Low-Synchronization, Mostly Lock-Free, Elastic Scheduling for Streaming Runtimes

Scott Schneider
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
scott.a.s@us.ibm.com

Kun-Lung Wu
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
klwu@us.ibm.com

Abstract

We present the scalable, elastic operator scheduler in IBM Streams 4.2. Streams is a distributed stream processing system used in production at many companies in a wide range of industries. The programming language for Streams, SPL, presents operators, tuples and streams as the primary abstractions. A fundamental SPL optimization is operator fusion, where multiple operators execute in the same process. Streams 4.2 introduces automatic submission-time fusion to simplify application development and deployment. However, potentially thousands of operators could then execute in the same process, with no user guidance for thread placement. We needed a way to automatically figure out how many threads to use, with arbitrarily sized applications on a wide variety of hardware, and without any input from programmers. Our solution has two components. The first is a scalable operator scheduler that minimizes synchronization, locks and global data, while allowing threads to execute any operator and dynamically come and go. The second is an elastic algorithm to dynamically adjust the number of threads to optimize performance, using the principles of trusted measurements to establish trends. We demonstrate our scheduler's ability to scale to over a hundred threads, and our elasticity algorithm's ability to adapt to different workloads on an Intel Xeon system with 176 logical cores, and an IBM Power8 system with 184 logical cores.

CCS Concepts • **Software and its engineering** → **Runtime environments**; *Data flow languages*; • **Computing methodologies** → Parallel programming languages

Keywords lock-free; elastic; runtime scheduling; stream processing

PLDI'17, June 18–23, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062366

1. Introduction

Parallel and distributed systems are no longer limited to expert developers for niche application domains. The emergence of multicore processors has made parallelism mainstream. The increasing reliance on cloud infrastructure forces developers to contend with distributed systems. As developers are expected to program parallel and distributed systems, programming languages and runtime systems must adapt so that non-experts can obtain high performance.

Solutions for online stream processing are at the forefront of this trend. Such systems are distributed and highly parallel to meet the throughput and latency requirements of real time data analytics. IBM Streams [13, 23] is one such system that is used in production at dozens of companies, in a wide variety of industries, including aviation, medicine, transportation, banking and telecommunications. Streams also powers the Streaming Analytics service on IBM Bluemix [1].

The main means of programming for Streams is through the Streams Processing Language (SPL) [11, 12]. SPL offers three main abstractions: operators, tuples and streams. Operators process continually arriving tuples, and communicate exclusively by sending and receiving tuples over ordered streams. The programming model is asynchronous dataflow.

At runtime, operators execute inside PEs (processing elements). Each PE is a separate operating system process. Streams that cross PE boundaries communicate using the network, allowing PEs to execute across distributed hosts. The process of assigning operators to PEs is called *fusion*.

Prior to Streams 4.2, operator fusion and thread placement was handled exclusively by developers at compile time. Through operator config options, developers could specify which operators should be fused together into PEs. Operators without fusion directives would end up in their own, isolated PE. Threads were manually added to PEs through the use of *threaded port* configuration options on operators, which specified that a separate thread should execute that particular operator input port. In general, the same thread would execute all downstream operators until it encountered a threaded port. Through these controls, expert developers could finely tune application performance when they had a

deep understanding of their application's runtime behavior and deployment system [20].

Experience in the field showed such experts were rare; few developers had the required constellation of skills. Developers need deep knowledge of SPL semantics, a full understanding of their application's runtime behavior and full awareness of the production systems. We encountered real deployments of applications with thousands of unfused operators that became thousands of PEs running on a handful of hosts with a modest number of cores. Such deployments suffer from excessive PE-to-PE communication in the application, and massively over-subscribed systems. In a cloud context, it becomes increasingly unlikely our users will have detailed knowledge and control over production systems.

We have also observed the difficulty that experts had optimizing large applications. Such experts have the ability to find the right places to insert threads, but doing so is time consuming: the search space is large, and the time allotted for performance improvement is small.

In Streams 4.2, operator fusion and thread placement is handled automatically. Fusion occurs when applications are deployed, and threads are placed at runtime. Our goal is not to automatically achieve the "best" fusion and thread placement, but a reasonable one that is good enough for most application deployments, which developers and administrators can use as a baseline for performance refinement. Other distributed streaming systems used in production [5, 10, 15, 22] assume a static number and assignment of threads. Existing solutions in the literature are still prone to load imbalance [24], or do not have scalability as a primary goal [17].

We present the new scheduler and elastic thread adaptation algorithms in Streams 4.2 that solve this problem. Under automatic fusion, large SPL applications with many operators will have PEs with many operators. (The fusion process itself is outside of this paper's scope.) Our scheduler must scale to hundreds of threads and thousands of operators, still maintain tuple order on streams between operators, and allow any thread to execute any operator so that we can elastically add and remove threads. Maintaining tuple order is fundamentally a PE-global concern, but naive use of PE-global state will hinder scalability. Our elasticity algorithms must be able to seek the optimal thread level under many different kinds of applications, workloads and systems. Specifically, the contributions of this paper are:

- Scheduling algorithms for an ordered streaming runtime that scale to hundreds of threads by minimizing synchronization, global data and locks.
- Elasticity algorithms that optimize the number of threads in a streaming runtime by establishing trust in measurements and discovering performance trends.
- Evaluation of the scheduler's scalability and the elasticity algorithm's accuracy on large multicore systems with over a hundred threads on two separate processor architectures.

```
composite LoginFailures(output Failures) {
type
  LogLine = timestamp time, rstring hostname, rstring srvc,
            rstring msg;
  Failure = timestamp time, rstring uid, rstring euid,
            rstring tty, rstring rhost, rstring user;
graph
  stream<rstring line> Lines = FileSource() {
    param format: line;
          file: "/var/log/messages";
  }
  @parallel(width=7)
  stream<LogLine> ParsedLines = Custom(Lines) {
    logic onTuple Lines: {
      list<rstring> tokens = tokenize(line, " ", false);
      rstring date = makeDate(tokens[1]);
      rstring time = makeTime(tokens[2]);
      timestamp t = makeTimestamp(date, time);
      submit({time = t, hostname = tokens[3],
              srvc = tokens[4], msg = flatten(tokens[5:])},
             ParsedLines);
    }
  }
  stream<LogLine> FailuresRaw = Filter(ParsedLines) {
    param filter:
            findFirst(srvc, "sshd", 0) != -1 &&
            findFirst(msg, "authentication failure", 0) != -1;
  }
  @parallel(width=4)
  stream<Failure> Failures = Custom(FailuresRaw) {
    logic onTuple FailuresRaw: {
      list<rstring> tokens = parseMsg(msg);
      submit({time = FailuresRaw.time,
              uid = tokens[0], euid = tokens[1],
              tty = tokens[2], rhost = tokens[3],
              user = size(tokens) == 5 ? values[4]: ""},
             Failures);
    }
  }
}
```

Figure 1: SPL example. The composite looks for failed logins from system messages.

2. Background

2.1 IBM Streams and SPL

The language for Streams is SPL, which provides an asynchronous dataflow programming model. SPL offers three main abstractions: operators, tuples and streams. Operators contain logic for processing incoming tuples, and potentially producing output tuples. Programmers can create their own custom logic in SPL, but SPL also provide libraries of predefined operators for both standard operations (such as filtering or aggregation) and specialized operations (such as processing time series data or interacting with external systems such as databases or HTTP servers). Streams are typed connections between operators that carry tuples from the sending operator's output ports to the receiving operator's input ports. Operators exclusively communicate over streams; operators can have local state, but no shared global state.

Figure 1 is an example SPL *composite* operator—it is not a full application, but a reusable component that produces a stream of failed login attempts on a particular system. The source operator is a `FileSource` that produces tuples on the stream `Lines`, where each tuple is a line from `/var/log/messages`. The lines are unstructured and stored in simple strings (`rstrings` in SPL). In order to transform unstructured to structured data, we invoke a `Custom` operator, which allow SPL developers to specify custom logic. For

every tuple, the logic in the `Custom` operator performs an initial parse of the line, producing a `LogLine` tuple. After initial parsing, a `Filter` operator filters out all tuples that are not from the `sshd` service, and which are not failed login attempts. Another `Custom` operator parses the `sshd` service message to produce `Failure` tuples which can identify the user names and remote hosts that initiated the failed login. These tuples are sent as output to the `Failures` stream.

