Automating Multi-level Performance Elastic Components for IBM Streams

Xiang Ni
IBM Research AI
xiang.ni@ibm.com

Scott Schneider
IBM Research AI
scott.a.s@us.ibm.com

Raju Pavuluri
IBM Research AI
pavuluri@us.ibm.com

Jonathan Kaus
IBM Watson Data Platform
jlkaus@us.ibm.com

Kun-Lung Wu
IBM Research AI
klwu@us.ibm.com

Abstract

Streaming applications exhibit abundant opportunities for pipeline parallelism, data parallelism and task parallelism. Prior work in IBM Streams introduced an elastic threading model that sought the best performance by automatically tuning the number of threads. In this paper, we introduce the ability to automatically discover where that threading model is profitable. However this introduces a new challenge: we have separate performance elastic mechanisms that are designed with different objectives, leading to potential negative interactions and unintended performance degradation. We present our experiences in overcoming these challenges by showing how to coordinate separate but interfering elasticity mechanisms to maxmize performance gains with stable and fast parallelism exploitation. We first describe an elastic performance mechanism that automatically adapts different threading models to different regions of an application. We then show a coherent ecosystem for coordinating this threading model elasticty with thread count elasticity. This system is an online, stable multi-level elastic coordination scheme that adapts different regions of a streaming application to different threading models and number of threads. We implemented this multi-level coordination scheme in IBM Streams and demonstrated that it (a) scales to over a hundred threads; (b) can improve performance by an order of magnitude on two different processor architectures when an application can benefit from multiple threading models; and (c) achieves performance comparable to hand-optimized applications but with much fewer threads.

Middleware '19, December 8–13, 2019, Davis, CA, USA

ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3361544

CCS Concepts • **Information systems → Stream management**.

Keywords Stream processing, elastic scheduling, runtime

1 Introduction

Online stream processing has emerged to meet the demands of processing large amounts of data with low latency and high throughput. The languages and frameworks for stream processing are parallel and distributed, and use a dataflow programming model to abstract the development for parallel and distributed systems [21, 5, 1, 18, 2]. However, despite these abstractions, tuning the performance of application deployments is still an intensive task for developers and administrators. Further, streaming application deployments are increasingly moving to the cloud. In an environment where the physical hardware that the streaming application runs on is unknown or can potentially change across deployments, intensive performance tuning becomes increasingly difficult.

In the context of performance optimization, stream processing applications have a property that can be exploited: they tend to be long-running. Because of the nature of the problems they solve—processing large amounts of continually arriving data—a typical application deployment is live for weeks or months. Long-running applications are amenable to online, automatic performance-based adaptation because adaptation phases are easily amortized. Such application adaptations can help solve the difficulties of optimizing the performance of complicated parallel and distributed applications deployed to unknown hardware.

IBM Streams [12, 13] is a parallel and distributed streaming platform used in production in dozens of companies in industries including aviation, medicine, transportation, telecommunication and banking. The programming language for IBM Streams is SPL [11, 19], which is a dataflow language with primary abstractions for streams, operators and tuples: operators receive and emit tuples on streams of data.

IBM Streams 4.2 introduced a dynamic threading model with thread count elasticity to the SPL runtime [20]. By dynamically adjusting the number of threads at runtime, thread count elasticity allowed the SPL runtime to automatically

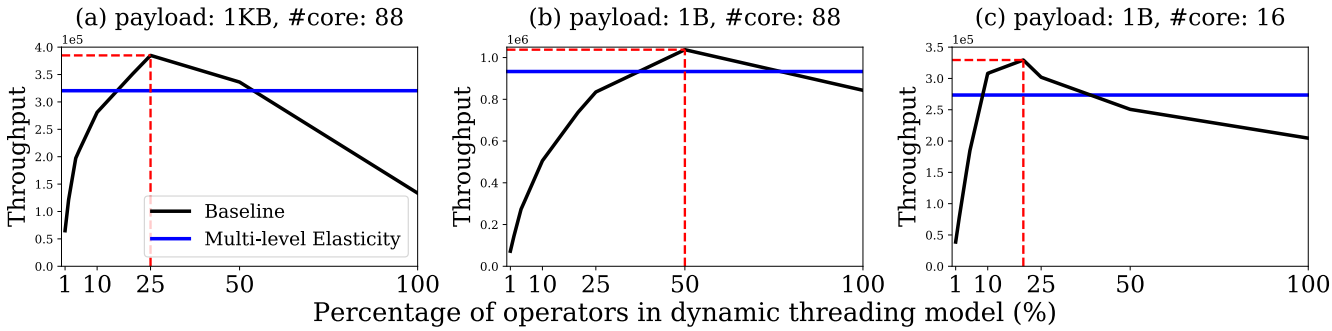

Figure 1. Black lines are the throughput of a chain of 100 operators with varying percentage of operators using the dynamic threading model. Blue lines are the throughput achieved by the proposed framework, which reaches good performance with automatic threading model adjustment and thread count adjustment. The x-axis of varing percentage of operators only applies to the black lines.

scale and take advantage of multicore systems. However, it achieved this scalability through a heavy-handed mechanism: it introduced scheduler queues in front of every operator. As a result, threads can freely execute any operator by pulling tuples from the corresponding queue. The use of schedulers queues incurs two major overheads: copy overhead due to the fact that tuples in SPL are statically allocated and synchronization overhead when tuples are passed to downstream operators. As the operator count scales, an increasing list of scheduler queues means that each thread has to spend longer time in finding work. Programmers can manually annotate regions that should be executed by a single thread to avoid such overheads, but that removes the benefit of automation.

In Figure 1, we show the throughput of a 100 operator pipeline while varying the percentage of operators executed under the dynamic threading model, i.e. operators with a scheduler queue in front of them. Operators not under the dynamic threading model are executed by the threads from upstream operators. The workload of each operator is 100 FLOPs per tuple. We vary the tuple payload from 1B to 1KB and the available resource from 16 cores to 88 cores. All throughputs are measured after thread elasticity has settled on the best number of threads for that configuration. The takeaway from Figure 1 is that the best throughput is not achieved when all operators are executed under the dynamic threading model, and that the optimal configuration varies. Tuple copying and thread synchronization costs dictate that some sections of the application should be single-threaded. The existing thread count elasticity solves a single dimensional problem, but these experiments illustrate that there is an additional dimension—threading model elasticity.

The addition of threading model elasticity means that the streaming runtime has two separate but interfering performance elastic components making online adjustments. These performance control algorithms run at separate intervals, modify different components of the runtime, and do not explicitly refer to each other. But they are interfering because the modifications made by one component affects the decisions made by the other. This paper presents an automatic solution that coordinates the adjustment of the threading model of individual operators with the existing thread count performance adapatation component. Our coordinating methodology finds a scheduling solution for multiple performance elastic components that improves performance with SASO guarantees [10], which means it provides *stability* (no oscillation between adjustments), achieves good *accuracy* (find the threading model and thread count that maximizes throughput), has *short settling time* (reaches a stable configuration quickly), and *avoids overshoot* (does not use more threads than necessary).

The specific contributions of this paper are:

1. A multi-level performance elastic framework in production use to coordinate the threading model choice at operator level in tandem with the adjustment of thread count.
2. A control algorithm that uses runtime metrics and local control to achieve SASO properties in order to quickly adapt to varying workload with performance guarantee.
3. Empirical evaluation of the proposed framework on two processor architectures using benchmarks and applications that demonstrates scalability to over a hundred threads, better resource utilization, and more than 10× throughput gains in some cases.

