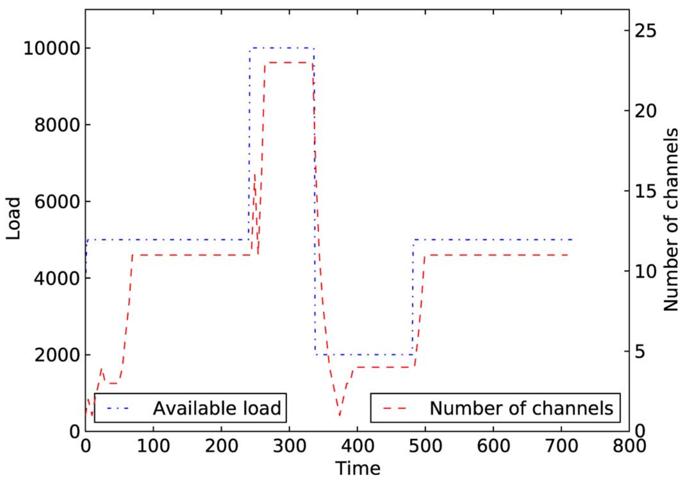


Fig. 15. Number of channels with varying load.

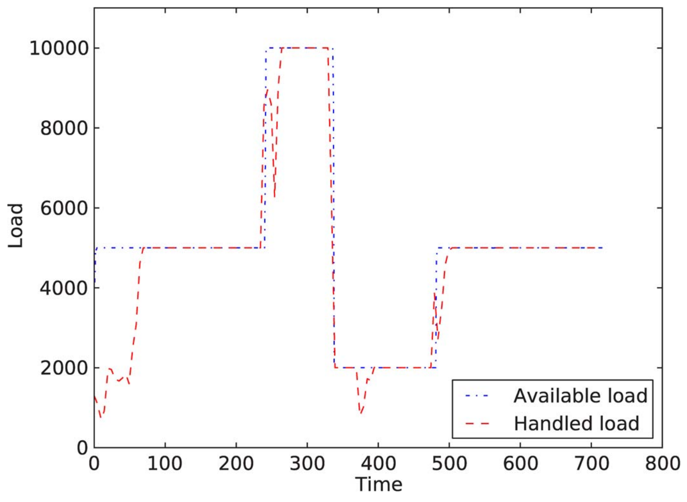


Fig. 16. Handled load vs. available load.

needed. This error, however, will be corrected after observing that increasing the number of channels is not helping the throughput. If the threshold is too low such that noise in a non-bottlenecked system is labeled as congestion, then the system will overshoot since it won't ever reduce the number of channels used. In our experiments, we found the range [0.01, 0.3] to be appropriate for the threshold.

Fig. 17 shows the behavior of the system when the threshold is set to 0 (too low). In this case, we observe that even though the load is handled properly, the control algorithm overshoots the target. It never reduces the number of channels.

Fig. 18 shows the behavior of the system when the threshold is set to 0.9 (too high). In this case, we see that the number of channels does not reach the required level and the accuracy suffers, as the available workload is often not handled by the system. The spikes in the handled load are due to the fine-grained throughput measurements that reach high values when space opens in the transport buffers every now and then.

B.4 Migration Cost

Fig. 19 plots the time it takes to perform the migration as a function of the number of channels, for different numbers of unique keys per partitioning attribute. This time includes synchronization, saving state that will transfer out, and loading state that will transfer in, for all operators involved. We observe that the migration time is linear in the number of keys. Furthermore, the migration cost decreases with the increasing number of channels. Thanks to the consistent hash, most data can remain on the host it was on at a different operating point, and only a small fraction of the data has to move. However, eventually the migration time flattens. This is because as the number of migrated data items decreases, the overhead of the migration protocol dominates the migration cost.

Migration cost is a concern only in highly unstable workloads. If the workload is relatively stable, initial costs for state migration will be amortized after the system settles down.