The two `Custom` operators have `@parallel` annotations, which are directives similar to OpenMP [16] pragmas. When applied to an operator invocation, `@parallel` creates data parallel replicas of the operator, and automatically handles splitting the tuples to the replicas and creating new streams. Pipeline and task parallelism also regularly appear in SPL programs. Pipeline parallelism occurs naturally, as operators can independently process different tuples. Task parallelism occurs when the same tuples are sent to different subgraphs.

Example SPL applications are available on GitHub [8, 9], in Hirzel et al. [12] and the documentation is online [21].

2.2 Threading Models

Streams 4.2 introduces *threading models*. The threading model prior to version 4.2, where developers had to manually place threads, is now called *manual*. A threading model that places threaded ports between every operator is called *dedicated*, as each operator input port has its own dedicated thread. The *dynamic* threading model allows any thread to execute any operator, which also enables runtime elasticity by being able to dynamically add and remove threads.

No one threading model is wholly superior. The manual threading model has the lowest latency, as there are no queues between operators, and no tuple copies, but it is single-threaded by default. Dedicated has multiple threads, and in the common case, can use synchronization-free single producer, single consumer queues between threads. But, the number of active threads scales linearly with the number of operators with the dedicated model. When the number of operators greatly outnumbers the available logical cores, it is not acceptable. The dynamic threading model offers a middle ground: multithreaded by default, and the number of threads can elastically adapt to maximize throughput. However, because any thread can execute any operator, there is necessarily more thread synchronization.

In Streams 4.2, developers can control the threading model from SPL through the `@threading` annotation. For example, we can invoke the `LoginFailures` composite from Figure 1 and write its tuple to a file while specifying the dynamic threading model:

```
@threading(model=dynamic)
composite Main {
graph
  stream<Failure> Failures = LoginFailures() {}
  () as Sink = FileSink(Failures) {
    param file: "failures.txt";
  }
}
```

Or, we can instead make the results available as a web service while using the dedicated threading model:

```
@threading(model=dedicated)
composite Main {
graph
  stream<Failure> Failures = LoginFailures() {}
  () as Sink = TCPSink(Failures) {
    param role: server; port: "http";
  }
}
```

We present the operator scheduler and elastic thread manager which implements the dynamic threading model. We experimentally compare it against manual and dedicated, showing that it provides a middle-ground: scalable parallelism by default, which can adapt to systems at runtime.

2.3 Constraints

Because Streams is an existing product with customers who use it in production, we must design our operator scheduler under the constraint of maintaining backwards compatibility. There are three main concerns: API compatibility, tuple ordering, and existing threads.

Figure 1 shows two types of operators. The `FileSource` and `Filter` operators are implemented in C++, and are invoked from SPL. In C++, developers write their tuple-processing logic in member functions that implement the `process(Tuple&)` interface. In order to send result tuples to downstream operators, they invoke the `submit(Tuple&)` interface. Operators can also be implemented in Java, with a similar interface. We cannot change this interface, or require operators implemented in C++ or Java to call any additional functions. This requirement means that all of our scheduling decisions must happen when operators explicitly call `submit()`—our operator scheduler is non-preemptible.

The other kind of operators in Figure 1 are `Custom` operators, where the logic is implemented directly in SPL. In principle, it would be possible for the compiler to insert hooks for a scheduler to approximate preemption between explicit calls to `submit()`. However, such an approach is brittle, and would still not allow preemption for operators implemented in a native language, which includes the standard library.

Any operator scheduler must also preserve global tuple order. The programming model for SPL is that operators can depend on the order in which operators see tuples, but they cannot depend on the order in which operators execute. SPL semantics are that streams are first-in, first-out queues, so all operators that receive tuples from a particular stream should see them in the same order the upstream operator sent them. Maintaining this order is inherently a PE-global issue.

Finally, there are threads we cannot control: operator threads, requested threaded ports, and PE input ports. Operators can create their own threads, which are managed by the operators themselves. Our scheduler cannot remove such threads—but it can get such threads to execute other operators on its behalf. Source operators are the most common kind of operators which create their own threads: they necessarily have at least one thread which retrieves data from

outside the system (or generates it on the fly), and submits new tuples downstream. Requested threaded ports are those explicitly requested by developers. PE input ports are the point where tuples enter the PE from operators in other PEs. Each PE input port has its own thread which is responsible for receiving data from the network, deserializing the tuple, and executing the receiving operators with that tuple. These threads must exist at runtime, and cannot be removed. Our dynamic scheduler must be able to work correctly even in the presence of threads it does not control.

3. Related Work

The work of Tang and Gedik [24] is a different solution to our current problem, and was implemented in a research prototype of Streams. Their solution does not introduce a scheduler, but instead tries to find the right places to inject threaded ports at runtime. Their work uses a model to make the search space tractable, injects the threaded ports, monitors the resulting performance, and adjusts decisions accordingly. The benefit over the approach presented in this paper is reduced thread synchronization because a single thread executes a particular set of operators. The difficulty is that because threads are fixed to executing a particular subsection of the stream graph, it is more prone to imbalance. We also purposefully separated the scheduling from the elasticity so that both parts could be simpler.

C-Stream [17] is, similar to this work, an elastic streaming runtime. It is implemented as a C++11 library. Unlike in SPL, operators in C-Stream use a pull-based model: operator implementations explicitly request incoming tuples from their input ports. Operators are executed as co-routines so that the runtime can suspend operators when they request or send data. C-Stream's runtime elastically changes the number of threads based on utilization: when overall thread utilization is low, it decreases the number of threads, and when the overall utilization is high, it increases them. The work presented in this paper changes the number of threads based on measured throughput. C-Stream also elastically changes the amount of data parallelism, while in SPL, data parallelism is currently fixed at runtime. C-Stream has a pluggable scheduler, allowing for different policies, but it was not designed explicitly for high scalability; C-Stream experiments scale to 12 threads. The scheduler algorithms presented in this paper were designed based on what techniques could scale to over 100 threads.

Storm [22] is a distributed streaming platform implemented in Java. Storm's *worker process* is similar to Stream's PE [25], and *spouts* and *bolts* are analogous to SPL's operators. Worker processes contain *executors*, which execute multiple spouts or bolts of the application. Executors have two threads, one for processing application logic, and one for sending result tuples out to other worker processes. The assignment of spouts and bolts to executors is static, unlike in our scheduler, and the number of threads is not elastic. Heron [10] is a re-implementation of Storm that retains the same API, but changes the internal architecture. The *heron instance* is akin to the worker process, and is constrained to running a single spout or bolt. The heron instance still has two threads, but one thread is a gateway thread responsible for all communication. The other thread is responsible for executing the spout or bolt. Again, unlike our solution, the number of threads cannot change at runtime, and threads cannot change which spouts or bolts they execute. Unlike Streams in general, users do not have control over threads.

Naiad [15] is a distributed system designed for data parallel, streaming and iterative workflows. Messages have logical timestamps and location-generation metadata that allows the system to reason about message order and priority. Workers in Naiad execute vertices (similar to operators), communicating through shared queues. Workers can execute different vertices, but there is no thread-level elasticity.

Cilk [2, 6], with its work-stealing scheduling, is one of the most influential schedulers for task-based multithreaded parallelism. However, task-based parallelism is fundamentally different than a stream graph.

In task-based parallelism, tasks produce their own work by spawning more tasks. When a thread steals a task, it can operate independently of other threads as long as its tasks spawn more tasks. In a streaming context, threads do not work independently. They receive work from other threads. In this context, *stealing* actually means *cutting in* and executing a subgraph for another thread. That is not productive: those operators would soon be executed anyway. Rather, it's better for an idle thread to go find an operator that is not currently being executed. It is likely such an operator will either produce work for the currently productive threads, or execute work that blocked threads depend on. Thinking of it in another way, task-based parallelism produces a dynamic dependency tree. Nodes at the same level of the tree are independent, and threads can pick-off nodes from that tree and stay independent. In streaming parallelism, the dependency graph is static. Trying to take work from other threads will increase thread interference and cause an imbalanced load. We did experiment with ideas inspired by work-stealing, but they always ended up creating more load imbalance and incurring unnecessary synchronization.