2 Background

The three primary abstractions in SPL, the language for programming in IBM Streams, are *operators*, *tuples* and *streams*. Operators are the primary actors: they are event-based and execute when they receive tuples on their input ports. Tuples are structured data items with strongly-typed attributes. There are no restrictions on the kind of logic that executes inside an operator, except that it can only natively access state local to the operator. Operators can produce tuples, which are submitted to their output ports. The input and output ports of operators are connected by streams.

```
composite WikiWordCount {
graph
  stream<rstring page> WikiPages = HTTPGetStream() {
    param url: "https://stream.wikimedia.org/v2/stream/recentchange";
  }
  @parallel(width=5)
  stream<rstring word> Words = Custom(WikiPages) {
    logic onTuple WikiPage: {
      list<rstring> words = tokenize(WikiPages.page, " ", false);
      for (rstring w in words) {
        submit({word=w}, Words);
      }
    }
  }
  @parallel(width=10, partitionBy=[{port=Words, attributes=[word]}])
  stream<rstring word, int32 count> Counts = Aggregate(Words) {
    window Words: sliding, time(60), time(1), partitioned;
    param partitionBy: word;
    output Counts: word=Any(), count=Count();
  }
  () as Published = WebSocketSend(Counts) { param port: 8082; }
}
```

Figure 2. Streams Processing Language (SPL) example.

At job submission, operators are divided among PEs (processing elements). PEs with connected operators communicate over the network. Inside of a PE, connected operators communicate either through function calls or queues. PEs are how Streams takes advantage of multiple hosts, and threads inside of a PE are how Streams takes advantage of multicore hosts. This paper is only concerned with the execution inside of a single PE, although all PEs in a job independently use the proposed work to maximize their performance.

Figure 2 is an example SPL composite operator that retrieves the latest changes to Wikipedia and publishes histograms of the word counts to a websocket. The `HTTPGetStream` operator from the `streamsx.inet` toolkit [14] continually retrieves the latest changes from Wikipedia. These changes are streamed to an operator with custom logic that tokenizes the page, and submits a new tuple for each word. As each page is independent, we use the `@parallel` annotation to use 5 data-parallel copies for this task. An `Aggregate` operator counts how many instances of a word it sees in every 60 second window, updating every second. Finally, it publishes the results through `WebSocketSend`, also from the `streamsx.inet` toolkit, viewable by any service that connects to that port.

While this example is a toy compared to production applications, it demonstrates several key properties common in real applications: a variety of tuple types and sizes; many opportunities for data and pipeline parallelism; and a variety in the computational cost of operators.

2.1 Threading models and elasticity

Streams 4.2 introduced threading models and thread count elasticity to the SPL runtime, which enabled PEs to launch multiple threads to execute operators. The *manual* threading model uses the threads already introduced by the programmers or operators. It is so-called because programmers must manually introduce threads between operators at compile-time to add more parallelism. The *dynamic* threading model injects scheduler queues between each operator, and the SPL runtime maintains a set of scheduler threads that can execute any operator. A thread count elasticity algorithm monitors total throughput across all operators and dynamically change the number of threads to maximize that throughput [20].

This paper is concerned with two types of threads in Streams. *Operator threads* drive the execution of source operators. *Scheduler threads* are used by the dynamic threading model. Scheduler threads obtain tuples from scheduler queues associated with the input ports of an operator and execute the operator. When any thread (scheduler or operator thread) encounters a scheduler queue, it pushes its current tuple into that queue and continues executing from its source operator. Scheduler threads are not bound to a specific input port or operator. They incur synchronization overhead—as any scheduler thread can execute any operator using the dynamic threading model—but they are adaptable and enable more parallelism.

This paper presents a solution that, in threading model terms, automatically partitions the application into dynamic and manual threading model regions.

3 Design and Implementation

We first present our new elastic algorithm to adjust the threading model when the thread count is fixed, and then extend it to adjust both the threading model and thread count coherently. This algorithm uses the *operator cost metric*, which is computed during runtime with low overhead.

The operator cost metric is an indicator of the relative computation workload of operators. To compute it, we register a runtime level per-thread *state* variable for each thread in the system, which is set to the corresponding operator index when threads enter the processing logic of that operator. A profiler thread wakes up every profiling period to take a snapshot of the *state* of all the actively running threads. It maintains counters for each operator and increments the counter by the number of times that operator appears in the snapshot. This counter directly correlates with the relative operator cost and is reported as the operator cost metric.

3.1 Threading model elasticity

Threading model elasticity aims to automatically select the threading model, dynamic or manual, for each operator in order to improve performance. Given N operators, the exploration space contains 2^N configurations, and thus exhaustive search is not a scalable solution. The following two observations allow us to reduce the search space to linear:

(O1) If the cost metric of an operator is relatively high, there are higher chances for it to benefit from the use of the dynamic threading model, i.e. it is more likely that additional parallelism amortizes scheduling and queuing overheads.

(O2) If there is performance improvement when an operator uses the dynamic threading model, we can expect similar

performance gain if other operators that have similar cost metrics are executed with the dynamic threading model.

Our algorithm begins with all operators under the manual threading model, i.e. there are no scheduler queues. Per (O1), the control algorithm prioritizes to select the dynamic threading model for computationally heavy operators, and terminates the exploration when turning more operators to use the dynamic threading model no longer improves performance. Per (O2), we perform logarithmic binning by dividing operators into profiling groups. Rather than testing the threading model choice with each individual operator, we now set the granularity of adjustment at the level of this group of operators. We start from the group with the highest relative cost, say G_h. If there is performance improvement with the use of the dynamic threading model for every operator in G_h, we move on examine the group with the next-highest relative cost among the remaining groups (G_{h-1}). If there is performance degradation due to the change in threading model choice for G_h, we further break down that group and study if performance improvement can be achieved with part of the group choosing the dynamic threading model (described next). Both (O1) and (O2) help to satisfy the *settling time* property of SASO.

Within a group, the elastic control algorithm performs a binary search to select the right set of operators that should use the dynamic threading model, guided by the performance trend learnt through exploration. Broadly speaking, the threading model elasticity algorithm tests new configurations in the direction that has the possibility of providing higher performance based on the data we have collected so far. For example, if we have data from fewer operators choosing dynamic threading model and we have not explored using more operators, and the data suggest that throughput increases with the operator count, the logical thing to do is to select the dynamic threading model for more operators.

Figure 3 and Figure 4 show the step-by-step view and algorithm of the threading model elastic component. In Figure 4, `currCount` refers to the number of operators choosing dynamic threading in the latest adjustment and `binSize` refers to the number of operators in the current group. The following rules are used to elastically partition operators into different threading model regions:

(R1) When there is an increasing performance trend compared to fewer operators using the dynamic threading model and we have not explored with more operators, add more operators in the dynamic threading model region (Figure 3(a)), line 8 in Figure 4).

(R2) If there is an increasing performance trend when operator count increases, increase the operator count (Figure 3(b)), line 8 in Figure 4).

(R3) When there is a decreasing performance trend compared to fewer operators using the dynamic threading model, decrease the operator count (Figure 3(c)), line 10 in Figure 4).

