RC25567 (WAT1510-067) October 21, 2015
Computer Science

IBM Research Report

Dynamic Load Balancing for Ordered Data-Parallel Regions in Distributed Streaming Systems

Scott Schneider, Joel Wolf, Kirsten Hildrum*, Kun-Lung Wu, Rohit Khandekar[1]
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

*Currently at Google
[1]Knight Capital Group

Dynamic Load Balancing for Ordered Data-Parallel Regions in Distributed Streaming Systems

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

Joel Wolf
IBM Research
jlwolf@us.ibm.com

Kirsten Hildrum*
IBM Research

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

Rohit Khandekar
Knight Capital Group
rkhandekar@gmail.com

ABSTRACT

Distributed stream computing has emerged as a technology that can satisfy the low latency, high throughput demands of big data. Stream computing naturally exposes pipeline, task and data parallelism. Meeting the throughput and latency demands of online big data requires exploiting such parallelism across heterogeneous clusters. When a single job is running on a homogeneous cluster, load balancing is important. When multiple jobs are running across a heterogeneous cluster, load balancing becomes critical. The data parallel regions of distributed streaming applications are particularly sensitive to load imbalance, as their overall speed is gated by the slowest performer. We propose a dynamic load balancing technique based on a system artifact: the TCP blocking rate per connection. We build a function for each connection based on this blocking rate, and obtain a balanced load distribution by modeling the problem as a minimax separable resource allocation problem. In other words, we minimize the maximum value of these functions. Our model achieves local load balancing that does not require any global information. We test our model in a real streaming system, and demonstrate that it is able to detect differences in node capacities, determine the correct load distribution for those capacities and dynamically adapt to changes in the system.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

Keywords

stream processing; automatic parallelization; dynamic load balancing

1. INTRODUCTION

The amount of data that must be processed and analyzed is increasing past the ability of conventional means to handle it, a phenomenon commonly referred to as *big data*. On a systems level, processing big data online requires highly parallel runtimes that can maintain low latencies and high throughput. For application developers and system administrators, such systems must provide meaningful abstractions that allow writing high performance, massively parallel applications which can be easily deployed to large, heterogeneous clusters.

Distributed stream computing has emerged as a technology which can meet these needs, with many examples from industry and academia [20, 25, 1, 16, 3]. Developers can express applications as dataflow graphs, which naturally exposes the inherent pipeline, task and data parallelism in the solution. The distributed runtime system is then responsible for executing the application in an efficient manner.

In IBM Streams [14], developers express their dataflow graphs in SPL [12], using the high-level abstractions of operators, streams and tuples. Operators are the logical unit of computation which process structured data items called tuples. Operators within an application communicate through streams of tuples. Through these abstractions, application developers are saved from having to directly deal with the complexity of distributed, heterogeneous clusters. Instead, they can depend on the high performance runtime that is a part of Streams, which can automatically exploit the various levels of parallelism exposed by the stream programming model.

The Streams runtime must be able to dynamically adapt to a variety of cluster types and loads. The runtime executes operators in processing elements (PEs). Of critical importance are the data parallel regions, where the runtime replicates these PEs. Inside data parallel regions, each PE processes a subset of the total tuples. These subsets are determined by a splitter, which is responsible for routing tuples to parallel worker PEs. In the presence of imbalanced capacities among the compute nodes, the splitter must balance the load among worker PEs to maintain high performance.

Dynamic load balancing in a distributed streaming system has several unique challenges. The data parallel regions must maintain sequential semantics [18]. In a streaming context, sequential semantics means that tuples must exit the data parallel region in the same order that they would have if they had all been processed by a single PE. Enforcing sequential semantics requires performing an in-order merge as tuples exit the parallel region. As a result, the performance of the entire region is gated by the performance of its slowest worker. Because of this merge, all connections will see the same throughput, which means that per-connection throughput at the splitter is not a useful metric for our problem. Instead, we must find another metric to infer PE capacity.

As we are in a distributed system, the splitter can only

*Kirsten Hildrum is now at Google, and can be reached at hildrum@google.com.

communicate with its worker PEs over TCP connections. We use an artifact of the system itself as our fundamental metric: each connection's blocking rate.

Calculating the blocking rate is cheap, which means that we are not harming performance while trying to improve it. However, the blocking rate itself does present an additional challenge: only one connection is likely to block during a sampling period. Hence, we will receive very little data during each such period.

We have designed, implemented and tested a model that overcomes all of these challenges. First, we build a blocking rate function, F_j, for each connection j. The value of $F_j(w_j)$ yields the amount of blocking that connection j either experienced or is predicted to experience when it is given a fraction w_j of the total tuples by the splitter. We model load balancing as a minimax separable resource allocation problem where we minimize the maximum across all F_j such that $\sum_{j=1}^{N} w_j = 1$ while respecting any minimum and maximum change constraints in w_j per connection. The work of Diao et al. [5] showed that such systems can be accurately modeled with either control theory or optimization techniques.

Our experimental results show that this technique works well in practice as a part of a distributed streaming system. We show that our model can detect differences in capacity due to both exogenous load and imbalance caused by heterogeneous compute nodes. It achieves stability with both load imbalance and equal capacity. Further, through an exploration mechanism, we show that it adapts to changes in the system.

This paper makes the following contributions:

- The blocking rate metric, both in how we derive it from our system and in how it behaves. As far we know, this paper is the first to propose using the blocking rate of the underlying transport layer in a distributed system to perform load balancing.
- Analysis and description of a model based on the blocking rate.
- Experiments in a real system which demonstrate the stability and correctness of our model.

2. DISTRIBUTED STREAM COMPUTING

Our platform is a research prototype of IBM Streams [14], a high performance streaming system which executes asynchronous, distributed streaming applications. The programming language for Streams is SPL [12], a language that naturally exposes pipeline and task parallelism.

SPL applications are expressed in terms of *operators* and *streams*, where the operators express a computation, and different operators are connected by streams. Each stream contains *tuples* of the same type. Tuples are structured data items, similar to a row in a relational database. Operators consume a tuple from an input stream, perform some computation on it, then potentially emit a result tuple on an output stream, to be processed by a downstream operator.

Different operators can execute different tuples in parallel. Hence, by arranging operators in simple chains, developers can naturally express pipeline parallelism. If developers split their streams, and send the same tuples to different operators, they have expressed task parallelism. The compiler and runtime system in our research prototype of Streams [1] are further able to determine where there are *data parallel regions*.

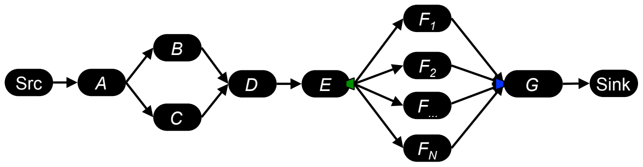


Figure 1: Sample streaming application running on IBM Streams.

Our runtime system executes collections of operators in a *Processing Element*, which we will refer to as a *PE*. Each PE maps to an OS process, and can be executed on different physical machines in a network.

Figure 1 shows a sample application as it would execute on the Streams runtime. PEs are represented as rounded squares, and streams are represented by the arrows connecting them. The arrows point in the direction of the flow of tuples, hence, in Figure 1, the tuples flow from *Src* to *Sink*. All of the PEs, A–G, execute in parallel, exploiting pipeline parallelism. PEs B and C are an example of task parallelism, because they receive the same tuples, yet perform different operations.