The novelty of the work presented in this paper is in the combination of a dynamic and elastic operator scheduler in a streaming runtime that can scale to hundreds of threads.

4. Design

Our operator scheduler has two main goals: scalability and elasticity. As the number of threads increases, the scheduler must scale. We also must be able to add and remove threads dynamically, so that we can discover the best number of threads at runtime. The rest of this section elaborates on these requirements, and presents the scheduling and elasticty algorithms which satisfy them.

4.1 Scheduling

We have the following requirements from a PE-global operator scheduler:

1. The PE must be able to add and remove scheduler threads at runtime. This property enables elasticity, and means that threads cannot be statically assigned to operators.
2. Tuple order must be maintained. Formally, if operator A emits tuples a_0, a_1, ..., a_n on an output port, and operator B has an input port subscribed to A's output port, operator B must see the same tuples in the order a_0, a_1, ..., a_n. Note that in the absence of this requirement, our scheduler could follow a generic thread pool pattern.
3. The scheduler must scale as we add threads. Not only is global locking not an option, we also need to minimize scheduler threads touching global data.
4. We cannot change the pre-existing operator API. We have a significant amount of user applications that must still work correctly under our new scheduler without having to make any changes to user code.

These requirements are in tension. The first requirement means that any thread must be able to execute any operator, and in particular, that if a thread executes an operator, a different thread may execute it next time. The second requirement, maintaining tuple order across operators, is inherently a PE-global problem. Both require some communication across threads. In order to meet the third requirement, we must eliminate unnecessary global communication, and find a way to delay necessary communication. The fourth requirement means that the scheduler can only take control when an operator voluntarily submits a tuple.

These requirements push us towards the design in Figure 2. Each operator input port has a single-producer, single consumer lock-free FIFO queue of tuples. The scheduler maintains a lock-free free list of operator input ports; when an operator input port is on this queue, it may be free to execute. To determine if an operator input port is actually free to execute, we check a flag associated with its queue. Checking this flag ensures that only a single thread will execute an operator input port at a time. By maintaining this property, we maintain our tuple order requirement: upstream threads will necessarily enqueue their tuples in submission order, and if only a single thread pops tuples from this queue to execute them, they must be processed in order.

Obtaining an operator input port by popping it from the free list is not enough to guarantee a scheduler thread's exclusive access to that input port. There are two opportunities where a thread will try to get exclusive access to an input port: from the main scheduler loop, by popping it off the free list, and when pushing a tuple into that input port's queue. Tuple queues can become full, preventing threads from pushing tuples into them. Such threads do not block, and nor do they go to the global `freePorts` list looking for work. Rather, they alternate between trying to execute the operator that blocks their progress and pushing the current

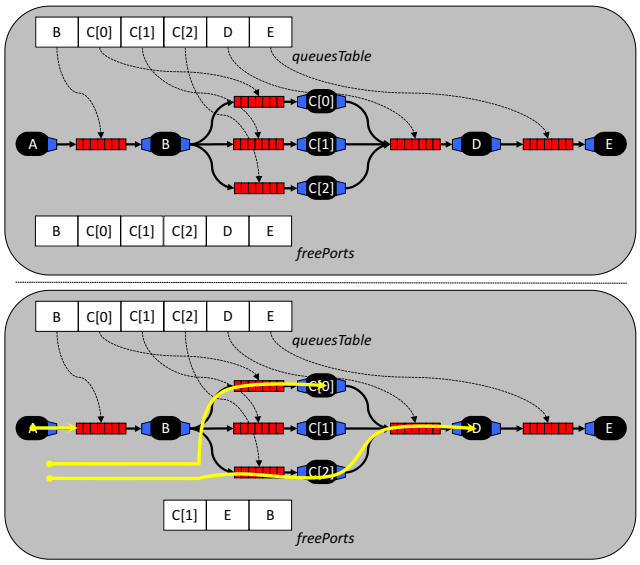

Figure 2: Top: PE at startup; all operator input ports are in the `freePorts` lock-free list, and the `queuesTable` maps an operator input port to its single-producer, single-consumer lock-free queue. **Bottom**: PE during execution; yellow lines represent threads executing operators. Source operator `A` has its own thread, and the PE maintains two in its thread pool. Note that the operators being executed are not on `freePorts`, and `queuesTable` remains unchanged.

tuple into that operator's queue. Such a policy is a result of the principles that enable scalability: avoid synchronization and global data where possible.

The following sections present our algorithms in detail. We present our algorithms in C++, rather than using generic psuedocode. Lock-free algorithms depend on synchronization primitives, whose particulars can change depending on the underlying hardware. Rather than defining our own semantics, we find it clearer to borrow the atomic interfaces defined in C++11 [4], and the lock-free lists defined in the Boost.Lockfree package [3].

4.1.1 Data Structures and Initialization

Figure 3 shows the data structures used in our scheduling algorithms. The `SPSCEnforcer` structure wraps a single-producer, single-consumer lock-free queue. It contains the flags (`prodLocked` and `consLocked`) which enforce when it is safe to produce or consume a tuple from the queue.

Protecting a lock-free queue with what are effectively locks is a seemingly odd design choice. The consumer lock is to ensure that only one thread executes an operator input port at a time. The programming model actually allows multiple threads to execute an operator; it is the operator's responsibility to protect its own state. But allowing only one thread to execute an operator input port at a time is how we enforce proper tuple ordering: if only one thread has exclusive access to the queue and the operator, we can ensure that the tuples in the queue are processed in the order they appear. The producer lock is needed because we want to use single-producer, single-consumer lock-free queues. While we have locks on both ends, we still allow simultaneous access by

```
struct SPSCEnforcer {
  lockfree::spsc_queue<Tuple> queue;
  atomic<bool> prodLocked;
  atomic<bool> consLocked;

  bool prodTryLock() {
    bool expected = false;
    return prodLocked.compare_exchange_strong(expected,
                                              true,
                                    memory_order_acquire);
  }
  void prodUnlock() {
    prodLocked.store(false, memory_order_release);
  }
  bool consTryLock() {
    bool expected = false;
    return consLocked.compare_exchange_strong(expected,
                                              true,
                                    memory_order_acquire);
  }
  void consUnlock() {
    consLocked.store(false, memory_order_release);
  }
  bool push(Tuple& tuple) {
    if (prodTryLock()) {
      if (queue.push(tuple))
        return true;
      prodUnlock();
    }
    return false;
  }
};

unordered_map<Port*, SPSCEnforcer*> queuesTable;
lockfree::queue<Port*> freePorts;
```

Figure 3: Data structures.

the thread on each end. We could remove the producer lock if we used multiple-producer, multiple-consumer lock-free queues. However, in our system, single-consumer, single-producer lock-free queues perform better.

The lock-free queues we use are from the Boost.Lockfree package. In our presented algorithms, we use two functions from their interface:

```
bool push(const T& item); // adds item to queue
bool pop(T& item);        // writes head of queue to item
```

Both functions return `true` if the operation succeeded, and `false` if the operation failed. As for all lock-free data structures, operation failure can mean either it is unable to alter the data structure further (it is either full or empty), or that it experienced contention with another thread.

We do not define `Tuple` and `Port` in this paper. A `Tuple` contains all of the data for the tuple to be processed, as well as meta-data related to executing it, such as the destination input port. (In Streams, we also support *punctuations*, which are in-band control signals sent over streams. We only show tuples for simplicity.) A `Port` knows how to execute the operator it is associated with, given a tuple.

The `queuesTable` and `freePorts` structures are global to the PE. The PE initializes both during startup, before any threads start executing tuples. During each operator's initialization phase, it registers each input port with the PE. The PE creates an association between that input port and its `SPSCEnforcer`, stored in `queuesTable`. It also adds that input port to `freePorts`. After initialization, `queuesTable` is read-only; it is not a lock-free data structure, but as long as we do not modify it, we can use it as a global look-up table.

```
void schedule(const atomic<bool>& suspended,
              const atomic<bool>& shutdown,
              const atomic<bool>& portsClosed) {
  Tuple tuple;
  while (findWorkBlocking(tuple, suspended,
                          shutdown, portsClosed)) {
    execute(tuple);
    SPSCEnforcer* q = queuesTable[tuple.port()];
    while (q->queue.pop(tuple)) {
      execute(tuple);
      if (suspended.load(memory_order_acquire) ||
          shutdown.load(memory_order_acquire) ||
          portsClosed.load(memory_order_acquire))
        break;
    }
    q->consUnlock();
    while (!freePorts.push(tuple.port())) {}
  }
}
```

Figure 4: Main scheduling loop.

