Network-Aware Operator Placement for Stream-Processing Systems

Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, Margo Seltzer
Division of Engineering and Applied Science, Harvard University
hourglass@eecs.harvard.edu

Abstract

To use their pool of resources efficiently, distributed stream-processing systems push query operators to nodes within the network. Currently, these operators, ranging from simple filters to custom business logic, are placed manually at intermediate nodes along the transmission path to meet application-specific performance goals. Determining placement locations is challenging because network and node conditions change over time and because streams may interact with each other, opening venues for reuse and repositioning of operators.

This paper describes a stream-based overlay network (SBON), a layer between a stream-processing system and the physical network that manages operator placement for stream-processing systems. Our design is based on a cost space, an abstract representation of the network and on-going streams, which permits decentralized, large-scale multi-query optimization decisions. We present an evaluation of the SBON approach through simulation, experiments on PlanetLab, and an integration with Borealis, an existing stream-processing engine. Our results show that an SBON consistently improves network utilization, provides low stream latency, and enables dynamic optimization at low engineering cost.

1. Introduction

Distributed stream-processing systems (DSPSs), such as Borealis [7], Medusa [9], PIER [14], GATES [8], and IrisNet [13], collect, process, and aggregate data across massive numbers of real-time streams. These systems support applications such as monitoring of financial markets, detecting network intrusion, and connecting geographically-diverse sensor networks. DSPSs move query operators into the network, eliminating centralized processing. Pushing filtering, compression, and aggregation logic downstream can greatly reduce network traffic and increase performance. Even though identifying good placement nodes to host operators for a query is crucial, this *operator placement problem* is not addressed by current DSPSs.

DSPSs face three challenges when solving the operator placement problem. First, the placement of operators must result in *good query performance* for the application, such as low delay. Second, the DSPS should *use the network efficiently*, minimizing the global impact of all its running queries on the network. Doing so makes the DSPS more scalable in the number of supported queries and avoids squandering network resources that could be used by other applications. Especially in shared hosting environments with a large number of streams such as PlanetLab [24], socially-acceptable use of network links is important because resources are scarce and shared between participants. Third, when queries submitted by different users overlap, load and network traffic can be further reduced by finding and *reusing existing operators*. Although DSPSs have the need for decentralized, efficient operator placement and operator reuse, no common infrastructure exists for optimizing distributed data streams in a network-aware fashion.

In this paper, we describe a network abstraction called a *stream-based overlay network (SBON)*. Designed for large-scale placement of query operators in DSPSs, an SBON manages operator placement within a pool of wide-area overlay nodes in order to make efficient use of network resources. The SBON makes placement decisions based on its on-going knowledge of stream, network and node conditions and continuously optimizes placement without global knowledge of the system. By hiding the details of network measurement, operator placement decisions, and dynamic adaptation, the SBON layer simplifies the development of efficient, network-aware DSPSs.

Our SBON architecture uses a scalable, decentralized, and adaptive optimization technique based on a multidimensional metric space we call a *cost space*. Every node in the SBON maintains its cost space coordinate such that the distance between two nodes represents the overhead for routing data between them. The SBON determines the placement of a query operator in this virtual space using a spring relaxation algorithm, then maps its decision back to a physical node. This algorithm minimizes the network usage of a query, while keeping the query delay low and picking nodes with adequate bandwidth. Existing operators are also

Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)
8-7695-2570-9/06 $20.00 © 2006 IEEE

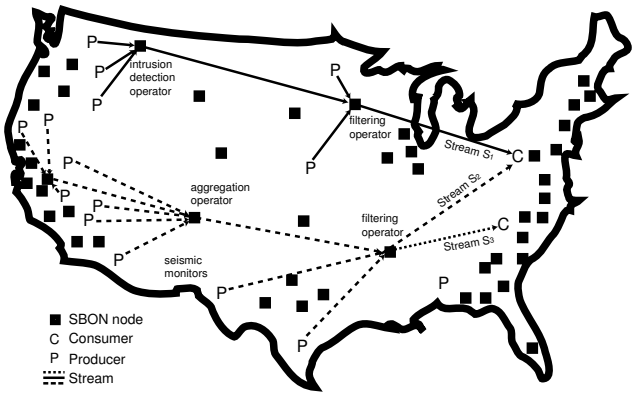


Figure 1. An example of a DSPS

located through the cost space and re-used for new queries when possible. SBON nodes dynamically re-evaluate operator placement decisions and migrate operators to new hosts based on changing conditions.

We evaluate our SBON implementation in simulation and with a deployment on PlanetLab. Our results show that, in simulation, our placement optimization technique exhibits close to optimal network usage. On PlanetLab, we benchmark the migration and reuse capabilities, in particular, and measure the engineering effort required to integrate it with the *Borealis* stream-processing system [7]. In our experiments, we found that operator migration reduced network usage by 17% and that re-use reduced it by 21%.

The rest of the paper is structured as follows. In the Section 2, we give the problem background and in Section 3 we describe our architecture. In Section 4, we evaluate the SBON implementation, and in Section 6 we conclude.