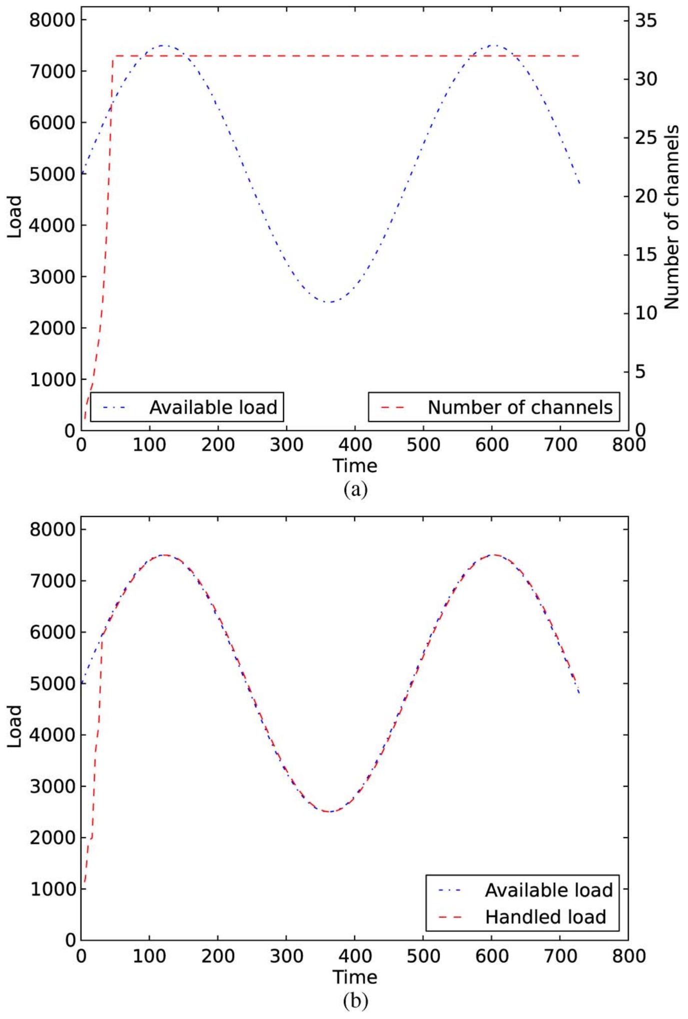


Fig. 17. Adaptation with a congestion index threshold of 0.

Appendix C Other Related Work

C.1 Map/Reduce Systems

Elastic auto-parallelization has been applied to Map/Reduce [9] systems as well (see [3] and [20]). It is important to note that stream processing applications often have ordering requirements that are absent in many commutative and associative workloads of Map/Reduce systems. Due to sequential segments that form bottlenecks, most streaming applications do not scale linearly with the number of hosts. As a result, it is important to apply runtime profitability analysis to determine the right level of parallelism for stream processing applications.

StreamMine uses active replication at no cost [20] to provide fault tolerance as well as a limited form of elasticity for streaming Map/Reduce applications—a variation of Map/Reduce where multiple partitioned stateful stages are connected to each other. It shifts processing resources to the active replicas during load spikes and falls back to state synchronization between the active and backup replicas when the latter's processing queues are full. The details as to how state synchronization is performed for stateful operators is not discussed in [20].