4.1.2 Main Scheduling Loop

After initialization, the PE launches the scheduler threads, which execute the algorithm in Figure 4. The values `suspended`, `shutdown` and `portsClosed` are local to each thread. The first, `suspended`, is for when the PE tells a thread that it is suspended. It is naturally a thread local value, and relevant for elasticity (see Section 4.2). The other two values, `shutdown` (for indicating an explicit shutdown request by the PE) and `portsClosed` (for indicating that all input ports have processed their final punctuations, and there will be no more tuples to process) are inherently PE-global concepts. However, each scheduler thread checking the same global value as a condition to continue processing can severely limit scalability when the number of threads is in the dozens or hundreds. In the event of false-sharing, the cache line which contains those values must bounce around all of the cores in the system.

To prevent false-sharing, we create a local value for each thread, and when we need to update these values, we iterate through all threads, updating them all. This process incurs a cost at shutdown, but that is far more preferable than paying the cost every iteration of the scheduling loop. We find that thread local values are a more portable way of preventing false-sharing than trying to anticipate cache line sizes and padding appropriately.

These local values illustrate the primary means through which our scheduler achieves scalability: limit global data as much as possible. As the number of threads in a single node reaches the hundreds, we must start to view multithreaded programming through the lens of distributed systems. With that view, global data between threads is the same as sending messages across hosts.

The main logic in Figure 4 calls `findWorkBlocking()` continually, which looks through the global `freePorts` free list for a free operator input port with a tuple to execute. If it ever returns `false`, then one of the stopping conditions has been met, the loop will exit, and the thread will return.

The common case, when it returns `true`, indicates two things: we have a tuple to execute, and we have exclusive access to that operator's input port. We do not define `execute()`;

```
bool findWorkBlocking(Tuple& tuple,
                      const atomic<bool>& suspended,
                      const atomic<bool>& shutdown,
                      const atomic<bool>& portsClosed) {
  long delay = 1;
  while (!shutdown.load(memory_order_acquire) &&
         !portsClosed.load(memory_order_acquire)) {
    if (suspended.load(memory_order_acquire))
      suspend();
    if (findWorkNonBlocking(tuple))
      return true;
    block(delay);
    if (delay < DELAY_THRESHOLD)
      delay *= 10;
  }
  return false;
}

bool findWorkNonBlocking(Tuple& tuple) {
  Port* first = NULL;
  if (freePorts.pop(first)) {
    SPSCEnforcer* q = queuesTable[first];
    if (q->consTryLock()) {
      if (q->queue.pop(tuple))
        return true;
      q->consUnlock();
    }
    while (!freePorts.push(first)) {}

    Port* port = first;
    while (freePorts.pop(port)) {
      q = queuesTable[port];
      if (q->consTryLock()) {
        if (q->queue.pop(tuple))
          return true;
        q->consUnlock();
      }
      while (!freePorts.push(port)) {}

      if (port == first)
        break;
    }
  }
  return false;
}
```

Figure 5: Looking for global work.

given a tuple, it knows how to execute that tuple on the operator it was submitted to. But, after executing that tuple, we do not immediately go back to the global free list. We paid a cost by touching global data; we now need to amortize that cost. Since we know we have exclusive consume access to this input port, we can go ahead and pop off and execute all of the tuples from its queue.

The first time we fail to pop a tuple from this queue means that the queue is empty. We unlock it, place it in the back on the global `freePorts` list, and look for more global work. Pushing recently executed input ports to the back of the free list favors executing the least recently executed operators.

4.1.3 Finding Work

The algorithms for finding global work, Figure 5, are split into blocking and non-blocking functions. The blocking algorithm, `findWorkBlocking()`, is mostly concerned with what to do if the thread has been suspended, or no work can be found. Both `suspend()` and `block()` should use synchronization primitives that prevent the thread from consuming resources. In Streams, we use Pthread mutexes and condition variables. Runtime systems which try to avoid synchronization sometimes try to use system calls like `nanosleep()` for such purposes. In our experience, such attempts to avoid synchronization primitives end up costing more.

If we can't find any work, we follow an exponential back-off pattern calling `block()`. `DELAY_THRESHOLD` is a scheduler parameter with a trade-off: lower thresholds will favor lower latency, and higher thresholds will favor consuming less resources when data is not flowing. In Streams, we block for up to ten milliseconds. Note that `findWorkBlocking()` will only return `false` when the PE is shutting down.

The function `findWorkNonBlocking()` is where the scheduler actually looks over the free lists for global work. It finds an operator input port that satisfies three properties:

1. Exists on the free list.
2. Is not taken by another thread.
3. Has a tuple in its queue.

We must satisfy these three properties while iterating over a lock-free list that other threads may also simultaneously iterate over, and has no deterministic "end." The algorithm performs a priming read of the list, so that it can remember the input port it saw first. Every operator input port it pops off the list that does not meet the following two requirements is pushed to the back of the list. If any pop of the free list fails, or if we recognize the first input port we saw, then we stop searching the list, return `false`, and the outer algorithm in `findWorkBlocking()` will block for a period of time.

Note that a thread is *not* guaranteed to find work if it exists. Of course, pops of the free list may fail because the list is empty, but they may also fail if there was interference from another thread. We stop looking for work the first time this happens. A principle that we apply throughout our algorithms is that if we encounter contention during an operation, and we do not need to complete that operation for correctness, we abandon it and do something else. A failed operation due to contention on one thread means a successful operation on another thread, so even though a particular thread is not guaranteed to find work if it exists, *some* thread will find work if it exists.

Because we abandon our search the first time we see the first port again, it's also possible, on a given call to `findWorkNonBlocking()`, that there are input ports we never tested. They may have been added to the free list by another thread after we started. Again, we favor abandoning our search early, rather than trying to determine if there are any operator input ports that we "missed."

4.1.4 Pushing Tuples

All prior algorithms are concerned with executing tuples—taking work out of the system. The algorithms in Figure 6 are concerned with pushing tuples into operator input port queues—adding work into the system. When an operator submits a tuple downstream, it will eventually call `push()` from Figure 6. If the push into the operator input port's queue fails, we assume that the queue is full, so we call `reSchedule()`. Again, it is possible that the queue is not full, and that instead we conflicted with another thread that had

```
void push(Tuple& tuple) {
  SPSCEnforcer* q = queuesTable[tuple.port()];
  if (!q->push(tuple))
    reSchedule(q, tuple);
}

void reSchedule(SPSCEnforcer* q, Tuple& tuple) {
  while (!q->push(tuple) && !isFinished()) {
    if (q->consTryLock()) {
      Tuple reTuple;
      int processed = 0;
      while (q->queue.pop(reTuple)) {
        execute(reTuple);
        ++processed;
        if (processed > RESCHED_LIMIT ||
            isFinished() || isSuspended())
          break;
      }
      q->consUnlock();
    }
  }
}
```

Figure 6: Pushing work into an operator input port's queue, and potentially executing work for it.

acquired the `prodLocked` flag for that queue. Rather than trying to distinguish these two cases, we execute `reSchedule()` immediately, which handles both.

The rationale behind `reSchedule()` is simple: this thread cannot continue until it pushes the current tuple into this operator's input port queue, but the queue is (likely) full. Rather than blocking and waiting for another thread to free up the queue for us, we do it ourself. Executing another operator input port inside `reSchedule()` is why we need to also place a lock on the input port queues: it is not practical for us to find this input port on the free list, as that would require a linear and destructive walk. Adding the lock allows the pushing thread to get exclusive access to the consume side of the input port queue without touching global data. Unlike in `schedule()`, we do not clear out the entire queue, but rather stop after having processed `RESCHED_LIMIT` tuples. Our goal is to clear out enough work so that we can push the original tuple and move on; we leave most of the execution to when `schedule()` grabs this operator input port off the free list. In Streams, `RESCHED_LIMIT` is $1/4$ of the queue size.