PEs F_1–F_N are a data parallel region. We assume that all copies of F are stateless; the other PEs in the application may have state. In our context, stateless means that the PE does not "remember" anything about each tuple it processes; stateless PEs are pure functions that, given a particular input tuple, will always produce the same output tuple. There is a *splitter* at E which splits the tuple stream. Each F_i receives only a subset of the total tuples, thus exploiting data parallelism. There is a *merger* before G. The merger ensures that the tuples, which may have been processed out-of-order, are put back in-order. Said differently, the merger is required to maintain sequential semantics: Tuples must exit the parallel region in the same order they would if there was only one replica of F.

Problem Statement

Assuming that all F_i are stateless PEs, we want to balance the load across them in the presence of load external to the application and/or assignments to heterogeneous processors. In the context of our streaming system, load balancing means that the splitter must decide which F_i to send each tuple to based on the load and capacity of the node assigned to that PE. We want to accomplish load balancing locally at the splitter, without querying the worker PEs about their status. Instead, we must only use information locally available on the machine on which the splitter is executing—in distributed systems, solutions that do not require global information and control are easier to implement and scale. Finally, the means by which we accomplish load balancing must not itself negatively impact performance.

3. BLOCKING TIME AND RATE

The metric that our model uses is the *blocking rate* per TCP connection, which we calculate from the *cumulative*

[1]The production version of IBM Streams enables programmers to annotate their operator invocations to indicate parallel regions. Because programmers manually create parallel regions, the production version of Streams does not maintain tuple order.

blocking time. In this section, we explain how we measure cumulative blocking time, and how we use it to calculate the blocking rate.

The data transport layer establishes a TCP connection for every connected PE. The splitter uses these TCP connections to send all tuples to the PEs in the parallel region. If the parallel worker PEs process tuples slower than the splitter sends them, then eventually an attempt to send a tuple on a TCP connection will block. When a TCP send blocks, we record how long it blocks.

We use two mechanisms to record the blocking time. First, when sending a tuple, we issue a send system call on a TCP socket with the flag MSG_DONTWAIT. This flag ensures that if the kernel would block while trying to write data into that socket's buffer, it immediately returns with a value indicating so. If a send would have blocked, we record that, and then issue a select system call on that socket, passing in a valid time-out object. When the socket's buffers are free so that it can send data, the select call returns. On Linux systems, select also writes the amount of time the process was blocked into the time-out object. We maintain a counter for each connection which tracks the cumulative blocking time on that connection. After each call to select, we increase this counter by the amount of time the system call blocked. In this manner, we are able to track the cumulative blocking time per each TCP connection.

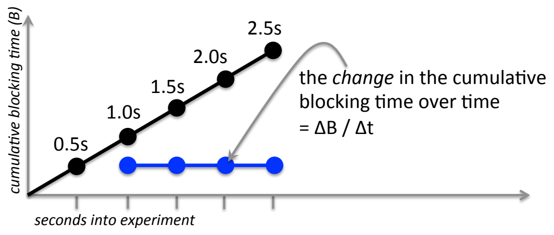


Figure 2: Idealized calculation of per-connection blocking rate.

Figure 2 shows the idealized behavior of the cumulative blocking time over time for a particular connection. The cumulative blocking time is reported every second, and constantly increases until it is periodically reset by the data transport layer. These increases are not actually useful for our model per se, but the rate of increase is. So to calculate the blocking rate over time, we periodically sample the cumulative blocking time from the data transport layer. We then take the differences between subsequent cumulative blocking values to obtain estimates of the blocking rate over that period. These turn out to be quite stable for a particular system load, and can be thought of as first derivatives of the cumulative blocking time with respect to time, as shown in idealized form in Figure 2. We use an appropriately smoothed single blocking rate value in our model.

4. DESIGN CHALLENGES

Each of the following challenges are present in a distributed streaming system. They informed the direction we took in order to model the problem, as the model needed to address them directly, or be robust to them.

4.1 In-order merges

Most parallel regions have an in-order merge at the end, [2]

[2]Some parallel regions end without merges, in parallel sinks.

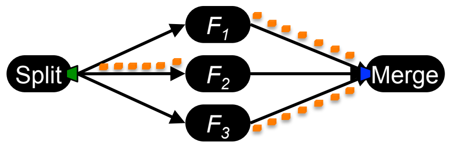


Figure 3: Parallel region with three PEs. The boxes on the edges are queued tuples, implying that F_2 is slower than F_1 and F_3.

as shown in Figure 3. Even though the splitter sends tuples to worker PEs to be processed in parallel, when those tuples exit the parallel region, they must do so in the same order they arrived at the splitter. In-order merges are required to maintain sequential semantics. That is, the tuples should appear downstream in the same order that they would if the PEs in the parallel region were not replicated and executed in parallel.

As a result of the in-order merge at the back of parallel regions, the splitter's connections in the front are not independent. Figure 3 demonstrates the implication of the merge. The boxes on the edges represent tuples that have been sent but not yet processed by the receiving PE. In our example, F_2 is much slower than the other parallel worker PEs. As a result, even though workers F_1 and F_3 have completed processing many tuples, those tuples are stuck in the merger's queues—the merger can only send tuples from F_1 and F_3 downstream when it has received the corresponding tuples from F_2 that preserve the sequential order.

The presence of the in-order merge makes load balancing even more important, as the overall performance of a parallel region is gated by its slowest performer.

4.2 Drafting

One difficulty with using the blocking rate as a metric is the phenomenon we call *drafting*. The splitter has a single thread of control; the same kernel thread is used to send tuples to all worker PEs in the parallel region. As we shall see, this implies that during a measurement period only one connection is likely to experience blocking, even if all of the connections can handle the same amount of load.

To understand why drafting happens, consider the simple example of a round-robin splitter sending tuples to three parallel workers, each worker capable of handling the same amount of load. Eventually, the splitter may block when trying to send a tuple on its TCP connection to a parallel worker. Suppose, for example, that connection 2 blocks. While the splitter is blocked waiting for the buffers for connection 2 to clear, all of the other buffers for the other connections also have the opportunity to clear. When the buffers for connection 2 are finally clear, the splitter successfully sends a tuple on connection 2. The splitter next sends a tuple on connection 3, and it is very unlikely to block on that connection—any amount of time that is sufficient for the buffers of connection 2 to clear is sufficient for the buffers of connection 3 to clear, given equal capacity. The same is true for connection 1, next in the round-robin order. Eventually, as the splitter once again distributes tuples, the probability of blocking will start to *increase*. However, connection 2 has had the least amount of slack-time since the first tuple being sent, so when the splitter does block, it is most likely to block on connection 2. The splitter and connection 2 are in a synchronized rhythm, and we call connection 2 the *draft leader*. The point is that the draft leader is likely to change less frequently than the measurement periods.

This phenomenon is similar to how cyclists and race-car drivers will draft behind a leader, with the leader bearing the brunt of the drag.

Drafting presents a challenge to load balancing. Suppose we only look at instantaneous information, and we observe a connection that experiences a high blocking rate. It is impossible to determine if that connection has a lower capacity than its siblings, or if it is merely the draft leader. This fact implies that any attempt to model our system must have a notion of history to overcome this limitation in the available data.

4.3 Per-connection throughput

Most people's intuition is to use throughput as the fundamental metric, but throughput is not useful for our problem. If, say, the splitter is distributing tuples by basic round-robin, then the throughput on all connections will be the same. If the splitter sends 3 tuples to one connection for every 1 tuple to another, their relative throughputs will always be 3:1. This counter-intuitive result is a consequence of having to merge the tuples at the end of the parallel region.

To understand why per-connection throughput has no extra information, once again consider the situation in Figure 3, where the splitter sends the same amount of tuples to each connection. As explained in the previous subsection, the splitter has a single thread of control. This thread will periodically block when trying to send tuples to F_2. Because F_2 is slow, its TCP queues for receiving tuples will fill up, causing the splitter's TCP queues for sending tuples to that connection to also fill up. We call this phenomenon *back pressure*: in a streaming pipeline, the steady-state throughput of the entire pipeline is that of its slowest member.