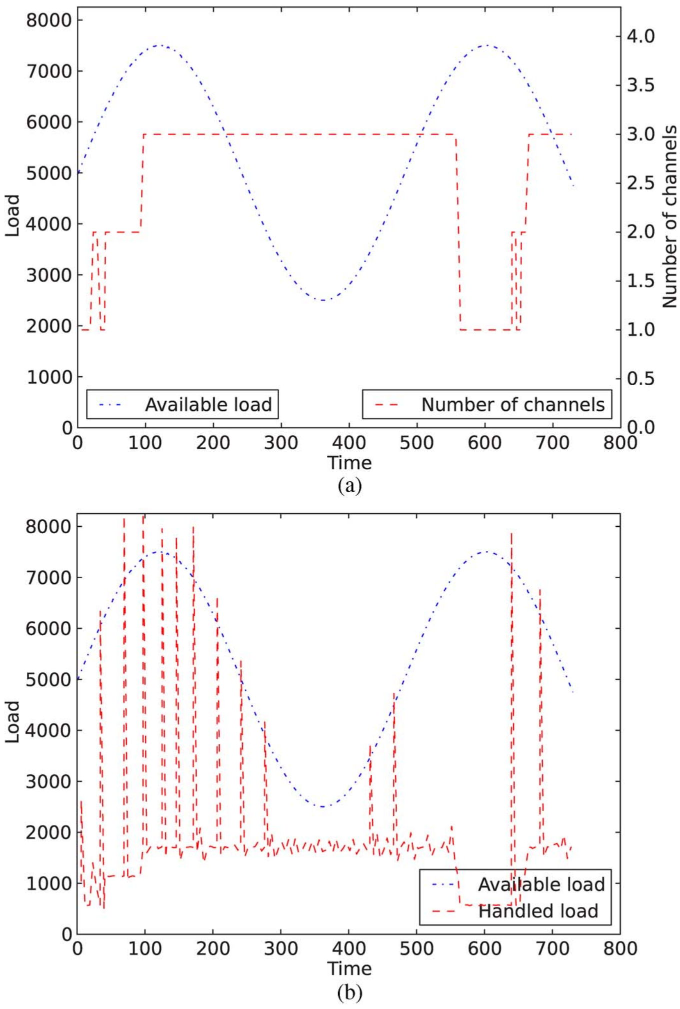


Fig. 18. Adaptation with a congestion index threshold of 0.9.

The Flood [3] system provides elasticity for Map/Reduce-like streaming systems where the source data streams are received from multiple clients. The approach is based on allocating additional VMs to handle increases in the number of clients or the volume of source data. However, this work does not recognize the need for state migration to achieve elasticity for stateful operators, let alone explain how such a state migration would work.

Many other systems that generalize the Map/Reduce paradigm to arbitrary DAGs, such as Nephele [2], Hyracks [7], and Dryad [17] exist as well. While such systems can stream intermediate results between certain stages, they are not designed to handle continuous data sources. Sequential and order-sensitive application segments, which often limit the scalability of streaming applications and make profitability analysis critical, do not exist in these systems. Furthermore, none of these systems perform elastic adaptation of the degree of parallelism involving state migration.

C.2 Other Systems

Flextream [15] is a dynamic auto-parallelization scheme for the StreamIt [11] system—a stream processing framework targeted at signal processing applications with strong emphasis on *static scheduling*. Flextream introduces dynamic

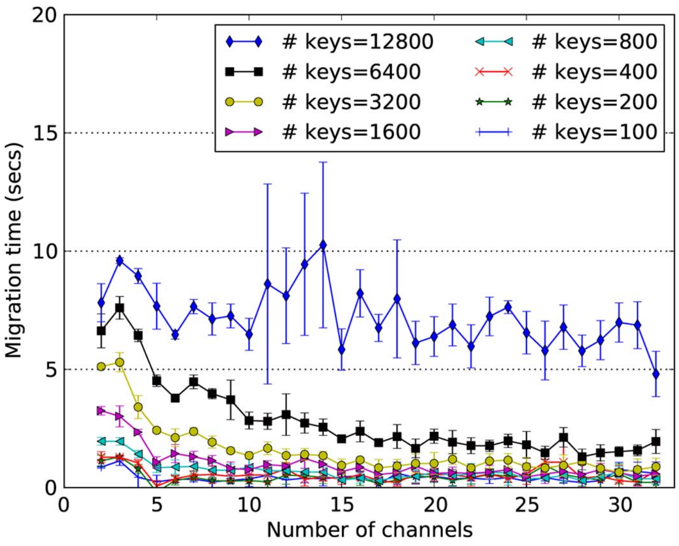


Fig. 19. Migration cost.

compilation techniques to adjust the level of parallelism used in order to adapt to changes in resource availability. However, multi-host elasticity and stateful operators are not addressed.

The feed-back directed pipeline parallelism approach of [29] performs dynamic adaptation for pipelines of parallelizable stages. It describes an online controller that decides the degree of parallelism for each stage. However, the approach is limited to shared memory and does not handle selective stages.