2. Distributed Stream-Processing Systems

A distributed stream-processing system streams data from multiple producers to multiple consumers via in-network processing operators, an example of which is shown in Figure 1. The figure shows three query streams that belong to two applications. S_1 performs intrusion detection for several networks and S_2 and S_3 aggregate seismic data from multiple sensor network deployments. Note that S_2 and S_3 share operators because the consumers are interested in the same data.

Query Model. We use the term *query* to denote the expression that is submitted by a single user describing her information need. Our basic model is one of multiple *queries*, each interconnecting multiple *operators*. The DSPS instantiates the query in the network in the form of a *query stream* of data tuples flowing through the network. Operators that are part of a query stream are interconnected by overlay *operator links*, each with a certain latency and a data rate.

Operator Model. Generic operators in a DSPS can be categorized into three classes: *producers*, *consumers*, and *operators*, which act as data generators, receivers, and processors, respectively. If an operator permanently resides at a physical network location, it is called a *pinned operator*. Producers and consumers are typically pinned. In contrast, an *unpinned operator*, such as a join or a select operator, can be instantiated at an arbitrary node in the network. Examples of processing operators include an aggregation operator for seismic data, a join operator for relational data, and a face recognition operator for video surveillance.

Operator Placement Problem. In Figure 1, there are many nodes that can host the four pictured unpinned operators. One of the main tasks of a DSPS is *operator placement*, or the selection of the physical node that should host the operator. The quality of a given placement is quantified by an *operator placement metric*.

2.1 Metrics for Operator Placement

In Section 1, we described three challenges when facing the operator placement problem: achieving good application query performance, using the network efficiently to service a query, and reusing existing operators when appropriate. These challenges should guide the choice of operator placement metric.

Achieving good application-perceived query performance, such as low delay, is a basic necessity. However, optimizing for individual application performance alone ignores the need to support a large number of streams. In DSPSs for financial markets, network monitoring, or wide-area sensor data collection, we expect many large concurrent streams to be a common use case. Individual streams must use network resources efficiently by conserving bandwidth and reusing existing operators where possible in order to support the largest number of concurrent streams. Furthermore, there is a monetary argument for minimizing bandwidth if the owners of a DSPS pay for the bandwidth usage of their system.

A tension exists between satisfying the application-perceived performance needs and minimizing overall bandwidth usage. At one extreme, an application-perceived delay metric will greedily blast data from producers to consumers, rather than favoring a lower bandwidth stream that uses in-network operator placement. At the other extreme, a metric that seeks to maximize the number of concurrent streams could make room by routing an individual stream on a circuitous path, increasing that stream's application-perceived delay.

In this paper, we introduce a metric for operator placement, *network usage*, that trades off application delay and consumed network bandwidth. This metric is similar to what others have proposed for the evaluation of application-level multicast where minimizing application delay and consumed network bandwidth have been seen to collide [10].

2.2 Current Techniques for Operator Placement

Even though all DSPSs are faced with the problem of in-network operator placement, current placement metrics do not satisfy the three operator placement goals listed above.

Most DSPSs, such as *Borealis* [7] and *GATES* [8], currently avoid the operator placement problem by supporting only pre-defined operator locations with pinned operators in the network. This leaves the burden of efficient operator placement to the system administrator, which is infeasible for a dynamic, large-scale system with thousands of queries.

Other DSPSs, such as *Medusa* [9], place operators to improve application performance by balancing load. This is appropriate within a single data center but leads to poor performance on a wide-area network, in which communication latencies can dominate processing costs. In addition, these approaches are not scalable to DSPSs with thousands of nodes that must be considered for placement.

The location of operators and corresponding relational tables in *PIER* [14], a distributed database built on top of a DHT, is determined through hashing, leading to effectively random placement of operators in the network. Such random distribution has good load-balancing properties but causes large query delays when operators are placed at nodes distant from both producers and consumers.

To our knowledge, the only prior work on network-aware operator placement in DSPSs is SAND [1, 2], which is proposed as an extension to Boeralis. Here, operators are placed either at the consumer side, at the producer side, or in-network on a DHT routing path between the two endpoints, depending on the bandwidth usage of a query. Applications can also specify delay constraints on the placement path in the DHT. In previous work [20], we have shown that DHT routing paths can lead to inefficient candidate sets for operator placement. This is because DHT routing tables are optimized for minimizing hop count and not for delay or bandwidth usage. Our approach of performing operator placement in a cost space is more general than SAND because placements are not tied to DHT routing paths.

2.3 Network Usage: A Blended Metric

We suggest a more appropriate metric for operator placement, *network usage*, that trades off overall application delay and network bandwidth consumption. The network usage $u(q)$ is the amount of data that is in-transit for query q at a given instant:

$$u(q) = \sum_{l \in L} \mathrm{DR}(l)\ \mathrm{Lat}(l). \qquad (1)$$

where L is the set of links used by the stream, $\mathrm{DR}(l)$ is the data rate over link l, and $\mathrm{Lat}(l)$ is the latency. This captures the bandwidth-delay product of the query. The network usage U for the entire DSPS is simply the sum over all queries in the system.