In any given period of time, the splitter will spend a disproportionate amount of time blocked on connection 2. It will easily send tuples on connections 1 and 3. But even though it easily sends tuples on connections 1 and 3—it rarely ever blocks on those connections—it will send the same number of tuples to all connections in a given period of time. Hence, throughput is not useful for our problem, but blocking time is.

A different, but equally correct, implementation could instead block at the merger; it is an artifact of our implementation *where* we block. But we fundamentally have to block *somewhere* in order to maintain order. It is the requirement to maintain tuple order that causes per-connection throughput to have no information.

4.4 Blocking is a rare event

A curious consequence of how we record blocking time (as explained in Section 3) is that we actually *elect* to block. That is, we detect when a TCP send will block, and then we block anyway, just making sure to record how long we block.

The obvious question is: Why block? Why elect to do nothing? Instead, we could send tuples to the other connections, thus achieving load balancing at the data transport level. We experimented with a data transport level re-routing approach that does exactly that.

The intuition behind the re-routing approach is appealing: only send tuples to connections that can currently handle them. If a connection blocks, try sending that tuple to another connection, hence distributing the load based on current capabilities.

This intuition, however, is naive. The fundamental problem with the approach is that blocking is a late indicator of congestion. In an experiment with two PEs, where the base cost of processing a tuple is 1,000 integer multiplies, and one of the PEs is 100× more expensive than the other, the re-routing approach makes no discernible difference in throughput versus basic round-robin. It only re-routes 0.5% of the tuples. When the base tuple cost is 10,000 integer multiplies, it does make about a 20% improvement in total throughput, while re-routing about 7.5% of the total tuples. This improvement, however, is not nearly enough, and it only appears when the base tuple cost is high. We require a more general solution with larger improvements.

The reason that the data transport level re-routing approach does not work in general is that blocking is a rare event, even in the presence of hugely disparate capacities. This fact is caused by the numerous system buffers between two processes that execute on different hosts in a network. By the time a TCP connection for an overloaded PE blocks, it already has at least two system buffers worth of unprocessed tuples (locally on the splitter and remotely on the worker). Those tuples still take 100× as long to process, and because of the ordered merge at the end, overall throughput suffers.

There are two lessons we draw from this experiment. One, because blocking is rare, we have to build a model that can function with a paucity of data. Two, we need to re-route tuples *before* a low-capacity connection becomes overloaded. The data transport re-routing approach does too little, too late. It re-routes too few tuples to make a difference, and it only does so when the connection is already overloaded. It does not learn from its mistakes. We need an approach that can predict how much load a connection can handle.

5. LOCAL LOAD BALANCING

In this section we describe the local load balancing problem we are solving for each parallel PE region. First we will describe the computation of the blocking rate functions themselves. These functions are the components of the objective function we will optimize. Then we describe the load balancing optimization itself, including both the formulation and the solution algorithm. Finally, we describe the way in which we encourage periodic exploration of the problem space, in order to react swiftly to dynamic changes in the underlying scenario. Failing to do this may result in relatively static blocking rate functions and thus to less adaptation than should occur in the optimization.

Figure 4 pictorially sketches the steps we take to achieve local load balancing.

5.1 Blocking Rate Function

We begin this section by describing the construction of the blocking rate function to be used as input to our load balancing optimization. There will be one such blocking rate function F_j per connection. The x-axis of this function will correspond to potential round robin *allocation weights*[3] w_j allotted to the connection by the splitter, in units of 0.1%. In other words, the x-axis for the jth blocking rate function will consist of 1001 discrete values between 0 and 100%.

[3]The name *allocation weight* is inspired by *weighted round-robin*, where the *weights* are determined by our model.

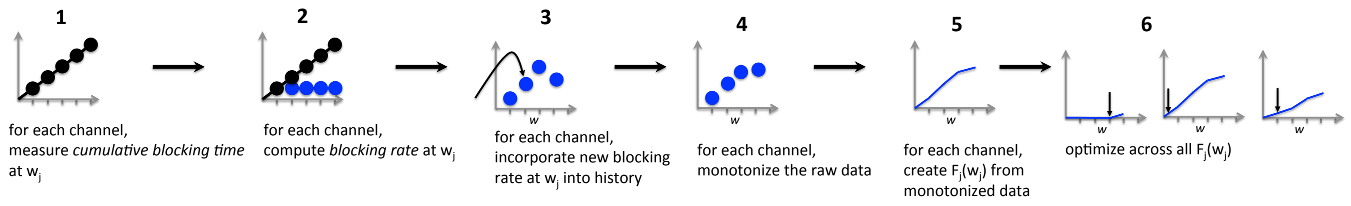

1
for each channel, measure *cumulative blocking time* at w_j
2
for each channel, compute *blocking rate* at w_j
3
for each channel, incorporate new blocking rate at w_j into history
4
for each channel, monotonize the raw data
5
for each channel, create F_j(w_j) from monotonized data
6
optimize across all F_j(w_j)
w

Figure 4: The steps to achieve local load balancing.

The y-axis will correspond to the blocking rate $F_j(w_j)$ if the connection is allocated weight w_j.

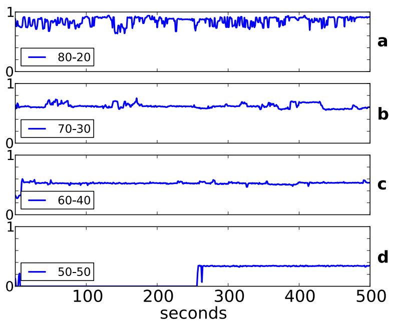


Figure 5: Blocking rates for fixed allocation weights.

To see that this function even makes sense we begin with a simple experiment designed to compare reality with the idealized example shown in Figure 2. Consider a two-connection scenario on a pair of homogeneous processing nodes. We experimented with dividing this load statically in four separate distributions. The first is an unvarying split with connection 1 getting 80% of the load and connection 2 getting the remaining 20%. In this case connection 1 was the draft leader, and connection 2 was the draftee. Figure 5(a) illustrates the blocking rate for connection 1 as a function of time. Note the stability (flatness) of this function—its behavior mimics that of Figure 2 quite closely. Figure 5(b) illustrates the corresponding connection 1 blocking rate in a 70%-30% unvarying split. Again the figure is quite flat, and notice that the blocking rate at 70% is less than that at 80%. Similarly, Figures 5(c) and (d) show the blocking rate for a 60%-40% split and a 50%-50% split, respectively. Note the monotonicity of the blocking rates across the 4 subfigures. As the allocation weight decreases from 80% to 50% the blocking rates consistently decrease as well. Note also the stability of each blocking rate over time, with the exception of Figure 5(d). What is happening here? The answer is simple. In a 50%-50% split the draft leader has become the draftee at some arbitrary point in time, and vice versa. A similar graph for connection 1 would show that this connection is now receiving blocking rate data.

Our goal now becomes to construct a single function $F_j(w_j)$ per connection j. It is important to observe that data for these functions will typically arrive infrequently. Specifically, changes in draft leaders will occur far less frequently than data collection intervals, which occur every second. At most data collection intervals this means that there will be only a single new data value for precisely one of the connections, and it will correspond to only one possible allocation weight, the current weight for that connection.

This function is derived and updated in three steps. First, new data is collected and smoothed into the existing "raw" data. (The value $(0, 0)$ is assumed.) Second, the raw data points are forced into non-decreasing order by a process known as *monotone regression* [7]. Third, the missing data points in the domain are computed via linear interpolation or extrapolation.