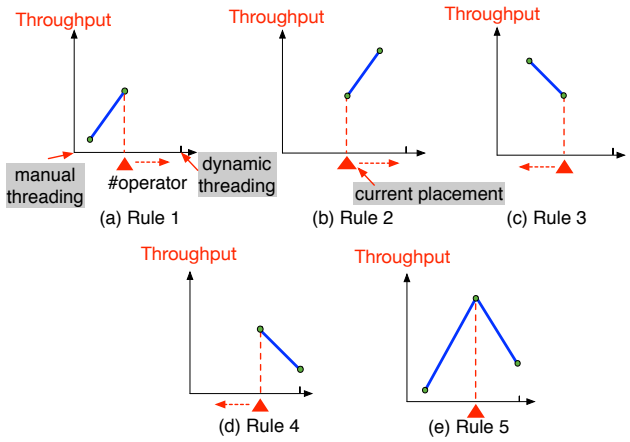

Figure 3. Steps to explore threading model choice; the blue line indicates the measured performance trends; the red triangle indicates the current number of operators that uses dynamic threading; the dotted arrow indicates the action for next step: whether to increase or decrease the number of operators for dynamic threading.

```
1  enum AdjustDecision{CONTINUE, STAY, CHANGE};
2
3  AdjustDecision threadingModelAdjustement() {
4    if (currCount == binSize - 1 && perfIncWithMore()) {
5      //continue the threading model adjustment with the next group
6      currGroup = nextGroup; threadingModelAdjustement();
7    } else if (perfIncWithMore()){
8      increaseCount(); return CONTINUE;
9    } else if (perfDecWithMore()){
10     decreaseCount(); return CONTINUE;
11   } else {
12     if (currCount == 0) return STAY;
13     else return CHANGE;
14   }
15 }
```

Figure 4. Elasticity algorithm for threading model elastic component.

(R4) When there is a decreasing performance trend with more operators and we have not explored with fewer operators, decrease the operator count (Figure 3(d), line 10 in Figure 4).

(R5) When there is an increasing performance trend compared to both fewer and more operators using the dynamic threading model, stop the search (Figure 3(e), line 12 or line 6 (when performance improvement of dynamic threading model applies to every operator in the current group, we move on to the adjustsment of next group) in Figure 4).

The current performance trend is composed by only two performance data points: the last adjustment and the current adjustment. Hence it is hard to tell where the optimal configuration may lie just through the performance trend. We explore the optimal configuration of threading model through adaptive adjustment. For example, in Figure 3(a), the optimal configuration may either lie on the left side of the current adjustment – indicating a performance increasing performance trend followed by decreasing performance

trend and the current adjustment actually falls on the downward trend, or it may lie on the right side of the current adjustment – indicating the current adjustment falls on the upward trend. However, as we proceed for the scenario in Figure 3(a), we will either end up with the case in Figure 3(a) again, which confirms that the optimal configuration lies on the right side of the current placement, or end up with Figure 3(c), which indicates that the optimal configuration lies on left side in Figure 3(a). Similar observations can be made about the scenario of Figure 3(d). As we make progress in this search, we limit the scope of the possible optimal configuration. Eventually, when the difference in operator count between two consecutive adjustments is short enough for us to establish the relationship in (R5), we stop the exploration.

Note that (R1) and (R2) satisfy the accuracy property of SASO by adding operators to the dynamic threading model region for potential performance gains if we observe an upward performance trend. (R3) and (R4) also provide the accuracy property in SASO by exploring in the reverse direction if a downward trend exists. All the rules from (R1) to (R5) satisfy the stability property in SASO: we do not oscillate between adjustments since observations from the past are remembered and represented as performance trends.

3.1.1 Discussion on implementation details

In our implementation, a dedicated *adaptation* thread is used to monitor the throughput changes periodically. The period between observation and adjustment should be long enough to have the change reflected in throughput and short enough to detect workload change. We use a period of 5 seconds and find that it works well for Streams applications. We have also experimented with the periods of 10s, 20s and 30s and have not observed significant performance impact due to the different period values within this range.

The observed performance change should be significant enough to differentiate from system noise. Hence, we introduce a sensitivity threshold, `SENS`. A smaller `SENS` value favors detecting changes while a larger `SENS` value favors stability. We choose the value of 0.05 in our implementation, meaning that we must observe at least a 5% performance difference before establishing a performance trend.

Exploring threading model changes within a profiling group presents a choice: given that N operators need to use the dynamic threading model, which N should it be? In our implementation, we choose an arbitrary set of N from within the group. It turns out such randomness not only provides the settling time property in SASO, but also incurs negligible disturbance to the system: reasonable good performance is achieved compared to manual tuning as shown in Figure 1 and little run-to-run variance as shown in Section 4. Limiting the scope of threading model adjustment within a group of operator does help reduce variance, since operators within a group have a similar cost metric.

3.2 Multi-level elasticity

So far, we have presented the design of the threading model performance elastic component. When integrating this feature to IBM Streams 4.3, the challenge was how to coordinate the threading model elastic component with the existing thread count elastic component [20]. If not careful, the potential incompatibilities between the adatptive components can lead to unstable poorly-tuned performance feedback loop. Our solution effectively and quickly explored the configuration space through an iterative refinement process by fixing one elastic component at a time while making adjustment for the other until no performance improvement can be gained.

Primary adjustment: The first design choice was the primary elastic adjustment, for which we explored two options:

1. Change in thread count: A thread count change triggers the search to find the locally optimal threading model configuration for that number of threads.
2. Change in threading model: Threading model changes trigger finding the locally optimal number of threads for the current threading model configuration.

We tried both options and adopted the first approach in our design for two reasons. First, we did not want to exhaust system resources. If the thread count adjustment was secondary, we would repeatedly increase the number of threads up to the point of performance degradation. Doing so was required in finding the optimal number of threads. Hence, the system would potentially be oversubscribed much more frequently during the adaptation period. In contrast, our choice favored the SASO property of avoiding overshoot.

Second, changes in thread count usually cause higher variation in performance than changes in threading models. Hence, if thread count adjustment was in the inner loop as the secondary adjustment, the performance impact of the outer loop threading model adjustment became less tractable, which in turn made it harder to apply consistent corrective adjustments to the threading model choices.

Adjustment direction: Our second design question was what should the starting conditions be? We could start with either:

1. The maximum number of available threads: Every operator would choose to use dynamic threading.
2. The minimum number of threads: No operator would choose dynamic threading.

The starting conditions determine the adjustment direction; starting with full parallelism means the algorithm will reduce it, and starting with no parallelism means the algorithm will introduce it. Initially in our design we adopted the first adjustment direction. However, when the maximum number of threads and queues were used (since every operator is under dynamic threading), it meant that the algorithm would reduce threads and take away queues from the least expensive operators. This performance difference was often

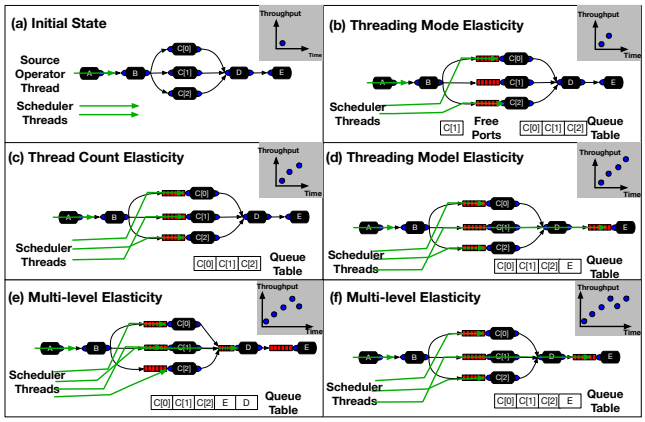

Figure 5. Steps involved in Multi-level elasticity.

indistinguishable from system noise, and thus the iterative refinement terminated earlier than it should. Choosing the second adjustment direction achieved better accuracy because it started by enabling parallelism with the most expensive operators, which provided a more reliable signal. It also had the benefit of being more likely to avoid system over-subscription. Both properties aligned with the SASO goals.