In the early phase of designing these algorithms, we experimented with `reSchedule()` calling `findWorkNonBlocking()` either immediately, or after a certain number of loop iterations. The resulting performance was poor, due to load imbalance and scalability limitations because we were touching global data unnecessarily. It is better to attempt to add the tuple to the queue, or to get permission to clear out the queue ourself if it's full, rather than executing unrelated work.

Note that rather than directly checking thread local values, we indirectly call `isFinished()` and `isSuspended()` to determine if processing should continue. Operator threads may execute `reSchedule()`; `schedule()`, `findWorkBlocking()` and `findWorkNonBlocking()` can only be executed by scheduler threads. Because the PE does not have full control over operator threads, they do not have the appropriate thread local values. Consequently, `isFinished()` and `isSuspended()` look at the appropriate thread local value if available, and fall back to the global value if not.

4.1.5 Discussion

The use of the free list—a linear walk, pop from the back, push to the front—is simple, almost to the point of seeming naive. But the scheduling policy it implements is roughly Least Recently Used. Our design was pushed in this direction because of its dependence on lock-free data structures to scale. But, prior work by Sahin [17] showed that for a streaming scheduler, LRU does perform well. At the least, it exhibits no pathological behavior in their experiments.

We have simplified our presentation of the algorithms in several ways. In Streams, our scheduler also handles punctuations. We actually use fixed size lock-free queues, which have a slightly different interface. It is possible to use non-fixed sized lock-free queues in our algorithms, but we would need some other mechanism to limit memory growth and induce back-pressure. Such a mechanism could be a `size` field in `SPSCEnforcer` that was updated on pushes and pops. It would not represent the "true" size of the operator input port queue, as it would not be updated atomically with the lock-free pushes and pops. Rather, when the `size` is over a threshold, it would indicate the queue is "full enough", and cause a call to `reSchedule()`.

Our single-producer, single-consumer queue is not actually from Boost.Lockfree. Our implementation is specifically designed to avoid needing dynamic memory allocation by reusing the same memory in the queue. Our implementation is pushed in this direction because our tuples are stack-allocated. In order to save tuples for later processing, we must copy the entire tuple. In systems where tuples are dynamically allocated by default, operator queues could instead be multiple-producer, single-consumer. Early versions of our implementation did use such operator queues, but that also required dynamic memory allocation on each tuple copy. Avoiding the dynamic memory allocation proved to be more important than allowing multiple producers.

In Streams we track which scheduler threads are "active." Operators can contain arbitrary user code; operators are allowed to *never* return. For example, they can block indefinitely, or effectively become source operators where they generate tuples and only return at PE shutdown. Scheduler threads set a flag when looking for work to indicate it is not stuck in operator code.

4.2 Elasticity

The scheduler algorithms were designed so that the PE could add and remove scheduler threads at runtime. The purpose of adding and removing threads at runtime is so that the PE can automatically determine the number of threads that maximizes throughput, saving users from figuring it out. We adapt our approach from prior work on elasticity in a streaming context [7, 18, 19]. The main idea in our elasticity approach is in recognizing performance trends, and trusting observed performance data.

```
struct ThreadRecord {
  uint64_t lastTime;
  double firstThput;
  double lastThput;
  bool trusted;
};
vector<ThreadRecord> recs;
int levelBelow = 0;
int level = 1;
int levelAbove = 2;
uint64_t time = 0;
```

Figure 7: Elasticity data structures.

4.2.1 Data Structures and Initialization

Figure 7 shows the data structures and persistent values used in our elasticity algorithm. We define a *thread level* as when the PE uses that number of threads to execute operator input ports. For every thread level, we store a `ThreadRecord` in the vector `recs` (the 0th element is unused). In a `ThreadRecord`, we need to remember the last time that we were at this thread level (`lastTime`), the first throughput value that we trust for this level (`firstThput`), the last throughput value that we saw for this level (`lastThput`), and whether or not we trust the data at this level (`trusted`). Initially, all levels are untrusted.

We need to remember our current thread level (`level`), and we use variables to track our levels below (`levelBelow`) and above our current level (`levelAbove`) so that we are not limited to linear functions for increasing and decreasing our thread level. Finally, we track logical time (`time`) for distinguishing when we recorded throughput observations.

4.2.2 Elasticity Algorithm

The PE periodically measures the aggregate throughput across all operators, and then calls `updateThreadLevel()` in Figure 8 with the latest throughput. The period of time between changing the thread level must be long enough for the change to have made a difference in throughput, but short enough to be adaptive to changes in workload. In Streams, we use a period of 10 seconds. The elasticity algorithm performs three steps:

1. Determine if the workload has changed; if it has, record that all prior observations are not trustworthy.
2. Record the latest observations.
3. Determine if the latest throughput implies that we should change the thread level.

We do not present `changeInLoad()`; it is similar to Algorithm 3 in Gedik, et al. [7]: given a throughput, historical throughputs at that level and a threshold, determine if the new throughput is different enough to consider it a change in workload. If it is, then we indicate that we not trust any of our previously recorded data. The threshold, `SENS`, is also used in `trendBelow()` and `trendAbove()`. Values closer to 0 favor being more sensitive to detecting changes, and values closer to 1 favor stability. In Streams, the value is 0.05, or reacting to differences of more than 5%.

After recording the current observations, we determine if they indicate we should change our thread level. Our algorithm assumes that the throughput versus thread level performance curve will have three phases: improvement as the thread level increases; a thread level that maximizes throughput; and past that thread level, throughput either plateaus or degrades. Our goal is to find the thread level that maximizes throughput. We use observed throughput to construct this performance curve at runtime. The following rules guide us towards finding the maximum point of this curve:

1. If relative performance between this and the level below indicates an upward trend, and we do not trust our data from above, increase the thread level.
2. If relative performance between this and the level above indicates an upward trend, increase the thread level.
3. If this thread level is 1, and we do not trust data from above this level, increase the thread level.
4. If we do not trust the data below this level, then decrease the thread level.
5. If there is no performance trend going from below to this level, then decrease the thread level.

```
int updateThreadLevel(double thput) {
  if (changeInLoad(thput))
    for (int i = 0; i < recs.size(); ++i)
      recs[i].trusted = false;

  recs[level].lastTime = time++;
  recs[level].lastThput = thput;
  if (!recs[level].trusted)
    recs[level].firstThput = thput;
  recs[level].trusted = true;

  if (((trendBelow(thput) && !trustAbove()) ||
       trendAbove(thput) ||
       (level == 1 && !trustAbove()))
      &&
      isCPUUsageAcceptable())
    increaseLevel();
  else if (!trustBelow() || !trendBelow(thput))
    decreaseLevel();
  return level;
}

bool trendBelow(double thput) {
  if (level == 1)
    return false;
  if (!recs[levelBelow].trusted)
    return false;
  if (thput > recs[levelBelow].lastThput &&
      (thput - recs[levelBelow].lastThput >
       SENS * recs[levelBelow].lastThput))
    return true;
  return false;
}

bool trendAbove(double thput) {
  if (!recs[levelAbove].trusted)
    return false;
  if (recs[levelAbove].lastThput > thput &&
      (recs[levelAbove].lastThput - thput >
       SENS * thput))
    return true;
  return false;
}

bool trustBelow() {
  if (level == 1) return false;
  return recs[levelBelow].trusted;
}

bool trustAbove() {
  if (level == recs.size() - 1) return false;
  return recs[levelAbove].trusted;
}
```

Figure 8: Elasticity algorithm.

6. If none of the above conditions are met, the thread level does not change.

These rules will cause the thread level to converge on the level at which there is performance improvement below, no performance improvement above, and we trust the data both above and below. That thread level maximizes throughput.

4.2.3 Discussion

We intentionally have a bias to seek out thread levels we do not trust so we can construct our performance curve. We also have a bias to seeking upwards before seeking downwards. However, our bias towards seeking upwards will not cause us to go too far in the direction of performance degradation, because at that point, we will have evidence of it. The special-case at thread level 1 is needed to kick-off the search process. Finally, combining the first step of `updatedThreadLevel()` which untrusts data if we detect workload changes, with the exploration both up and down if we don't trust data, will cause us to find new settling points after workload changes.