5.2 Load Balancing Optimization

Our problem can be formulated as a so-called *minimax separable resource allocation problem* (RAP). Specifically, we wish to minimize $\max_{1 \le j \le N} F_j(w_j)$ subject to the two constraints $\sum_{j=1}^{N} w_j = 1$ and $m_j \le w_j \le M_j$ for all $1 \le j \le N$.

Note that the objective function $\max_{1 \le j \le N} F_j(w_j)$ corresponds to the blocking rate of the weakest link among the connections, the one whose blocking rate is largest. The first constraint is the RAP constraint itself: All the traffic from the splitter must be allocated. Separability here means that each term F_j is a function of a single decision variable w_j. The second constraint provides minimum and maximum bounds for the connection allocation weights, typically incrementally from the *current* weights during each problem instance. (If there is no lower bound for connection j, then $m_j = 0$. If there is no upper bound, then $M_j = 1$.)

The optimization literature on RAPs is well established [13]. In our particular case the problem is discrete in the sense that we only consider solutions in which each allocation weight is a multiple of $r = 0.001$, in other words 0.1%. So essentially, we can say that there are $R = \frac{1}{r} = 1000$ total units of resource. Our problem is also monotone non-decreasing: We insist that $F_j(w_{j,1}) \le F_j(w_{j,2})$ whenever $w_{j,1} \le w_{j,2}$. This monotonicity should be a logical tautology, but as we have already stated that we force it to be true in the (rare) cases where the empirical data does not support it.

Minimax discrete separable RAPs with monotone non-decreasing functions can be solved exactly by a simple greedy algorithm generally attributed to Fox [8]. Consider the "matrix" $\mathcal{F}$ whose (i, j)th term is $F_j(r \cdot i)$. Each column j is monotone non-decreasing in i. Assuming the minima are described in multiples of r, we start by setting $\bar{w}_j = m_j$. At each stage we compute the column $j^* = j$ for which $\bar{w}_j + r \le M_j$ and $F_j(\bar{w}_j + r)$ is minimum. Then we set $\bar{w}_{j^*} = \bar{w}_{j^*} + r$. We repeat this greedy process until $\sum_{j=1}^{N} \bar{w}_j = 1$ or each $\bar{w}_j = M_j$. A simple interchange argument will show that this algorithm produces an optimal solution. With the proper data structures the algorithm has complexity $O(N + R \log N)$. (The number of actual iterations required is often much smaller than R.)

There do exist faster algorithms which also solve this problem exactly. For example, a scheme based on binary search by Galil and Megiddo [10] has complexity $O(N \log^2 R)$. Frederickson and Johnson [9] designed a scheme based on geometric search space reduction with complexity $O(\max(N, N \log(R/N)))$. But for our problem instances,

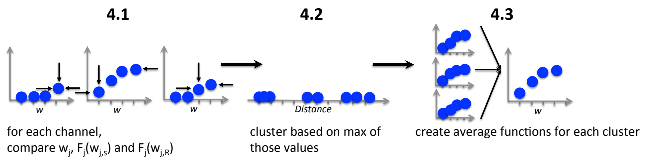

4.1
4.2
4.3
w
w
w
Distance
w
for each channel, compare w_j, $F_j(w_{j,s})$ and $F_j(w_{j,R})$
cluster based on max of those values
create average functions for each cluster

Figure 6: Clustering steps.

the greedy Fox scheme suffices because both the number of connections N and the maximum number of iterations R are modest, and due to the incremental constraints. Hence, we use it in our implementation.

5.3 Clustering

The prior local load balancing scheme works well in practice when the number of parallel connections is modest. (In our experiments, a modest number of connections is 16 or less.) However, as explained in Section 4, we effectively have a fixed amount of available data to collect. As we increase the number of connections, this fixed amount of data gets spread out more and more. Consequently, as we increase the number of connections, the amount of data available to each individual connection's function decreases. Each function then becomes less accurate, and the load balancing solution provided by the optimization process suffers.

We can address this problem by starting with a systems insight: multiple PEs may reside on the same host. If that host either experiences external load, or inherently has less processing capacity than other hosts, the impact will be the same on all of its PEs. Hence, performance is likely to be correlated per host. We take advantage of this insight by using clustering to discover groups with similar performance, and aggregating their data into a single function for the group.

Figure 6 sketches the additional steps taken to apply clustering to our local load balancing technique.

To perform clustering on the connection functions, we must first define a distance function to measure how "close" two functions are. That is, we require a function $Distance(F_j, F_k)$ which will yield 0 when F_j and F_k are indistinguishable, and some large positive value when they are "far."

In order to define our distance function, we exploit the characteristics of the predictive connection functions. Figure 7 shows three examples. The left function in Figure 7 represents a channel which does not see any blocking until it has about 0.5 of the total load, at which point it experiences low blocking. The middle function also does not experience any blocking until it has about 0.5 of the load, at which point it experiences moderate blocking. The function on the right experiences severe blocking even with 0.001 of the load.

These functions tend to have a sharp knee at a particular weight $w_{j,s}$, which is effectively the service rate for channel j. For $i < w_{j,s}$, $F_j(i)$ is 0. That is, until the load on channel j is equal to its service rate, it experiences no blocking. The amount of blocking channel j experiences for $i >= w_{j,s}$ is proportional to how much load it can handle.

We define our distance function with these properties in mind. In particular, for two given connections j and k, we want to compare their service rates ($w_{j,s}$ and $w_{k,s}$), the amount of blocking they observe at those service rates ($F_j(w_{j,s})$ and $F_k(w_{k,s})$), and finally, their expected blocking with a high fraction of the total load ($F_j(w_{j,R})$ and $F_k(w_{k,R})$):

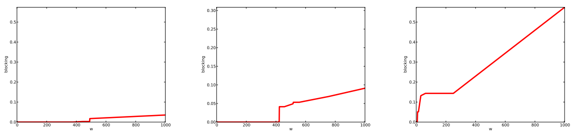


Figure 7: Sample predictive functions, F_j.

$$Distance(F_j, F_k) = \max\left(\left| \log \frac{w_{j,s}}{w_{k,s}} \right|, \alpha \left| \log \frac{F_j(w_{j,s})}{F_k(w_{k,s})} \right|, \alpha \left| \log \frac{F_j(w_{j,R})}{F_k(w_{k,R})} \right| \right)$$

We compare the logarithms of the ratios of these values to penalize large differences far more than small differences. We use the max of these values, rather than their sum or product, to avoid the information loss inherent in aggregating numbers. The scaling factor, α, ensures that all of the values are on the same scale. We define it as $\alpha = \frac{\log R}{|\log R\delta|}$ where R is the maximum discrete value that w_j can be, and δ is the value we introduce when we need to force monotonicity.

With *Distance*, we can define a distance between any two functions F_j and F_k. Using these distances, we perform agglomerative clustering [2] to discover clusters among the connections. After forming the clusters, we create a new function for the cluster which incorporates all data from the individual connections in the cluster. We then solve the optimization problem presented in the previous section with these new, clustered functions.

Clustering is effective because it reduces the dimensions of the problem. Rather than solving, say, a 64-way optimization problem, we may end up solving a 3-way optimization problem. The clustered functions will also tend to be more robust, because they incorporate more data than is available to just a single channel.