Interaction between threading model and thread count elasticity:

Figure 5 shows the multi-level elasticity in action. At the start (Figure 5(a)), no operator uses the dynamic threading model, hence the only source operator thread will execute all downstream operators. Two scheduler threads exist but remain idle since there are no scheduler queues for them to find work from. Next, the PE explores the threading model choices, which adds scheduler queues for operators $C[0]$, $C[1]$ and $C[2]$ (Figure 5(b)). As a result, the two scheduler threads are no longer idle and throughput improves. Next, if we have not reached the maximum number of threads allowed, thread count elasticity kicks in to further increase the thread count for better performance as shown in Figure 5(c). With more threads, another round of threading model elasticity places one more scheduler queue to advance the performance in Figure 5(d). At some point, further adjusting the number of threads or threading configuration could no longer improve the performance as can be seen in Figure 5(e). Hence, in Figure 5(f), the elasticity algorithm reverts the adjustment and stabilizes until the workload changes.

3.3 Optimizations to shorten the adaptation period

Figure 6 presents sample runs of multi-level elasticity with different sets of optimizations to demonstrate how they effect settling time. The x-axis shows the time into the experiments in seconds. The left y-axis is the throughput. The right y-axis shows the number of scheduler queues. The top x-axis is the number of threads for that window of time. The blue line shows the changes in the scheduler queues caused by threading model elasticity. The dotted and solid black lines shows the changes in throughput induced by the thread count elasticity and threading model elasticity, respectively. The graph used in all these experiments is a 500 operator pipeline with varying cost of 10,000, 100 and 1 FLOPs to process one tuple. The tuple payload is 1024B. Note that sometimes the converged throughput in Figure 6 appears to be lower than the maximum throughput: this is because the maximum throughput is observed when there is a sudden change in the number of threading queues, which causes the temporary peak in throughput.

Figure 6(a) uses what we have described so far. Adjustments to thread count trigger threading model elasticity. Due to these adjustments, the throughput keeps increasing and finally stabilizes after 1000 seconds. Note that both thread count and threading model elasticity contribute to improving the throughput as they are iteratively triggered. In Figure 6(a), we can also see that though threading model elasticity is triggered every time the thread count changes, the threading configuration may remain unchanged after exploration. For example, in Figure 6(a), after the window of time with 64 threads, the scheduler queue placement changes many times, but it eventually settles back to the level it started at. This observation motivates our first optimization to shorten the adaptation period: *learning from history*. As can be seen from Figure 6(a), at 96 threads, adapting the threading model configuration can no longer further improve the performance in comparison to the throughput achieved at 64 threads. Hence, the same queue placement works for both 64 and 96 threads. With this history information, when the thread count is decreased from 96 to 80, we can skip adjusting the threading model by approximating that the same configuration is best for any thread count between 64 and 96.

Learning from history: The essence of this optimization is to keep track of the thread range (N, M) that works well with the recent threading model adjustment, meaning from thread count N to thread count M, the optimal threading model configuration remains unchanged. Information about the past adjustment is stored. Inside each history record of threading model adjustment, we record the maximum and minimum number of threads that have worked well with this configuration.

When the thread count changes, we look into the record of the most recent queue placement. If the new thread count is within the thread range that has worked well with the current placement, we skip adjusting the threading model for now and continue to adapt the thread count. If the new thread count is above the upper bound of the thread range, we explore if using more scheduler queues can help improve performance. Otherwise if the thread count is below the lower bound of the thread range, we try to switch more operators to use manual threading.

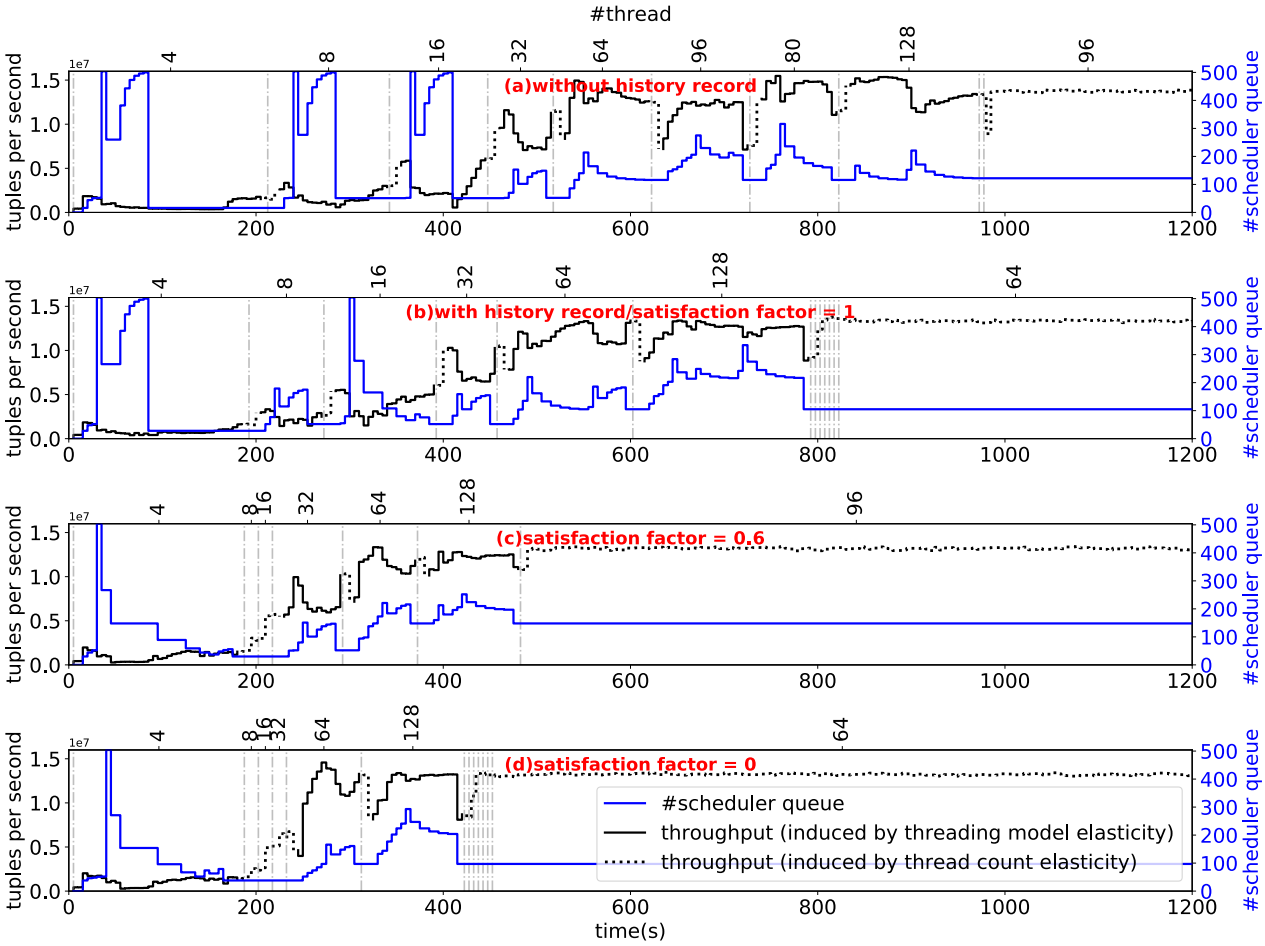

Figure 6. Effect of optimizations to shorten the adaptation period.