So far, all of our elasticity discussion has had the goal of maximizing the performance of a single PE. But a secondary goal is to not oversubscribe a single system—multiple, competing greedy actors can overload a system, harming performance and reliability for all. We expect multiple PEs to run on the same host, and we use two mechanisms to prevent an oversubscribed system. First, the thread level for a PE cannot increase past the number of available logical processors that the PE can execute on. Second, before increasing the thread level, we check `isCPUUsageAcceptable()`. If it is not, even if we have evidence that increasing the thread level will improve performance, we do not increase the thread level. The function can be implemented using any mechanism that observes total system usage. In Streams, we use `/proc/stat` to calculate total system CPU usage. If it is greater than 80% of system capacity, we do not increase further.

Our scheduling discussions in Section 4.1.5 introduced active threads. Keeping track of which threads are active matters when we try to suspend them: if they are not active, they cannot be suspended. After a measuring period, we always check if threads that were supposed to be suspended actually were suspended. If they were not, we mark such threads as unsuspendable, and put off making any changes to the thread level until we have a measurement period where all actions that were supposed to happen did happen.

Finally, there is a minimum thread level for each PE. In order to avoid deadlock introduced by our scheduler, we have to ensure that the number of scheduler threads never drops below one more than the maximum number of input ports that a single operator has. Operators are allowed to block indefinitely; some standard operators such as `Gate` and `Switch` depend on this behavior in their implementations, as does the protocol for establishing guaranteed tuple processing in Streams [14]. By ensuring that we always have more threads than the number of input ports for a particular operator, we ensure that our scheduler will not cause deadlock. Note that we cannot guarantee that a PE will not deadlock; because the programming model allows users to write arbitrary blocking operators, and the stream graphs can have cycles, it is always possible. Our aim here is to prevent deadlock introduced by our scheduler.

5. Experimental Results

Our experiments test three claims. First, we test the claim that our operator scheduler scales as the number of threads increases. Second, we test the claim that our elasticity algorithm discovers the thread level with the best throughput. Finally, we test our claim that the dynamic operator scheduler is a reasonable default, with performance between the two extremes of manual and dedicated threading.

All experiments have a source operator that generates tuples as fast as the downstream operators can process them. The cost of processing each tuple is fixed in a run; tuple processing cost is measure in floating point operations. We vary the tuple processing cost across different runs. We fix the total amount of operators in each experiment to 1,000. We vary the graph structure so that the experiments have varying amounts of data and pipeline parallelism. During each run, we measure total application throughput every 5 seconds as seen by the sink operator. All runs were repeated 5 times; the reported throughput is the average, and the error bars are the standard deviation.

In all experiments *dynamic static* uses our operator scheduler, but with a fixed number of threads. We vary the number of threads under dynamic static to explore the performance curve as the number of threads increases. Our elasticity algorithms are active under *dynamic elastic*. Because the measured throughput will change as the elasticity algorithms explore different thread levels, we only measure the final 5 samples, or 25 seconds, to represent the level the elasticity algorithms have settled on. Under *manual*, a single thread executes all operators, and under *dedicated*, a different thread executes each operator. Note that under dedicated, there are 1,000 threads active on the system.

We present experiments on two systems, both running RedHat Linux 7.1, kernel version 3.10.0. Our *Xeon* system has 4 Intel Xeon E7-8880 v4 processors at 2.2 GHz. Each processor has 22 cores, and each core is 2-way SMT, giving 176 logical cores. Our *Power8* system has 2 IBM Power8 8247-22L processors at 3 GHz. Each processor has 12 cores, and each core is 8-way SMT. But, in our system, one core has been disabled, so the system has a total of 184 logical cores. All experiments use IBM Streams version 4.2.0.1.

5.1 Pure Pipeline

The top two rows of Figure 9 are the results for a 1,000 operator pipeline. The tuple costs in floating point operations per tuple are 1, 100 and 1,000, from left to right. Deep pipelines present an enormous amount of inherent parallelism. The only limitation in exploiting the inherent parallelism comes from the overheads from the underlying runtime system.

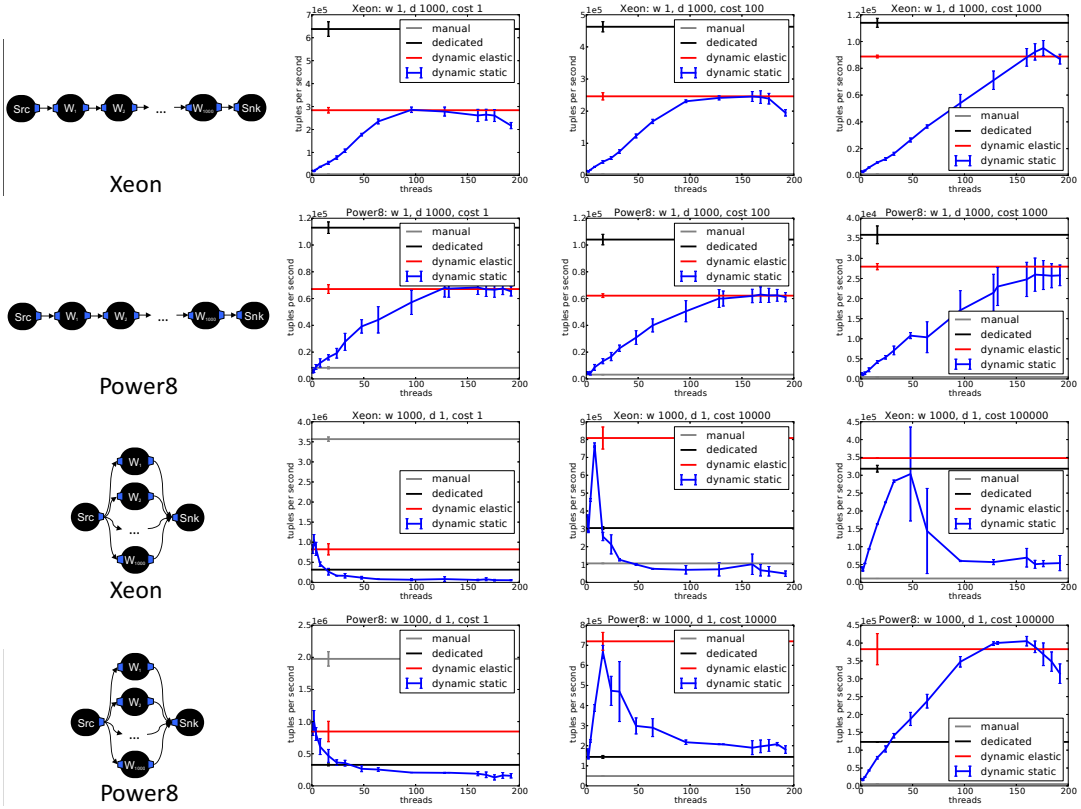

Figure 9: Xeon and Power8 results for 1,000 operators in pure pipeline and data parallel graphs.

Both architectures present similar trends. The manual model, with a single thread, always performs the worst: a single thread has to execute each operator in the pipeline, in order. The dedicated model always performs best: even though the system is over-subscribed with 1,000 threads, all threads communicate exclusively with single-producer, single-consumer lock-free and explicitly synchronization-free queues. The system has to perform many context switches for each thread to be scheduled—over 10 million for dedicated, versus about 160,000 for dynamic. But those context switches are amortized over many operator executions. In the dedicated case, each thread only directly communicates with two other threads, rendering most threads independent of each other. When any particular thread is scheduled, it is likely to have either production or consumption work. The dynamic model, however, spends proportionally more time pushing tuples into queues and looking for work. While the dynamic model has been designed to avoid thread synchronization, it still does sometimes require it. The dedicated model is able to avoid it entirely.

Note, however, that in this and all other experiments, because the dedicated model massively over-subscribes the system, it is not a realistic option for this number of operators in production applications. The load average on these systems was as high as 5,000; for the Xeon system, 176 is fully subscribed, and for the Power8 system, 184 is fully subscribed. Anecdotally, even performing interactive tasks through the shell on such systems suffers high latency.

The dynamic model presents an acceptable middle ground. Performance scales with the number of threads, and the elasticity algorithms are able to find the appropriate thread level automatically. Note that as the tuple cost increases, the performance gap between dedicated and dynamic shrinks from between 60–40% to about 25%. The performance difference comes from thread synchronization, even if we have minimized it. As the cost of the work increases, the relative cost of the synchronization decreases.