5.4 Encouraging Exploration

We have noted that distributed streaming systems are inherently dynamic. Although we have focussed on a single parallel region and its corresponding hosts, exogenous load will arrive, depart and change frequently. Streaming systems can also be bursty. On the other hand, we have also seen that new (and thus up-to-date) load balancing data arrives rather infrequently: At any given moment in time, data is potentially being collected only at the *current* allocation weights $\bar{w}_j$ for the various connections. In fact, because of drafting, it is only typically being collected at *one* of these connections in a given data collection interval. This paucity of new data is problematic in a dynamic environment, precisely because it encourages static optimization decisions. Handling dynamic behavior implies that we must encourage exploration rather than hinder it.

We deal with this issue in a simple but effective manner. Encouraging exploration means that some allocation weights must rise and some must fall. So our intuition is to "flatten" the blocking rate function for all weights beyond the current allocation weight of each connection. Specifically, if $\bar{w}_j$ represents the current allocation weight for connection j, we reduce the blocking rate $F_j(w_j)$ for each $w_j > \bar{w}_j$ by a fixed amount (chosen to be 10%) during each iteration of the algorithm. Reducing such values geometrically in this fashion, together with the enforced monotone regression scheme,

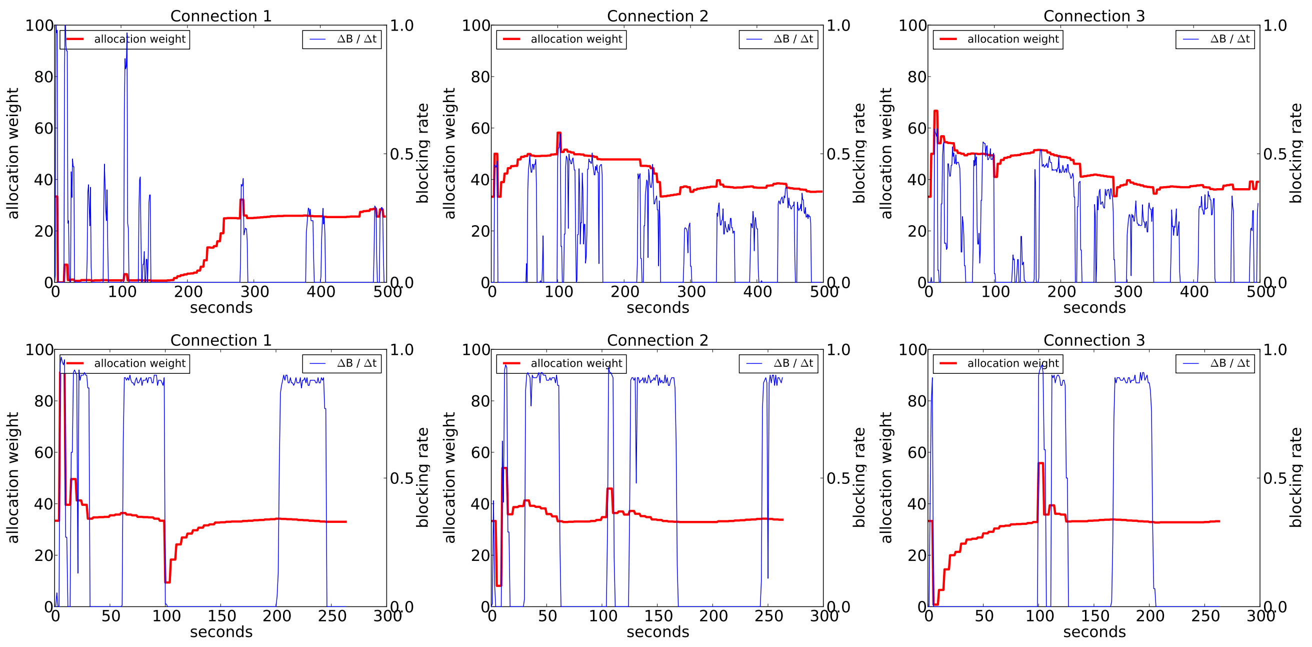


Figure 8: **Top**: experiment with 3 PEs, base tuple cost of 1,000 integer multiplies, 1 PE is 100× more expensive. **Bottom**: experiment with 3 PEs, base tuple cost of 10,000 integer multiplies, all PEs have same capacity.

causes the blocking rate to become essentially flat beyond $\bar{w}_j$ over time. Given this scenario, the optimization scheme will automatically start the exploration process quickly, causing fresh data collection.

6. EXPERIMENTAL RESULTS

To demonstrate the effectiveness of our load balancing scheme, we present two kinds of experiments. The first set show the in-depth behavior of a single run. The second set show total execution time and final throughput for many runs as we vary the number of PEs, with some PEs experiencing simulated load.

Our in-depth experiments show one graph per connection. Observing how the system adapts in the presence and absence of external load shows how our model is able to overcome the challenges presented in Section 4. The x-axis is the number of seconds into the experiment. The left y-axis is the allocation weight, which is the percentage of tuples that connection is receiving at that moment in time. The right y-axis is the blocking rate for that connection.

Observing a single run is not enough to demonstrate that our model works, or is better than the alternatives in a variety of conditions. To that end, we present experimental results which compare the total execution time and final throughput of runs where half the PEs are experiencing simulated external load. These graphs have four different alternatives: *Oracle** is the best distribution for the configuration, determined offline and by-hand; *LB-static* is our model without the decay mechanism which encourages exploration; *LB-adaptive* is our model with the decay mechanism to encourage exploration; and *RR* is naive round-robin with no dynamic load balancing. The purpose of *Oracle** is to provide a best case for the performance. However, we name it *Oracle** because in the dynamic case, it will change the allocation weights earlier than is optimal. The purpose of *RR* is to show what would happen without any load balancing. All execution times are normalized to *Oracle** for that run, and all final throughputs are in millions of tuples processed per second.

Unless otherwise mentioned, all of our experiments were run on machines with 2 Intel Xeon X5365 processors at 3.0 GHz. These processors have 4 cores, yielding 8 cores per machine. Each machine has 15 GB of RAM, and they are connected with InfiniBand. In our experiments, we distribute PEs across nodes so that we have one PE per core. The splitter and merger reside on different machines than the parallel workers. Hence, when we use 16 PEs, we are using two machines for the 16 parallel workers, and a third for the splitter and merger.

6.1 3 PEs with load imbalance

Our first in-depth experiment, top of Figure 8, has a parallel region with 3 worker PEs processing tuples with a base cost of 1,000 integer multiplies per tuple. In the beginning of the experiment, one PE has a simulated external load causing it to take 100× longer to process tuples. An eighth through the experiment, we remove the simulated external load.

The heavily loaded connection is *Connection 1* in the top of Figure 8. It starts out with its even share of the allocation weight, and as a result, it experiences a high blocking rate. To compensate, our model decides to change its allocation weight to 0. However, we can observe that as a result, the other connections experience a sharp increase in their blocking rate. The loaded connection then tries an allocation weight of 9, which still results in a high blocking rate. It finally tries an allocation weight of 3, and goes back and forth a few times between 2 and 3. It still experiences some blocking at these weights, but giving some allocation to the loaded connection still yields less total blocking in the whole system.