When no further improvement can be obtained by solely changing the threading model configuration, we switch back to the thread count elasticity phase. If there is any change in the threading model configuration, we update the adjustment history record to reflect it. Otherwise, we update the thread range for the existing queue placement.

By learning from history, in Figure 6(b) we are able to shorten the adaptation period by 20%. At around 800 seconds in Figure 6(b), there are several narrow dotted vertical lines which indicate that the thread count changed but queue adjustments were skipped.

How can we further shorten the adaptation period without sacrificing the performance? Another pattern can be observed in both Figure 6(a) and Figure 6(b) is that when the thread count increases to 16 and 32, the performance improvement is solely brought by changing the thread count. After exploring the threading model configuration, the queue placement remains unchanged. Hence when the thread count change alone already improves the performance significantly, it leaves little space for the threading model adjustment to play. Such observations lead to the second optimization to shorten the adaptation period: **satisfaction factor**.

Satisfaction factor (sf): The essence of this optimization is that if the primary adjustment (thread count) alone can improve the performance by a significant amount, the secondary adjustment (threading model) can be skipped unless the thread count alters again. We use the following condition to determine when to skip the secondary adjustment:

$$\left(\frac{\text{currThroughput}}{\text{prevThroughput}} - 1\right) > \texttt{sf} \left(\frac{\text{newThreadCount}}{\text{prevThreadCount}} - 1\right)$$

`sf` indicates the relative performance gain we expect compared to the increase in the thread count, and its value is between 0 and 1. When `sf` is closer to 0, we favor faster adaptation and we skip the queue adjustment more frequently. When `sf` is closer to 1, we favor better performance and perform the threading model adjustment more frequently.

In Figures 6(c, d), we present the scenarios with the satisfaction factor set to 0.6 and 0. In Figure 6(c), the system skips the threading model adjustment when the thread count is

```
1  void init() {
2    threadingModelElasticity = true;
3    threadCountElasticity = false;
4  }
5  void adapt() {
6    if (threadCountElasticity) {
7      threadLevel = threadCountAdjustment();
8      sf = calculateSatisfactionFactor();
9      if (sf < THRE) { //Satisfaction factor optimization
10       direction = lastAdjustment.toContinue(threadLevel)
11       if (direction != NONE) { //Learning from hitory
12         threadingModelElasticity = true;
13         threadCountElasticity = false;
14       }
15     }
16   } else if (threadingModelElasticity) {
17     decision = threadingModelAdjustement();
18     if (decision == CHANGE) createNewAdjustHistEntry(newThreadLevel);
19     if (decision == STAY) updateAdustHistEntry(newThreadLevel);
20     if (decision == CONTINUE) return;
21     threadCountElasticity = true;
22     threadingModelElasticity = false;
23   }
24 }
```

Figure 7. Automating multi-level performance elastic components.

8 and 16 because the thread count change alone results in more than an 80% throughput improvement. In Figure 6(d), with `sf = 0`, the threading model adjustment is also skipped at 32 threads as `sf` of 0 means unless there is performance drop with an increased thread count, we will not trigger the threading model elasticity. Overall, with the use of the history record and satisfaction factor, the adaptation period is reduced from 1,000 seconds to just over 400 seconds. The improvement in adaptation time is achieved without sacrificing throughput; as can be seen in Figure 6, the final throughput after adaptation is close across different runtime setups. Both optimizations of learning from history and the use of satisfaction factor provide the settling time property of SASO.

Enhanced multi-level elasticity

In Figure 7, we show the core of the scheduling algorithm to automate multi-level elasticity. The iterative refinement process is composed of two components: the threading model elasticity (`threadingModelElasticity`) to adjust the scheduler queue placement and the thread count elasticity (`threadCountElasticity`) to adapt the number of threads. When the thread count changes (line 7 in Figure 7), we first calculate the satisfaction factor. If the satisfaction factor (`sf`) is less than the pre-defined threshold, we look into the record of the most recent threading model adjustment. Otherwise we skip adjusting the threading model. If the new thread count is within the thread range that has worked well with the current placement, we skip adjusting the threading model for now and continue to adapt the thread count.

If the number of threads is larger than the upper bound of the thread range observed so far, we explore if switching more operators to use dynamic threading can help improve performance. Otherwise if the thread count is smaller than the lower bound of the thread range, we try to decrease the number of operators using dynamic threading model. In Section 3.1, we have explained the threading model adjustment algorithm with the adjustment direction set to `UP`. When exploring the effect of decreasing the number of operators under dynamic threading model, the same algorithm is used in the reverse order, e.g., we start with the group of the lowest relative cost.

When no further improvement can be obtained by solely changing the threading model adjustment, we switch back to the thread count elasticity phase. If there is any change in the threading model choices for operators, we update the adjustment history record to reflect the change. Otherwise we update the thread range for the existing queue placement.

4 Evaluation

We evaluate our solution to automate multiple elastic components in terms of throughput and its ability to adapt to application workloads and available resources. We use representative benchmarks as well as production applications run on two different architectures to compare multi-level elasticity with the pure dynamic and manual threading models.

We measure application throughput at the sink operator. During the initial adaptation period, the elasticity algorithms explore the configuration space to find the best thread count and threading model choices, resulting in drastically varying throughput. Hence, we only compare the converged throughput to other baselines. We present results for two different processor architectures: 1) Xeon with maximum 176 cores, and 2) Power8 with two 3 GHz processors, each with 12 8-way SMT cores. One core has been disabled in our Power8 system, yielding 184 logical cores.

4.1 Representative benchmark evaluation

In order to evaluate our multi-level elasticity in a wide range of scenarios, we experiment with representative benchmarks that contain four graph architectures that form the basic building blocks for many Streams applications. Our benchmarks allow us to emulate the different operator load distributions and payload sizes we observe in practice.

Figure 8 shows the four graph architectures used in our benchmarks: pipeline, data-parallel, mixed and bushy. Each graph exposes a different amount of data and pipeline parallelism. We assign operators' tuple costs using either the *balanced* distribution or the *skewed* distribution. With the balanced distribution, every operator performs the same number of floating point operations per tuple. With the skewed distribution, 10% of the operators have the tuple cost of 10,000 FLOPs as *heavy-weight* operators, 30% are *medium-weight* operators with 100 FLOPs per tuple while the remaining are *light-weight* operators with a per tuple cost of 1 FLOPs. We randomly place the heavy-, medium- and light-weight operators in the graph without any prior knowledge. We also vary the number of operators as well as the tuple payload to span a broad range of possible scenarios. For these benchmarks, the manual threading model uses only one thread to execute all operators.