5.2 Pure Data Parallel

The bottom two rows of Figure 9 are the results for a 1,000 width data parallel split. The tuple costs in floating point operations per tuple are 1, 10,000 and 100,000, from left to right. Unlike our pure pipeline experiments, all 1,000 data parallel worker operators communicate directly with the `Snk` operator. The `Snk` operator has local state to track the number of processed tuples, and this state must be protected by a lock—but only when multiple threads will potentially access the data. (SPL handles this state protection and lock elision automatically [20].) Because of these structural differences, the pure data parallel experiments exhibit different trends than the pure pipeline experiments.

When the cost is 1, there is no effective parallelism to exploit: it is much faster to execute all 1,000 worker operators sequentially, as it avoids all thread synchronization

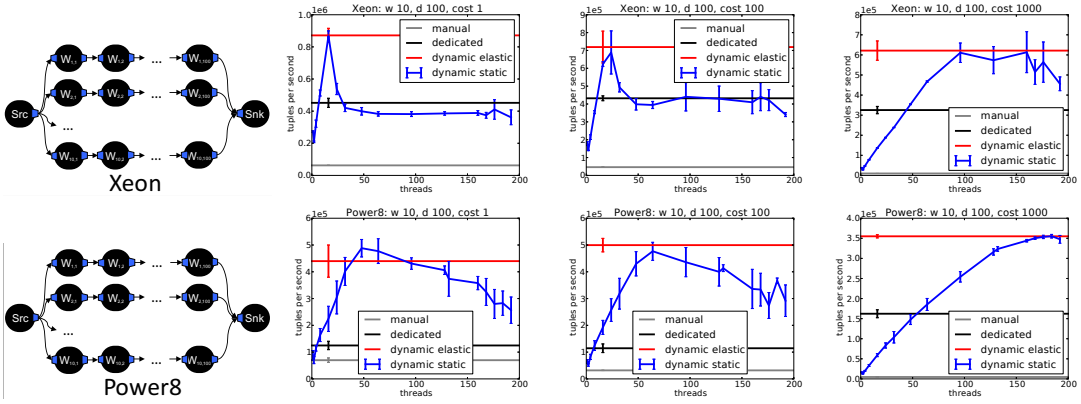

Figure 10: Xeon and Power8 results for 1,000 operators in a mix of pipeline and data parallel.

costs. The dedicated model incurs significant overhead as all threads converge on `Snk`. Unlike in the pure pipeline case, in the pure data parallel case, each thread is likely to interfere with each other thread. Any individual thread only explicitly communicates with two other threads, but all threads converging on `Snk` causes an enormous amount of interference. Because the threads are not independent, when any particular thread is scheduled, it is unlikely to have work to do. The result is many more context switches and causing most of the time to be spent spinning inside the kernel. The dynamic model fares better when it has only a few threads, but even with one operator scheduler thread, there are two total threads, as the `Src` operator also has a thread. The elasticity algorithms, however, correctly discovers that very few threads are best.

As the cost increases, the relationships flip. The cost of locking the `Snk` operator is significantly less than the cost of processing a tuple. However, the dedicated model's fixed 1,000 threads is still far from optimal; it still spends a large fraction of its execution time on spin locks in the kernel. The elasticity algorithms are able to discover that the best number of threads are about 8–10 threads on the Xeon system, and 16–24 threads on Power8.

5.3 Data Parallel and Pipeline

The experiments in Figure 10 show a modest degree of data parallelism (a width of 10) with a substantial amount of pipeline parallelism (a depth of 100). These parameters are a more realistic scenario than the extremes of pure data and pipeline parallelism. Under this more realistic scenario, dynamic threading is always the best option across both systems and all tuple costs. Manual is worst because it is never able to take advantage of pipeline parallelism. Dedicated performs better than manual, but never close to dynamic because while it is able to take advantage of both data and pipeline parallelism, it still has more opportunity for thread interference. The elasticity algorithms are able to dynamically figure out when there are too many threads, while the dedicated model is always stuck with too many threads.

5.4 Elasticity Details

Our prior elasticity results showed end-to-end performance over many runs. Such results demonstrate that our elasticity algorithms perform well in a variety of scenarios, but they do not show behavior. Figure 11 shows the elasticity algorithms during individual runs. The x-axis is time into the experiment in seconds; the left y-axis is throughput in tuples per second; the right y-axis is the number of active threads. Each graph on the same row shows a different run with the same stream graph and tuple cost. We present a subset of the spectrum that was explored in previous sections.

Note that the throughput values on these graphs are larger than in previous sections. The throughputs in previous sections are end-to-end application throughput, as measured at the sink. The throughputs reported in Figure 11 are all tuples processed across all operators in the PE, which will be necessarily larger than throughputs measured at one operator.

Pure Pipeline: The top two rows of Figure 11 are the elastic runs for Xeon and Power8 with a pure pipeline of 1,000 operators and a trivial cost of 1. All runs exhibit the same trend of a quick ramp-up of threads, then backing off to a lower thread level. On Xeon, the lower thread level goes between 72–132 threads. While this is a large variation in thread level, it is not a large variation in throughput (seen in both these figures and in the top of Figure 9). The Power8 runs settle on a tighter bound of 128–160, which is again within a tight throughput bound, and consistent with the static results from Figure 9.

Pure Data Parallel: The middle two rows of Figure 11 show pure data parallel runs with 1,000 operators. The Xeon runs use a cost of 10,000 floating point operators per tuple. Across all runs, the elasticity algorithms explore up to 16 threads, experience significant performance degradation, then back off to 8–10 threads. In contrast, the Power8 runs use a more expensive cost of 100,000 to show how the elasticity algorithms behave with the same stream graph, but with a different cost. These runs show a large variability and repeated oscillation between 130–184 threads. This oscillation is caused by the high variability in the measured throughput—even during time periods where the thread level