At around 100 seconds, the data decay finally forces re-

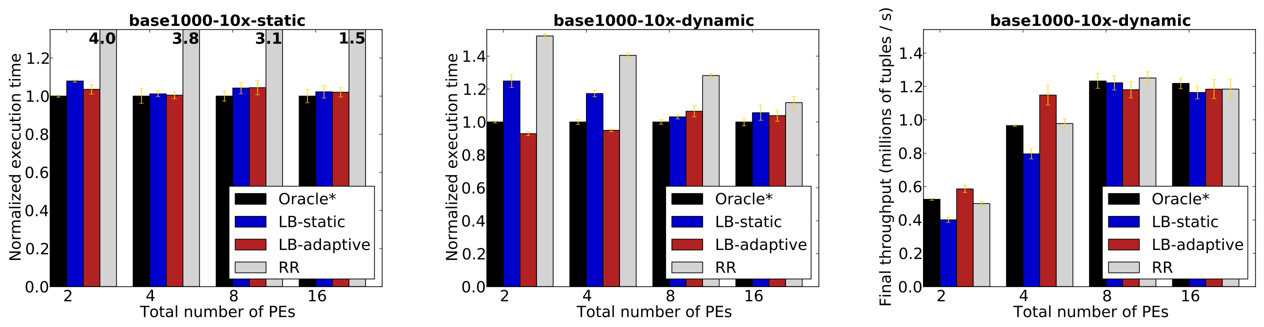


Figure 9: Base tuple cost of 1,000 integer multiplies and half of the PEs are 10× as expensive.

exploration: *Connection 1* tries an allocation weight more than 3, but it still sees a severely high blocking rate, and backs off. At the next re-exploration, around 175 seconds, *Connection 1* does not see increased blocking because at this point in the experiment, we have removed the 100× load. That connection then starts a slow climb back up to an even tuple distribution—slow because its function still indicates that blocking is probable at higher allocation weights, and the new data is slowly changing that function to indicate otherwise. The spikes around 275, 375 and 475 seconds are further re-explorations, but at this point, all of the connections have the same capacity, so no significant changes occur.

This experiment demonstrates three important behaviors. First, our model is able to quickly detect and adapt to severe load imbalance. Just 15 seconds into the experiment, we settle on a sustainable load distribution. Second, if re-exploration shows that the system has not changed, our scheme recovers. Finally, if re-exploration shows that the system has changed, our scheme adapts.

6.2 3 PEs with no load imbalance

Our second in-depth experiment, bottom of Figure 8, uses 3 parallel PEs which process tuples with a base cost of 10,000 integer multiplies, and no external load. The purpose of this experiment is to observe the behavior of our scheme when all connections have equal capacity, but a high blocking rate is unavoidable.

In the beginning, *Connection 3* in the bottom of Figure 8 is the drafting leader; it experiences most of the blocking even though all of the connections have equal capacity. As a consequence, its allocation weight drops to 0. *Connection 2* observes some blocking, and its allocation weight drops to 8. *Connection 1* experiences very little blocking, and picks up over 90% of the total allocation weight. For the remainder of the first 100 seconds, *Connection 1* and *2* remain relatively even while *Connection 3* recovers from dropping to an allocation weight of 0.

At 100 seconds, *Connection 2* and *3* are induced to explore higher allocations weights, taking weight from the drafting leader, *Connection 1*. However, the resulting distribution results in too much blocking, and *Connection 1* starts taking allocation from the other two connections until they stabilize at an even split. Note that after 150 seconds, which connection is the drafting leader changes several times. (For example, from 200–250 seconds, the drafting leader is again *Connection 1*.) But, at this point, all of the connections have explored enough of the allocation weight space to build essentially the same functions. Thus, even in the presence of drafting, our model is able to detect equal capacity.

6.3 Varying PEs with medium-cost tuples

The experiment on the left in Figure 9 uses a variable number of PEs where the base cost of a tuple is 1,000 integer multiplies. Half of the PEs in each experiment have a simulated load which causes them to take 10× as long to process tuples. The load remains unchanged (it is static) throughout the run.

With 2-16 PEs, our load balancing scheme is 1.5-4× better than basic round-robin. Since the load is kept at a constant during the experiment, the load balancing does not need to be adaptive. However, the marginal difference between *LB-static* and *LB-adaptive* demonstrates that with medium-cost tuples, there is only a marginal cost to being adaptive.

The experiments on the middle and right in Figure 9 are the same as the previously discussed experiment, with one difference: an eighth through the experiment, we remove the simulated load from half the PEs. The middle graph in Figure 9 shows the normalized execution time; the right graph shows the absolute final throughput. Dynamically removing the load an eighth through the experiment demonstrates the importance of adaptation. It also means that total execution time does not tell the whole story, since it includes the period of time when the load was present. Hence, we include the final throughput, which is well after the load has been removed. This throughput is indicative of the performance the configuration would achieve if it ran longer—which is important for streaming systems that are designed to run continuously.

Perhaps surprisingly, *LB-adaptive* outperforms *Oracle**. This result is caused by the fact that while we know the best distribution to use when there is load and when there is no load, in order to get an optimal run, we have to time the change-over exactly. The technique we use for *Oracle** changes distributions too quickly. For that reason, we do not call it *Oracle*.

At 2 and 4 PEs, the benefit of adaption is apparent in both the total execution time and the final throughput. At 8 PEs, however, we have reached the point where the workload stops scaling; for a base cost of 1,000 integer multiplies per tuple, 8 PEs is the point at which additional parallelism does not improve performance. This point is reached in the dynamic experiments because once the load is removed, all PEs can operate at full capacity. The *Oracle** schedule for 16 PEs with 10× load only uses 8 of the PEs.

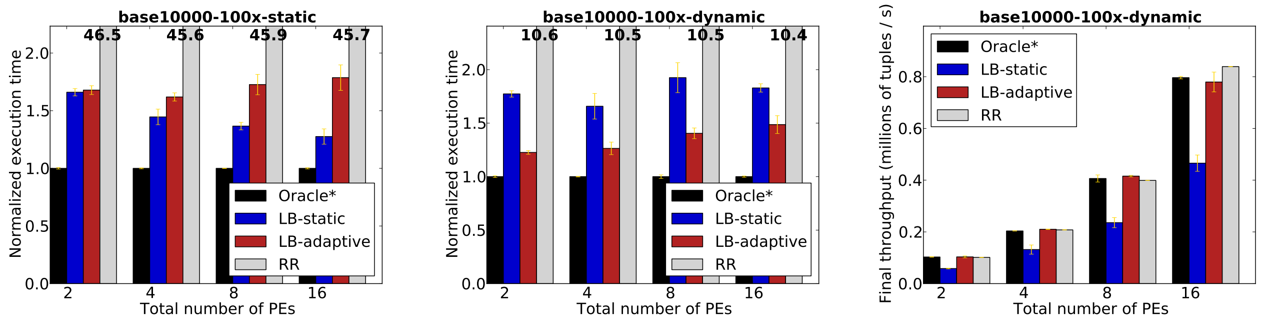


Figure 10: Base tuple cost of 10,000 integer multiplies and half of the PEs are 100× as expensive.

6.4 Varying PEs with heavy-cost tuples

The experiments in Figure 10 use a base tuple cost of 10,000 integer multiplies, using a 100× load on half of the PEs. The graph on the left of Figure 10 represents a static experiment, where the 100× load remains unchanged for the entire experiment. The load balanced approaches take about 1.3–1.8 longer than *Oracle**. However, basic round-robin, which naively sends an even amount of tuples to all worker PEs, takes 45× as long to complete as *Oracle**. Dynamic load balancing is clearly needed in this case.

As the number of PEs increases, the gap between *LB-static* and *LB-adaptive* grows from marginal to about 30%. This gap is the cost of being adaptive.

The benefit of being adaptive can be seen in the middle and right graphs of Figure 10. Both graphs represent an experiment where the 100× load is removed an eighth through. The middle graph is the total execution time normalized to *Oracle**, and the graph on the right is the final throughput in millions of tuples per second. During the first eighth of the experiment, both *LB-static* and *LB-adaptive* build severe blocking rate functions for the PEs with 100× the load. However, because *LB-static* is never induced to re-explore, it also never discovers that the load has been removed. *LB-adaptive* does discover that the load has been removed, and as a result, its final throughput is almost twice that of *LB-static*. In the dynamic experiments, the final throughput for *RR* is always roughly that of *Oracle** and *LB-adaptive*. However, note that *RR* took at least 10× as long to reach this throughput.