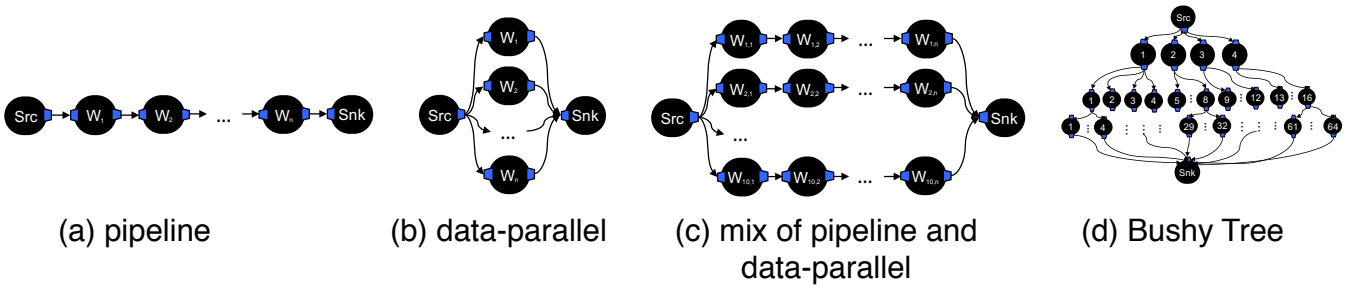

Figure 8. Stream graph architectures used.

Pure pipeline: Figure 9 presents the results for the benchmark runs with pipeline graphs (Figure 8(a)) on the two system architectures using the balanced and skewed cost distributions. For the balanced cost distribution, each operator has a tuple cost of 100 FLOPS. We vary the number of operators as well as tuple payload as seen on the x-axis. The operator count ranges from 100 to 1000 while the tuple payload is changed from 128B to 16384B. The left y-axis shows the speedup in throughput compared to the throughput of manual threading. The right y-axis shows the ratio of the operators using dynamic threaidng model. In full dynamic threading, this ratio is always one, i.e. every operator has a scheduler queue placed in front of it. Red bars are the throuhgput of manual threading model. Grey bars are the throughput achieved by thread count elasticity (dynamic threading). Black bars are the throughput by multi-level elasticity. Shaded light gray bars represent the ratio of operators under dynamic threading model with multi-level elasticity. The number on top of the black bars are the throughput speedups of our design compared to thread count elasticity alone.

The throughput speedups shown in Figure 9 demonstrate that the proposed multi-level elasticity scheme provides significant improvements over using only thread count elasticity (up to 22× when the payload is 16384 B). The general trend is that as the tuple payload increases, so does the performance impact of multi-level elasticity. With increasing payload, the fraction of operators that have scheduler queues (shown by the shaded gray bar) also decreases. This is because as the tuple payload increases, cost of using scheduler queues increases.

Figure 9(a) shows the results of using a balanced tuple cost distribution. When the operator count is 100 and the tuple payload is only 128B, the throughput improvement due to our design is negligible. This is expected since more than 80% of the operators end up choosing dynamic threading model, which is similar to the case of thread count elasticity alone. When the tuple payload is increased to 16384B, using only thread count elasticity hurts performance in comparison to manual threading. Combined with threading model elasticity, similar performance as manual threading is obtained by

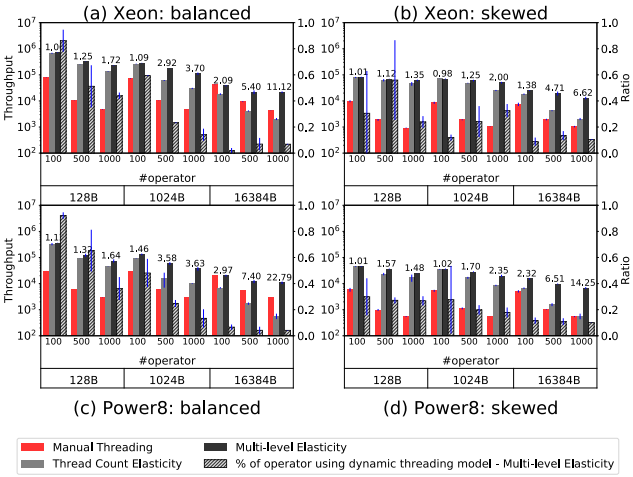

Figure 9. Throughput comparison for **pipeline** graphs.

only allowing a small fraction of operators to use dynamic threading model.

Another trend is that as the number of operators increases, multi-level elasticity has a higher impact on performance. This is because as the size of graph scales up, each individual thread is more critical because of the higher overall system workload. A better placement of scheduler queues not only reduces the queuing overhead but also helps balance the workload among the threads, improving utilization. For example, in Figure 9(a), when the operator count is increased to 1,000, queue elasticity helps improve the performance by more than three times. with the tuple payload fixed to 1024B, when the operator count is 100, the improvement brought by the queue elasticity is not significant Correspondingly, the ratio of operators using dynamic threading model decreases due to the limited parallelism available in the system.

The trends summarized above show that multi-level elasticity can serve as a safe default choice and can outperform both dynamic and manual threading for pipeline graphs. The trends apply to both processor architectures and both tuple cost distributions. We observe only minor changes in the trends due to the variance of the processor architecture or tuple cost distribution.

We also find that multi-level elasticity consistently improves resource utilization by using fewer threads. In Figure 9(d), when the payload of the 100-operator graph is 128B, though multi-level elasticity achieves similar throughput as dynamic threading, it successfully reduces the thread usage from 88 to 46.

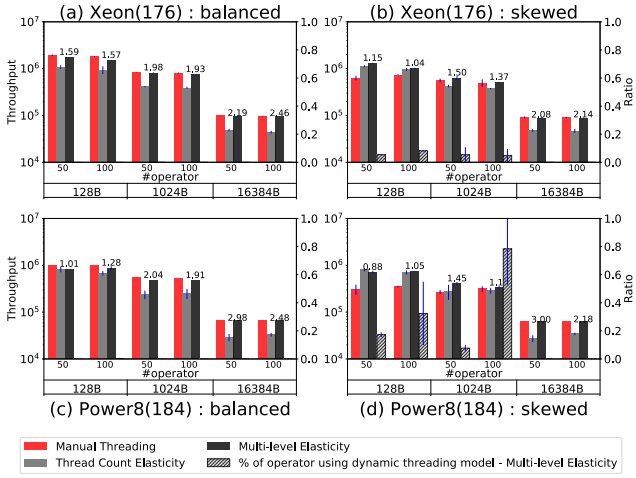

Figure 10. Throughput comparison for pure **data-parallel** graphs.

Pure data parallel: Figure 10 shows the performance comparison for the data-parallel graph. We vary the data parallel width from 50 to 100. The most notable trend is that sometimes thread count elasticity performs *worse* than manual threading. This is mostly due to the structure of the streaming graph. As can be seen in Figure 8(b), the Snk operator communicates directly with all the parallel worker operators. The implementation of the Snk operator maintains a local variable protected by a lock to track how many tuples have been processed for throughput calculations. Hence, as the thread count increases, contention among threads on the Snk operator also increases.

With the multi-level elasticity, the throughput achieved is consistently equal or better than that of manual threading. This is because the algorithm decides to choose dynamic threading model for only a few number of operators, leading to a similar configuration as manual threading. Given that this is all done when the streaming runtime had no prior knowledge of the graph architecture and used no user input, these results show the utility of our design for finding optimal configurations for such a graph architecture.