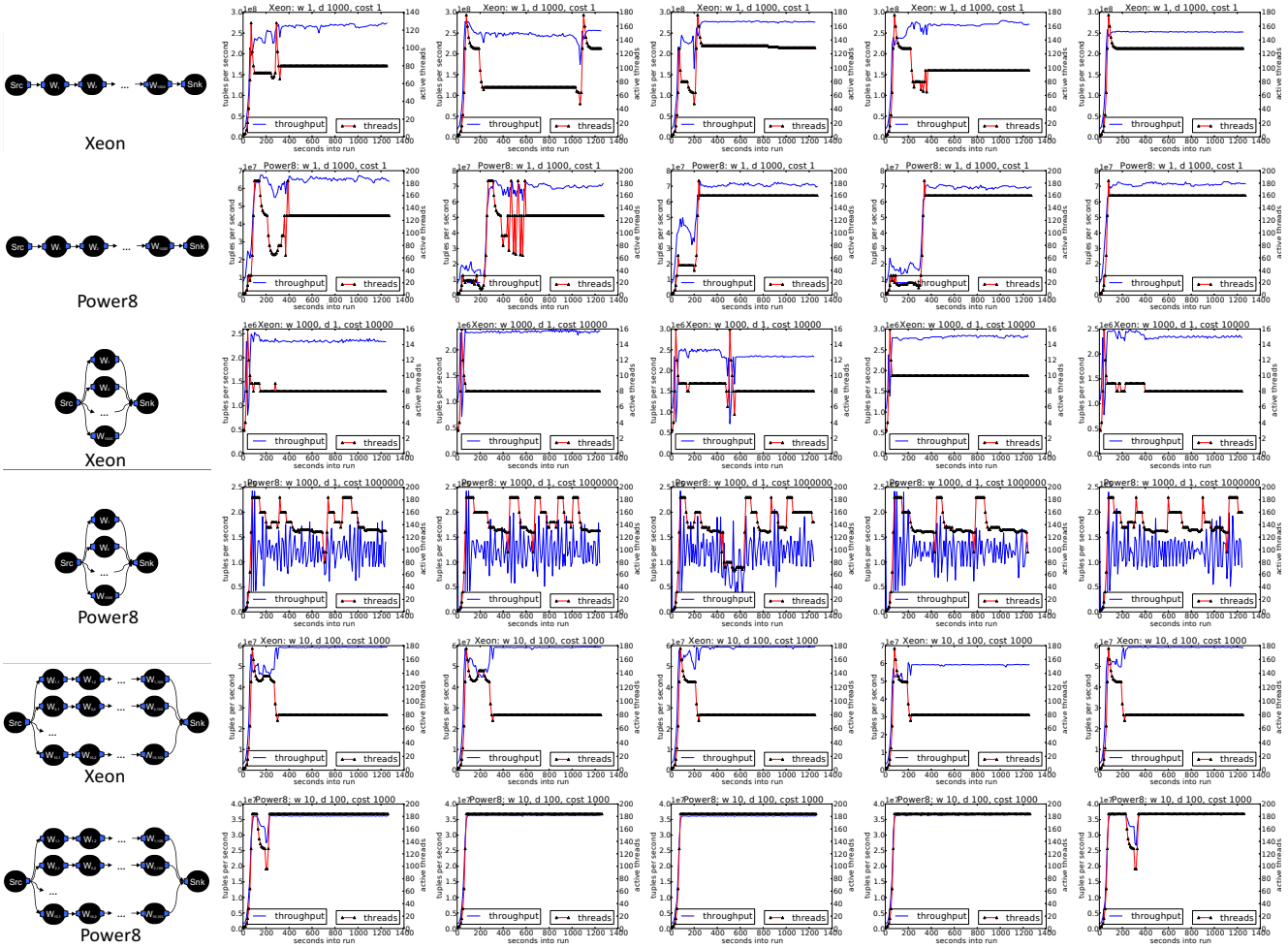

Figure 11: Individual elasticity runs, comparing throughput and the number of active threads.

is stable, the throughput still changes significantly. The variability is caused by the stream graph's sensitivity to too many threads. As discussed in Section 8, our elasticity algorithms react to differences of more than 5%. If we tuned the sensitivity so that it did not overreact to the changes in this run, it would not be sensitive enough in other scenarios. A better alternative is designing a mechanism for remembering some history. After detecting throughput changes, our elasticity algorithms currently wipe away all historical data. We will explore mechanisms for remembering some data to avoid such oscillations.

Data Parallel and Pipeline: The bottom two rows of Figure 11 are the elastic runs for Xeon and Power8 with a data parallel width of 10, a pipeline depth of 100, and a cost of 1,000 floating point operations per tuple. Because of the difference in architectures, this set of experiments shows the value in elastic adaptation. On Power8, it is best to max out the system with 184 threads. However, on Xeon, the elastic algorithm also explores up to the maximum number of available threads, 176, but it quickly determines that 80 threads yields the best performance. That the same application has such different needs on different architectures shows the value of having an elastic runtime.

6. Correctness Testing

We present performance as the primary evaluation criterion in this paper. But correctness is also an important evaluation criteria, and lock-free algorithms are notoriously difficult to implement correctly. Our confidence in correctness comes from extensive testing as a part of Streams as a product.

SPL itself has close to 2,000 tests. There are also hundreds of platform and application-level tests that incidentally rely on the SPL runtime functioning properly. During development we continuously ran all tests on three versions of Linux (Red Hat Enterprise Linux 6 and 7, SUSE Enterprise Linux Server 11) on two different architectures (Intel x86-64 and Power 64-bit big and little endian). The danger of changing a fundamental part of an existing product—such as how the SPL runtime processes tuples—is that getting it wrong is catastrophic. Getting the memory model wrong and introducing race conditions lead to immediate test failure. Hundreds of threads processing millions of tuples per second means that race-conditions will out: "rare" occurrences happen frequently. But the benefit is that something so fun-

damental will be implicitly relied on in the vast majority of existing tests. An alpha and beta period with customers gave us further confidence in correctness.

We did discover surprises during testing. A few dozen of our tests relied on operators executing in a deterministic order, as it would happen with the manual threading model. Such tests were written by SPL developers who were intimately familiar with the runtime's implementation, and exploited its properties to elicit specific behavior. A lesson we draw from these tests is that developers will tend to depend on actual behavior. That dependence is a result of the programming model being less restrictive than its initial implementation. Relaxing the implementation to meet the programming model can present a danger to existing code.

We also discovered race conditions in the implementation of our consistent region protocol [14]. Running the consistent region tests under the dynamic threading model became a stress test for the protocol's implementation. While the event interleavings the dynamic threading model introduced were always legal under SPL's programming model, they were either extremely rare or impossible to achieve in the old runtime's implementation. We caught and fixed these bugs during development.

7. Conclusions

We presented the design of the dynamic, elastic operator scheduler from IBM Streams 4.2. We achieved scalability to over 100 threads by not just avoiding thread synchronization, but by not touching global data as much as possible. Modern systems that we would usually think of as one "node" are, internally, distributed systems. In order for thread-level parallelism to scale, we must think of thread communication the same as messages in classical distributed systems. We also showed the accuracy of our elastic algorithms, which applies the principle of taking measurements, but continually evaluating if we should continue to "trust" those measurements while looking for clear performance improvement trends.

Finally, we note that our ability to add a dynamic and elastic scheduler to a mature product with existing customers is thanks to the programming model. Because SPL's programming model enforces no shared state between operators, our runtime has enormous flexibility in how it executes streaming applications.

Acknowledgements

We would like to thank Kirsten Hildrum and Gabriela Jacques da Silva for providing feedback both in the early stages of this work, and much later on the paper itself. We would also like to thank Xiang Ni and the anonymous reviewers for feedback on the paper. Finally, we would like to thank Nagui Halim for his many years of support for stream processing research.

References

[1] Streaming Analytics. `https://console.ng.bluemix.net/catalog/services/streaming-analytics`. Retrieved April, 2017.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, New York, NY, USA, 1995. ACM.

[3] Boost.Lockfree. `http://www.boost.org/doc/libs/1_63_0/doc/html/lockfree.html`. Retrieved March, 2017.

[4] C++ std::atomic. `http://en.cppreference.com/w/cpp/atomic/atomic`. Retrieved March, 2017.

[5] Apache Flink. `http://flink.apache.org`. Retrieved March, 2017.

[6] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Programming Language Design and Implementation (PLDI)*, 1998.

[7] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2014.

[8] IBM Streams Demo. `https://github.com/IBMStreams/streamsx.demo.logwatch`. Retrieved March, 2017.

[9] IBM Streams Samples. `https://github.com/IBMStreams/samples`. Retrieved March, 2017.

[10] Heron. `https://twitter.github.io/heron`. Retrieved March, 2017.

[11] Martin Hirzel, Henrique Andrade, Buğra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, and Kun-Lung Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4), 2013.

[12] Martin Hirzel, Scott Schneider, and Buğra Gedik. SPL: An extensible language for distributed stream processing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(1), March 2017.

[13] IBM Stream Computing. `http://www.ibm.com/analytics/us/en/technology/stream-computing`. Retrieved March, 2017.

[14] Gabriela Jacques-Silva, Fang Zheng, Daniel Debrunner, Kun-Lung Wu, Victor Dogaru, Eric Johnson, Michael Spicer, and Ahmet Erdem Sariyuce. Consistent Regions: Guaranteed Tuple Processing in IBM Streams. In *Very Large Data Bases Conference (VLDB)*, 2016.

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

[16] OpenMP. `http://openmp.org/`. Retrieved March, 2017.

[17] Semih Sahin. C-stream: A coroutine-based elastic stream processing engine. Master's thesis, Bilkent University, June 2015.

[18] Scott Schneider. The ElasticLoadBalance Operator. `https://developer.ibm.com/streamsdev/2015/01/27/elasticloadbalance-operator`, 2015.

[19] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *IEEE International Parallel and Distributed Processing Symposium*, 2009.

[20] Scott Schneider, Bugra Gedik, and Martin Hirzel. Language runtime and optimizations in IBM Streams. *IEEE Database Engineering Bulletin*, 38(4), 2015.

[21] SPL Reference. `http://www.ibm.com/support/knowledgecenter/SSCRJU_4.2.0/com.ibm.streams.ref.doc/doc/spl-container.html`. Retrieved March, 2017.

[22] Apache Storm. `http://storm.apache.org`. Retrieved March, 2017.

[23] StreamsDev: IBM Streams Developer Community. `https://developer.ibm.com/streamsdev`. Retrieved March, 2017.

[24] Yuzhe Tang and Bugra Gedik. Auto-pipelining for data stream processing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 24(11), 2013.

[25] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, New York, NY, USA, 2014. ACM.