6.5 PEs on heterogeneous hosts

So far, all of our experiments have run on the same kind of host machines with simulated load. Our final set of experiments uses hosts with different capabilities, and no simulated load. Thus, these experiments will require dynamic load balancing solely because of the inherent capacities of the systems. Our "slow" hosts are those used in all prior experiments. Our "fast" hosts are machines with 2 Intel Xeon X5687 processors at 3.6 GHz and 62 GB of RAM. The processors on the fast hosts have 4 cores, and 2 SMT threads per core, which means that the fast hosts can support 16 threads.

The top of Figure 11 shows an in-depth experiment with two PEs, where *Connection 1* is to the fast host, and *Connection 2* goes to the slow host. The initial behavior is the same to the other in-depth experiments, in that there are some brief oscillations as the two connections explore the allocation weight space and the model builds its functions.

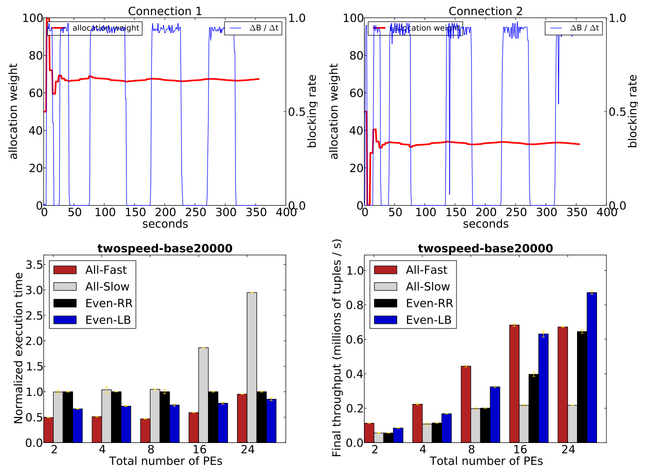


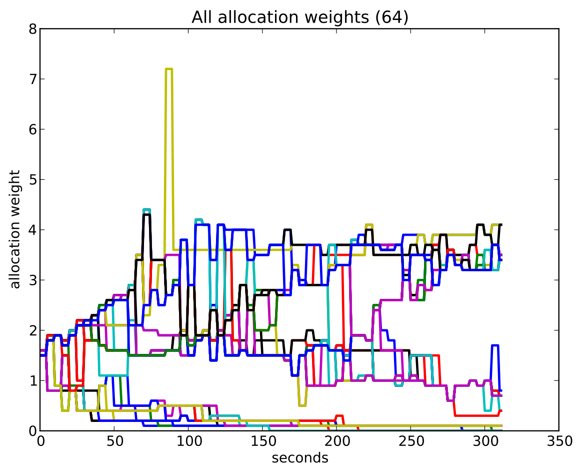
Figure 11: Experiments with "fast" and "slow" hosts using a tuple cost of 20,000 integer multiplies. **Top**: in-depth with 2 PEs; one on "fast", one on "slow." **Bottom**: varying the number of PEs.

The oscillations stabilize by 30 seconds into the experiment, where they settle on about a 65%-35% split, with small variations because of the exploration mechanism. And again, once the connections have explored enough of the allocation weight space, the model is robust to changes in who is the drafting leader.

In the bottom of Figure 11, we vary the number of PEs distributed across the heterogeneous hosts. There are four alternatives: *All-Fast* distributes all of the PEs to the fast node, using basic round-robin; *All-Slow* distributes all of the PEs to the slow node, using basic round-robin; *Even-RR* distributes half of the PEs to the slow node, half to the fast node, using basic round-robin; and *Even-LB* distributes half of the PEs to the slow node, half to the fast node, using our load balancing scheme. All execution times are normalized to *Even-RR* and all throughputs are in millions of tuples processed per second.

Up to 8 PEs, *All-Slow* and *Even-RR* perform the similarly, which is expected: overall performance will be gated by the slowest PE because of the merge. *All-Fast* outperforms *Even-LB* because even good load balancing cannot make up for the fact that half of its PEs are executing on slower hosts.

The slow host can only execute 8 PEs simultaneously; any more than 8 PEs, and the slow host becomes an oversubscribed system. Hence, performance degrades with *All-Slow* with 16 and 24 PEs. The fast host, however, can execute 16



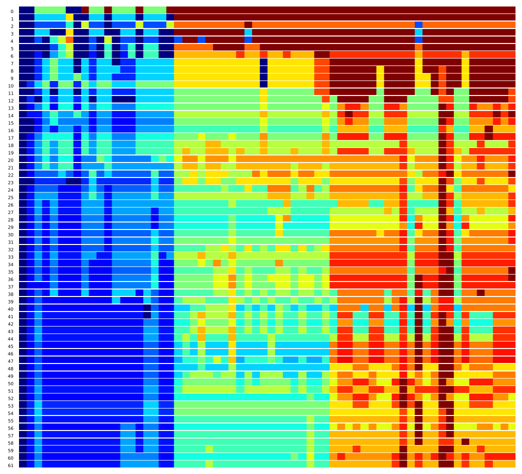


Figure 12: Experiment with 64 PEs and a base tuple cost of 60,000 integer multiplies; 20 PEs are 100× as expensive; 20 PEs are 5× as expensive; and the remaining 24 PEs just use the base tuple cost. On the left are the allocation weights per channel over time, and on the right is the "heatmap" for how channels were clustered. Matching colors are in the same cluster; one row represents one time step, so the x-axis is channel number and the y-axis is time, with $t = 0$ at the top.

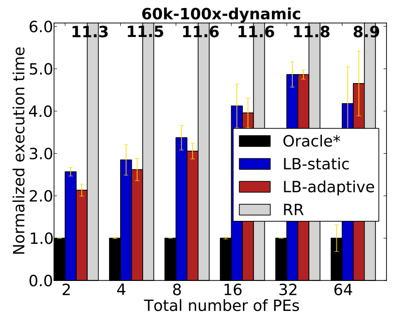
PEs simultaneously, since each core is a two-way SMT and our workload is integer multiplications. Because the fast host can handle 16 threads, its throughput increases when going from 8 to 16 PEs, but it does not improve with 24 PEs.

The fastest overall throughput is when 16 PEs are on the fast host, 8 PEs are on the slow host, and we use dynamic load balancing. Up until this point, *Even-LB* was at a disadvantage compared to *All-Fast* because half of its PEs were on the slow host. But the final configuration with 24 total PEs shows how adding a slow host to the system can *improve performance* if we use load balancing that can dynamically detect capacity.

6.6 Clustering

Our prior results did not include clustering, as explained in Section 5.3, as it only becomes necessary as the number of channels scales to 32 and higher PEs. The experiments in Figure 13 use a base tuple cost of 60,000 integer multiplies, and half of the PEs start with 100× the load, but that load was removed an eighth through the experiment. At 16 PEs and below, the experiments in Figure 13 behave similarly to the experiments in Figure 10. At 32 and 64 PEs, however, the total execution time for *LB-static* and *LB-adaptive* are similar, yet are both still close to 9× better than *RR*. The trend in final throughput, however, remains. Because *LB-adaptive* learns that the 100× initial load was removed, it is able to use more of the total number of PEs to achieve higher final throughput than *LB-static*.