The SEDA architecture [31] employs control mechanisms for automatic tuning of parallelism in event-driven stage pipelines. SEDA is targeted at building highly concurrent Internet services and cannot be applied directly to distributed stream processing systems (issues such as stateful operators, ordering, and cross-host parallelism are not addressed). Yet it is similar to our solution in its application of control mechanisms to perform run-time parallelization tuning.

Finally, process migration is a well studied topic in the area of distributed systems [21]. In this work, we do not migrate processes, but instead perform partial state migration to re-distributed work to a new host or over remaining hosts.

References

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, ‘‘The Design of the Borealis Stream Processing Engine,’’ in *Proc. CIDR*, 2005, pp. 277-289.
[2] A. Alexandrov, D. Battre, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke, ‘‘Massively Parallel Data Analysis with PACTs on Nephele,’’ in *Proc. VLDB*, 2010, pp. 1625-1628.
[3] D. Alves, P. Bizarro, and P. Marques, ‘‘Flood: Elastic Streaming MapReduce,’’ in *Proc. Int'l Conf. ACM DEBS*, 2010, pp. 113-114.
[4] H. Andrade, B. Gedik, K.-L. Wu, and P.S. Yu, ‘‘Processing High Data Rate Streams in System S,’’ *J. Parallel Distrib. Comput.*, vol. 71, no. 2, pp. 145-156, Feb. 2011.
[5] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom, ‘‘STREAM: The Stanford Stream Data Manager,’’ *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 1-8, 2003.

[6] R. Avnurand and J.M. Hellerstein, ''Eddies: Continuously Adaptive Query Processing,'' in *Proc. Int'l Conf. ACM SIGMOD*, 2000, pp. 261-272.

[7] V.R. Borkar, M.J. Carey, R. Grover, N. Onose, and R. Vernica, ''Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing,'' in *Proc. IEEE ICDE*, 2011, pp. 1151-1162.

[8] S. Brin and L. Page, ''The Anatomy of a Large-Scale Hypertextual Web Search Engine,'' *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107-117, Apr. 1998.

[9] J. Dean and S. Ghemawat, ''MapReduce: A Flexible Data Processing Tool,'' *Commun. ACM*, vol. 53, no. 1, pp. 72-77, Jan. 2010.

[10] A. Deshpande, Z.G. Ives, and V. Raman, ''Adaptive Query Processing,'' *Found. Trends Databases*, vol. 1, no. 1, pp. 1-140, Jan. 2007.

[11] M.I. Gordon, W. Thies, and S. Amarasinghe, ''Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs,'' in *Proc. Int'l Conf. ASPLOS*, 2006, pp. 151-162.

[12] J.L. Hellerstein, Y. Diao, S. Parekh, and D.M. Tilbury, *Feedback Control of Computing Systems*. Hoboken, NJ, USA: Wiley, 2004.

[13] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu, ''IBM Streams Processing Language: Analyzing Big Data in Motion,'' *IBM J. Res. Develop.*, vol. 57, no. 3/4, pp. 7:1-7:11, May-July 2013.

[14] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, ''A Catalog of Stream Processing Optimizations,'' IBM, New York, NY, USA, Research Report RC25215, 2011.

[15] A.H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, ''Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures,'' in *Proc. Int'l Conf. PACT*, 2009, pp. 214-223.

[16] IBM InfoSphere Streams. [Online]. Available: http://www-01.ibm.com/software/data/infosphere/streams/Retrieved, Nov. 2012.

[17] M. Isard, M.B.Y. Yu, A. Birrell, and D. Fetterly, ''Dryad: Distributed Data-Parallel Program from Sequential Building Blocks,'' in *Proc. EuroSys*, 2007, pp. 59-72.

[18] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, ''Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core,'' in *Proc. Int'l Conf. ACM SIGMOD*, 2006, pp. 431-442.

[19] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, ''Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web,'' in *Proc. Int'l ACM STOC*, 1997, pp. 654-663.

[20] A. Martin, C. Fetzer, and A. Brito, ''Active Replication at (Almost) No Cost,'' in *Proc. Int'l SRDS*, 2011, pp. 21-30.