Mix of pipeline and data parallel: The results in Figure 11 are for graphs with a mix of data and pipeline parallelism. The degree of data parallelism is 10 while the number of operators in each data parallel path varies from 50 to 100. This graph architecture is a close representation of many realistic production scenarios. Despite the differences in the graph architecture, the performance trends obtained here are similar to those obtained in the previous cases. The performance

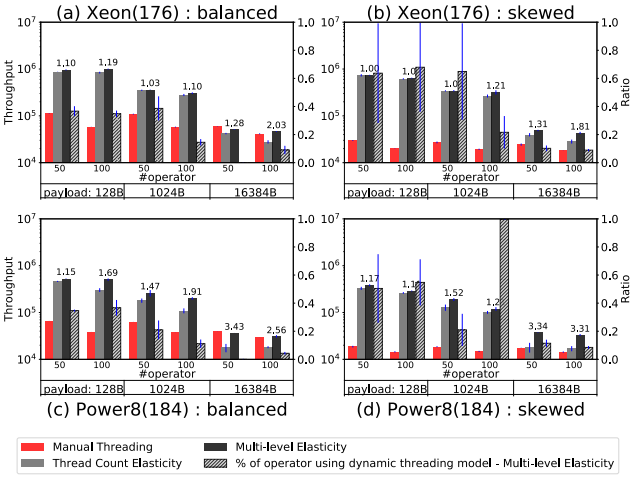

Figure 11. Throughput comparison with **mix of pipeline and data-parallel** graphs.

improvement obtained by use of multi-level elasticity increases as the operator count and tuple payload increases. At the same time, the fraction of operators under the dynamic threading model decreases with the increase of the number of operators and tuple payload. These results suggest that the proposed algorithm should apply to real-world scenarios with mixed data and pipeline parallelism, especially when tuple payload is at least a few hundred bytes.

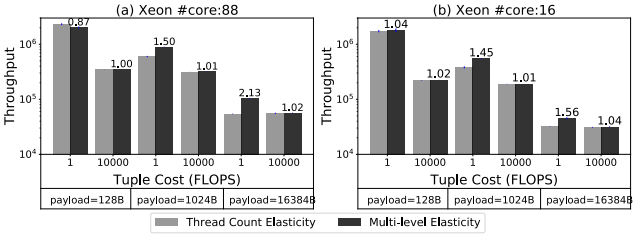

Figure 12. Throughput comparison with **bushy** graphs.

Bushy tree: Figure 12 shows the throughput comparison for the bushy graph (Figure 8(d)). The total number of operators is fixed at 82. We vary the number of available cores from 16 to 88 as well as the tuple cost for each operator from 1 to 10,000 FLOPS using the balanced cost distribution. The number on top of the shaded light gray bar are the throughput speedups of multi-level elasticity compared to thread count elasticity. When the tuple cost is low, the benefits of multi-level elasticity are high. This is because the cost of using dynamic threading is relatively higher for smaller workloads, making threading model selection more important. When the number of available cores changes, the multi-level elasticity adapts to the resource variation and still provides performance benefits.

Multi-level elasticity also uses fewer threads for better performance. For example, when the tuple payload is 16,384B

and tuple cost is 1 FLOPS, multi-level elasticity not only improves the performance by more than 50% but also reduces the number of threads used from 4 to 2.

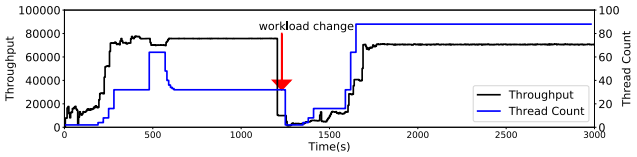

Figure 13. Adaptation to workload phase change.

Adaptation: We demonstrate the ability of multi-level elasticity to adapt to workload change in Figure 13 using a pipeline graph with 100 operators. The ratio of heavy-weight operators changes from 10% to 90% after 20 minutes into the run. After detecting the workload change, it takes multi-level elasticity 500 seconds to find a new configuration for better performance. Our scheme quickly adapts the thread count from 32 to 88 and the number of operators under the dynamic threading model from 42 to 86 in reaction to the increased workload.

4.2 Mini-application evaluation

Next, we evaluate the performance of multi-level elasticity with a small-scale application **VWAP** (52 operators) as shown in Figure 14(a). The goal of VWAP is to detect bargains and trading opportunities based on processing the volume-weighted average price from bids and quotes. The manual threading version in Figure 15(a) has no user-inserted threads. The hand optimized version is achieved by manually inserting *threaded ports* by the application developers. Both thread count elasticity and multi-level elasticity achieve at least two fold speedup compared to the manual threading and hand optimized version with fewer threads. The hand optimized version has 9 hand-inserted threads while the two elastic schemes stabilize at only 3 threads. On four cores, the multi-level elasticity helps improve performance further by 15% in comparison to thread count elasticity alone. The difference due to multi-level elasticity is marginal on 16, but 6% improvement is obtained on 88 cores. This is because VWAP has few operators, relatively low tuple payload, and light computation workload, and thus additional benefit of threading model choice for VWAP shows up when less resources are available such as on 4 cores.

4.3 Application evaluation

In order to stress test the multi-level elasticity support, we evaluate its performance using a hand-optimized highly dynamic production application, PacketAnalysis. This application was developed by IBM for a major telecommunications company for general network monitoring and specific threat analysis. It ingests packets directly from a 10 Gb/s Intel NIC using DPDK [6], which uses techniques originally designed for virtualization (SR-IOV) to achieve kernel bypass. After ingesting packets, the source operators forward tuples containing those packets down a variety of analysis pipelines, including DGA detection, tunneling detection and volumetric analysis. PacketAnalysis must operate as close to line-rate as possible, since it processes live packets.

All of our PacketAnalysis experiments were run on the 176 logical core Xeon system. The test data was sent over the network from a different system, looping over a PCAP file with historical DNS queries from one hour on a large corporate network. We experiment with two different application configurations: 1 source operator and 8 source operators. Each source operator uses DPDK to read live packets. The application with 1 source operator has 387 operators, and the 8 source operator application (Figure 14(b)) has 2305 operators.

The manual threading version has no user-inserted threads. The hand-optimized version uses multiple threads that have been inserted by its developers after spending significant performance analysis effort. As a result, the 1 source variant has 17 hand-inserted threads, and the 8 source variant has 129 hand-inserted threads. These hand-inserted threads are also bound to cores in a NUMA-aware manner.

Figure 15(b) presents PacketAnalysis's throughput with manual, hand-optimized manual threading, thread count elasticity, and multi-level elasticity. The executions with thread count elasticity and multi-level elasticity used between 8 and 20 threads over the runs, and still obtained performance close to the hand-optimized version using 129 threads. However, for both scenarios, multi-level elasticity resulted in only a marginal performance difference. We were surprised by this result, but it is explained by the fact that the tuples in PacketAnalysis are relatively small (~256 bytes) compared to the computationally expensive analytics.

4.4 Discussion

Our benchmarks show that the presented multi-level elasticity solution scales and improves performance when the amount of computational work and tuple sizes scale. As the tuple size increases, the performance gain increases proportionally. This performance increase is coupled with less resource usage. Our applications do not show the same dramatic performance improvement because for the specific scenarios we tested, computation costs and relative tuple sizes are modest. These results are in line with our results for benchmarks, as even there we see modest performance differences with tuple sizes less than 1 KB. Our application results also show that real applications are largely composed of pipelines of data-parallel regions. As the computation and tuple sizes of Streams applications increase, our results indicate that our multi-level elasticity algorithms should be able to handle them and obtain high performance.