However, total execution times and final throughputs only tell part of the story. In order to understand the dynamics of clustering and load balancing across 64 channels, we present the experiments in Figure 12. This experiment uses 64 PEs with a base tuple cost of 60,000 integer multiplies per tuple. This time, however, there are three classes of load: 20 PEs are at 100× the base cost, 20 PEs are at 5× the base cost, and the remaining 24 PEs just use the base cost. The graph on the left shows the allocation weight per channel over the course of the experiment. We can see that the PEs with 100× the load quickly learn they cannot handle much



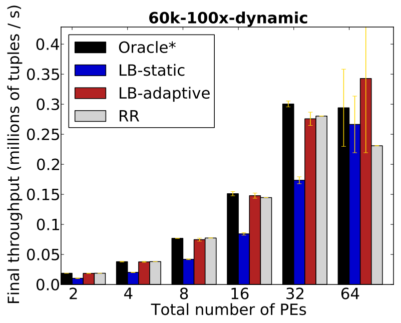


Figure 13: With clustering on, base tuple cost of 60,000 integer multiplies and half of the PEs are 100× expensive.

load. However, it takes longer for the unloaded PEs and the PEs with 5× the load to figure out which channel belongs where. Note that the last "switch" happens at about 220 seconds, where some channels from unloaded clusters realize they should be clustered with the 5× channels, and vice-versa. In the end, however, they all sort out where they should be.

The right graph in Figure 12 is the clustering "heatmap" for the experiment. Each row, starting at the top with $t = 0$, represents a clustering across all channels. Time into the experiment progresses downward from the top. The vertical slice of each row represents one of the 64 channels. Looking at a single row shows the clustering decision for a single timestep, and looking at a vertical column shows all clustering decisions over the lifetime of the experiment for a single channel.

In this experiment, we expect three classes of clusters to emerge, but this does not necessarily mean we will only see three clusters. All channels from, say, the 5× group do not need to all be in the same cluster. However, it is imperative that clusters emerge which have *only* channels from the 5× group, and the same for the other performance groups. If this is not the case, then channels will either have too much or too little work.

We can see this behavior in the heatmap in Figure 12; even though more three clusters emerge, there are only three classes of clusters in the end. Comparing the heatmap with the allocation weight graph, we can also see that the 100× clusters end up with a minimum allocation weight; the 5× clusters end up with an allocation weight no greater than 2; and the unloaded clusters end up with an allocation weight around 4.

7. RELATED WORK

The literature on load balancing in computer science is vast. It has been studied for topics as diverse as clustered web farms [21], cloud [19], grid [15], disk accesses in video-on-demand systems [22], and disks in general [6]. The constraints of a distributed, streaming system make our problem unique: there is an ordered merge and no global information.

Other distributed streaming systems do not perform dynamic load balancing for their data parallel regions in a manner similar to our scheme. The work of Nasir et al. [17] explores load balancing techniques in Storm, but it focused on the problem of data skew. Spark Streaming implements streaming through micro-batches of data, and uses delay-scheduling [24] to determine when and where to launch individual micro-batches based on data locality. Our scheme focuses on stateless streaming operations, where data locality is not a consideration, with strict ordering constraints.

Gordon et al. [11] present a compiler for StreamIt that produces a program with a good speedup on a multicore processor. By relying on the synchronous dataflow model, their techniques can take into account instruction-level information to target their application to the processor. In contrast, our streaming system is distributed and asynchronous: we have no knowledge of the processing requirements of the PEs; we do not know the processing capabilities of the hosts; and we do not know whether the hosts chosen for the PEs are shared.

The flexible filters work by Collins and Carloni [4] is a load-balancing scheme for working within a single multiprocessor. They create alternates for filters that can be bottlenecks. A splitter checks to see whether the destination filter's queue is full, and if so, sends the next data token to a designated alternate. By clever placement of alternates, they can achieve high throughput. Our approach is in the same spirit, but a direct application to the distributed scenario would be similar to the failed re-routing approach we describe in Section 4.4. In a distributed environment, by the time a queue is full, it is too late to make good load balancing decisions.

Xia et al. [23] devise a distributed algorithm for dealing with distributed resource management in a streaming system. Their scheme considers load balancing constraints to a degree, though their level of control is based on admission control and coarse-grained data routing. Their paper is theoretical in nature, not implemented on an actual streaming system.

8. CONCLUSIONS

We have presented a load balanacing scheme based on a novel metric, the blocking rate of the underlying data transport layer. We use this metric to construct functions for each connection, and minimize the maximum value of those functions to arrive at a balanced load distribution. Our model was implemented and tested in a real distributed streaming system.

Thus far we have concentrated only on the local version of our overall cluster load balancing problem. That is, we have considered a single parallel region of a single application, assuming that the parallel PEs have already been assigned to hosts. We have certainly seen that our scheme provides leverage for load balancing these specific hosts. But what about the cluster as a whole? How do we encourage this leverage across as many hosts as possible? Our future work will consider cluster-wide load balancing by assigning the parallel PE workers to many nodes. With many parallel regions, there will be flexibility in the whole system to adapt to changes.

9. REFERENCES

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. 2005.

[2] C. Aggarwal and C. Reddy. *Data Clustering: Algorithms and Applications*. CRC Press, 2014.

[3] 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):1792–1803, Aug. 2015.

[4] R. L. Collins and L. P. Carloni. Flexible filters: Load balancing through backpressure for stream programs. In *International Conference on Embedded Software*, Oct. 2009.

[5] Y. Diao, C. Wu, J. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh, C. Garcia-Arellano, M. Caroll, L. Chu, and J. Colaco. Comparative studies of load balancing with control and optimization techniques. In *American Control Conference*, 2005.

[6] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Comput. Surv.*, 14(2):287–313, June 1982.

[7] B. Everitt and D. Howell. Encyclopedia of statistics in behavioral science. J. Wiley, 2005.

[8] B. Fox. Discrete optimization via marginal analysis. *Management Science*, 13:210–216, 1966.

[9] G. Frederickson and D. Johnson. Generalized selection and ranking. In *Proceedings of the Symposium on Theory of Computing*, pages 420–428, 1980.

[10] Z. Galil and N. Megiddo. A fast selection algorithm

and the problem of optimum distribution of effort. *Journal of the ACM*, 26(1):58–64, 1979.

[11] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[12] M. Hirzel, S. Schneider, and B. Gedik. SPL: An extensible language for distributed stream processing. Research Report RC25486, IBM, July 2014.

[13] T. Ibaraki and N. Katoh. *Resource Allocation Problems*. MIT Press, 1988.

[14] IBM Streams. `http://ibmstreams.github.io/`. Retrieved September, 2015.

[15] Y. Li and Z. Lan. A survey of load balancing in grid computing. In *Proceedings of the First International Conference on Computational and Information Science*, CIS'04, pages 280–285, Berlin, Heidelberg, 2004. Springer-Verlag.

[16] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Symposium on Operating Systems Principles (SOSP)*, pages 439–455, 2013.

[17] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *International Conference on Data Engineering*, 2015.

[18] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 53–64, 2012.

[19] S. Shaw and A. Singh. A Survey on Scheduling and Load Balancing Techniques in Cloud Computing Environment. In *International Conference on Computer and Communication Technology*, 2014.

[20] 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, pages 147–156, New York, NY, USA, 2014. ACM.

[21] J. Wolf and P. Yu. Load balancing for clustered web farms. *ACM Transactions on Internet Technology*, 28(4):11–13, 2001.

[22] J. Wolf, P. Yu, and H. Shachnai. Disk load balancing for video-on-demand systems. *ACM Multimedia Systems Journal*, 5(6):358–370, 1997.

[23] C. Xia, D. Towsley, and C. Zhang. Distributed resource management and admission control of stream processing systems with max utility. In *International Conference on Distributed Computing Systems*, 2007.

[24] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[25] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.