[21] D.S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, ''Process Migration,'' *ACM Comput. Surveys*, vol. 32, no. 3, pp. 241-299, Sept. 2000.

[22] N.W. Paton, J.B. Chavez, M. Chen, V. Raman, G. Swart, I. Narang, D.M. Yellin, and A.A.A. Fernandes, ''Autonomic Query Parallelization Using Non-Dedicated Computers: An Evaluation of Adaptivity Options,'' *Very Large Data Bases J.*, vol. 18, no. 1, pp. 119-140, Jan. 2009.

[23] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, ''Elastic Scaling of Data Parallel Operators in Stream Processing,'' in *Proc. IEEE IPDPS*, 2009, pp. 1-12.

[24] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, ''Automatic Exploitation of Data Parallelism in Stateful Streaming Applications,'' in *Proc. Int'l Conf. PACT*, 2012, pp. 53-64.

[25] M.A. Shah, J.M. Hellerstein, S. Chandrasekaran, and M.J. Franklin, ''Flux: An Adaptive Partitioning Operator for Continuous Query Systems,'' in *Proc. IEEE ICDE*, 2003, pp. 25-36.

[26] M. Stonebraker, U. Çetintemel, and S.B. Zdonik, ''The 8 Requirements of Real-Time Stream Processing,'' *SIGMOD Rec.*, vol. 34, no. 4, pp. 42-47, Dec. 2005.

[27] Storm Project, Oct. 2013. [Online]. Available: http://storm-project.net/

[28] StreamBase, May 2012. [Online]. Available: http://www.streambase.com

[29] M.A. Suleman, M.K. Qureshi, Khubaib, and Y.N. Patt, ''Feedback-Directed Pipeline Parallelism,'' in *Proc. Int'l Conf. PACT*, 2010, pp. 147-156.

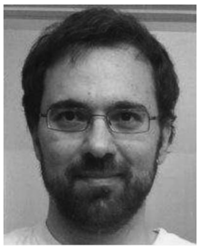
[30] S. Viglas, J.F. Naughton, and J. Burger, ''Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources,'' in *Proc. VLDB Conf.*, 2003, pp. 285-296.

[31] M. Welsh, D. Culler, and E. Brewer, ''SEDA: An Architecture for Well-Conditioned, Scalable Internet Services,'' in *Proc. Symp. OSDI*, 2001, pp. 230-243.

[32] S. Wu, V. Kumar, K.-L. Wu, and B.C. Ooi, ''Parallelizing Stateful Operators in a DSPS: How, Should You and How Much?'' *Proc. Int'l Conf. ACM DEBS*, 2012, pp. 278-289.

[33] Z. Xu, M. Hirzel, G. Rothermel, and K.-L. Wu, ''Testing Properties of Dataflow Program Operators,'' in *Proc. Conf. ASE*, 2013, pp. 103-113.

[34] S4 Distributed Stream Computing Platform, Oct. 2013. [Online]. Available: http://incubator.apache.org/s4



Buğra Gedik received the PhD degree in computer science from Georgia Tech., Atlanta, GA. He is currently with the Computer Engineering Department, Ihsan Doğramacı Bilkent University, Turkey. Prior to that, he worked as a Research Staff Member at the IBM T. J. Watson Research Center. His research interests include stream computing, distributed systems, and data bases.

Scott Schneider received the PhD degree in computer science from Virginia Tech, Blacksburg, VA and the MSc degree in computer science from William & Mary, Williamsburg, VA. He is currently a Research Staff Member at the IBM T.J. Watson Research Center. His research focuses on improving the programmability and performance of distributed streaming systems, with a focus on exploiting parallelism.

Martin Hirzel received the PhD degree in computer science from the University of Colorado at Boulder. He is currently a Research Staff Member and a Manager at the IBM T.J. Watson Research Center. He manages the Data Languages Science group at IBM. His research focuses on stream computing, programming languages, and compilers.

Kun-Lung Wu received the MSc and PhD degrees in computer science from the University of Illinois at Urbana-Champaign. he is currently the manager of the Data-intensive Systems and Analytics Group at the IBM T.J. Watson Research Center. His research interests include stream computing, big data analytics, and data bases.