Our results also show that our control algorithm meets the SASO properties. Low run-to-run variance suggests that the

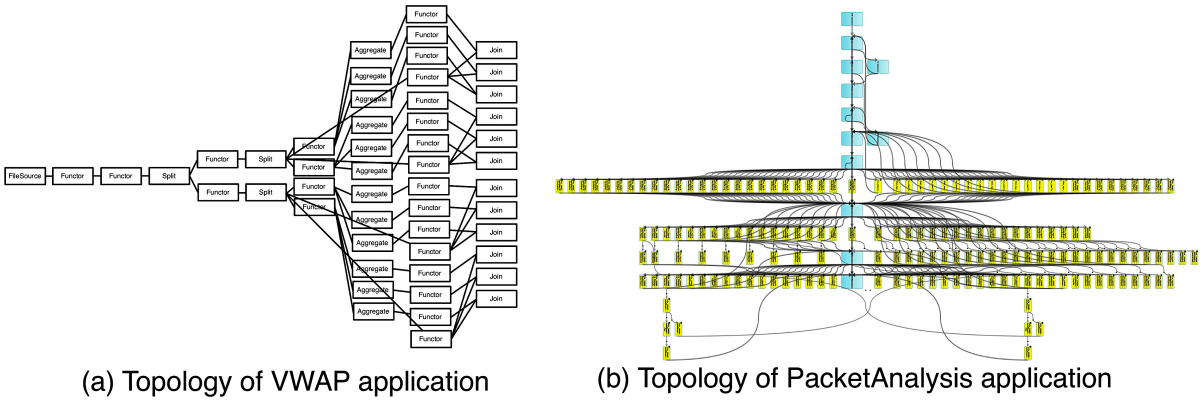

Figure 14. Application topologies: a) VWAP contains 52 operators, b) PacketAnalysis consists of 2305 operators in total; blue boxes represent 100+ operators, while yellow operators with the same name represent data-parallelism.

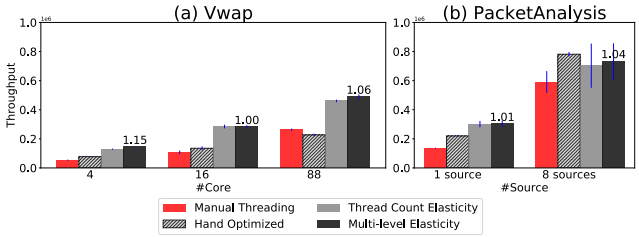

Figure 15. Throughput comparison for VWAP and PacketAnalysis.

multi-level elasticity solution provides stability, while performance benefits for benchmarks and performance matching the hand-optimized version for applications indicate accuracy of the elasticity solution. Shortened adaptation period and use of less resources usage demonstrate that the remaining two SASO properties of short settling period and avoiding overshooting are provided.

5 Related Work

Google Dataflow [7, 2] is a distributed system for stream and batch data processing. Based on the measured throughput, CPU utilization, and backlog, it auto-scales the number of workers to execute multiple instances of sub-graphs. Apache Flink [5], an open-source distributed stream processing system, supports manual rescaling of a streaming job, i.e., users can change the level of parallelism after restarting their programs from a savepoint. StreamMine3G [16] is an event stream processing system which provides both horizontal elasticity (changing the number of nodes) and vertical elasticity (changing the number of threads) for accommodating the fluctuation in the data stream. The focus of auto-scaling in Google Dataflow, manual rescaling in Flink, and elasticity in StreamMine3G is data parallelism, which is different from our work that exploits runtime elasticity to automatically adjust both data and pipeline parallelism.

Matteis et al. [3] adopt model predictive control techniques for elastic scaling to satisfy energy and latency constraints. Gordon et al. [9] utilize compiler techniques to automatically exploit data, task and pipeline parallelism in stream programs. In contrast our work relies on a lightweight online profiler and perfoms runtime adjustments to improve parallelism and resource usage.

StreamBox [17] focuses on exploiting out-of-order processing to maximize parallelism. It dynamically allocates thread from a thread pool to adapt to the different workload. Go [8] adopts the M:N threading model with a work-stealing scheduler for task-based parallelism. Dhalion [4] uses backpressure and congestion to identify bottleneck in stream processing and automatically adjusts the parallelism of each operator. DS2 [15] estimates the optimal level of parallelism for each operator in order to to maximize streaming system throughput. In our work, we keep the configuration of the parallelism of each operator constant while dynamically adjust the threading model and the coherent system to control both the thread count and threading model elasticity, while existing work only focuses on adjusting the thread count.

6 Conclusion

In this paper we describe our experience in automatically scheduling multiple performance elastic components in IBM Streams. Such endeavor is non-trivial, as different performance elastic components are designed with different objectives, have distinct adjustment granularity and unique resource requirements. We found that choosing the right adjustment order and adjustment direction based on the characteristic of each performance elastic components is essential to good parallelism and improved resource utilization (e.g. by more than 10× in some cases). We also found that thread count elasticity alone, usually adopted by modern streaming runtimes, is not enough for optimal performance. Augmented with the threading model elasticity and the right control mechanism to adjust multiple elastic components coherently, we are able to obtain promising results on over a hundred threads on Xeon and Power 8.

References

[1] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11), Aug. 2013.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12), Aug. 2015.

[3] T. De Matteis and G. Mencagli. Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. *SIGPLAN Not.*, 51(8):13:1–13:12, Feb. 2016.

[4] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[5] T. A. S. Foundation. Apache Flink. http://flink.apache.org, 2018. Retrieved August, 2018.

[6] T. L. Foundation. DPDK: Data Plane Development Kit. https://dpdk.org, 2018. Retrieved August, 2018.

[7] Google. Google Dataflow. http://cloud.google.com/dataflow/, 2018. Retrieved August, 2018.

[8] Google. The Go Programming Language. http://golang.org, 2018. Retrieved August, 2018.

[9] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGPLAN Not.*, 41(11):151–162, Oct. 2006.

[10] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback control of computing systems*. John Wiley & Sons, 2004.

[11] 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. 57(3/4), 2013.

[12] IBM. IBM Streams. https://ibmstreams.github.io, 2018. Retrieved August, 2018.

[13] IBM. IBM Streams Developer Community. https://developer.ibm.com/streamsdev, 2018. Retrieved August, 2018.

[14] IBM. streamsx.inet. http://ibmstreams.github.io/streamsx.inet, 2018. Retrieved August, 2018.

[15] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 783–798, 2018.

[16] A. Martin, A. Brito, and C. Fetzer. Scalable and elastic realtime click stream analysis using streammine3g. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 198–205, New York, NY, USA, 2014. ACM.

[17] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. Streambox: Modern stream processing on a multicore machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 617–629, 2017.

[18] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, New York, NY, USA, 2013. ACM.

[19] S. Schneider, B. Gedik, and M. Hirzel. Language runtime and optimizations in IBM Streams. *IEEE Database Engineering Bulletin*, 38(4), 2015.

[20] S. Schneider and K.-L. Wu. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 648–661, New York, NY, USA, 2017. ACM.

[21] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, New York, NY, USA, 2014. ACM.