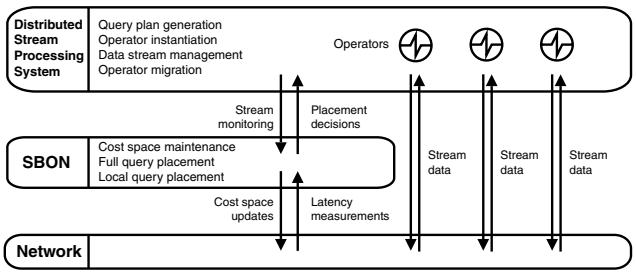


Figure 2. Architecture Overview

From an individual application's perspective, this metric may appear frightening as it seemingly says little about an individual stream's perceived delay. However, in this paper we show that negative effects on application-perceived delay are small. Intuitively, this is true because of the way in which the *network usage* metric effectively favors nodes that are "close" to the centroid of the producers and the consumers in a query.

From the network perspective, one can think of this equation as scaling the network bandwidth by the relevant link latencies. This interpretation captures the idea that the longer data stays in the network, the more likely it is to traverse nodes and links that could be used for other queries. At the same time, a path that requires extra latency to reach an in-network operator is justified if that the operator can commensurately reduce the overall stream bandwidth.

While high latency links are obviously poor choices for latency-sensitive applications, they are poor choices for networking reasons as well. High latency between two nodes indicates that the routing path traverses a long geographical distance or passes through multiple physical links with intermediate routers. In both cases, the monetary cost of operating such a path is likely higher compared to shorter ones. Also, if path congestion is causing a link's high latency, this link should be avoided for the global good.

3. Architecture

In Figure 2, we show how we extend the functionality of a DSPS with an SBON layer. Our approach augments existing DSPSs with efficient operator placement, while exposing their full features to users. In particular, the query optimizer of the DSPS remains unchanged and creates an efficient query plan before passing it to the SBON layer for placement. A network node runs two components: (1) the DSPS handles operator instantiation, migration, and state, and stream data transport and (2) the SBON layer monitors local performance, manages the cost space, and informs the DSPS when to migrate its services.

Stream instantiation and optimization take place over several stages. First, the DSPS defines a query plan through

its internal query optimizer, taking into account estimates of data rates and selectivity of operators, for example. Second, it passes the query plan to the SBON layer, which performs operator placement based on current network and node conditions, binding every operator to a node. During the lifetime of a query, nodes hosting operators periodically reconsider local placements, potentially migrating operators.

As illustrated in Figure 2, the SBON layer interacts with the DSPS through a simple interface: the SBON requests the DSPS (1) to *instantiate* and *destroy* operators on the local node, (2) to *connect* and *disconnect* the input and output links of an operator to other operators and (3) to *migrate* an operator to a remote node. The DSPS is responsible for tearing down connections, packaging state, and instantiating the operator on the new node during a migration operation.

In turn, the SBON layer monitors information about the operators hosted on the local node in order to make optimization decisions: (1) the SBON is aware of the number of *input and output links* that an operator supports, (2) an operator may advertise its *selectivity* (*i.e.,* the ratio of the input and output data rates) or the SBON may obtain this information through runtime measurements of data rates on operator links, and (3) an operator exports information about whether it can be *migrated* between nodes and *reused* between multiple queries.

3.1. Operator Placement Algorithm

The main challenge for an operator placement algorithm is the potentially large number of nodes that need to be considered for placement. No entity has complete information about current network and node conditions to make an optimal decision. Unfortunately, simple heuristics that consider only a subset of all nodes for placement [1] or perform a localized search [5] risk never finding a good placement.

We wanted to design our operator placement algorithm to satisfy three basic requirements. First, it must be *scalable* in the number of concurrent queries, as well as the number of network nodes where operators can be placed. This implies that the algorithm must be decentralized, not depend on global knowledge of network conditions, and have low communication overhead. Second, the placement algorithm should be *efficient* and yield "good" placements in terms of latency and network usage. Finally, placement decisions should be *adaptive* to changing conditions, such as latency, stream data rates, and load.

The SBON achieves efficient, decentralized in-network operator placement and optimization through two mechanisms: (1) a *cost space*, which is a metric space that captures the cost for routing data between nodes and (2) a *relaxation placement* algorithm, which places operators using a spring relaxation technique that manages single- and multi-query optimization.

3.1.1 Cost Space

A cost space encodes network and node measurements from all nodes and is constructed in a decentralized fashion. By projecting the placement problem into a virtual cost space and then mapping the solution back to physical nodes, SBON nodes can compute operator placement decisions in a continuous mathematical space with the use of sophisticated optimization techniques. In addition, any SBON node can approximate globally optimal decisions. For example, an SBON node in Europe making a placement decision for a query with producers located in Japan can determine that a good placement node may be on the US West coast without having to probe the node directly.

A cost space is a d-dimensional metric space where the Euclidean distance between two nodes is an estimate of the cost of routing data between those nodes. Each SBON node is responsible for maintaining its own coordinate. A wide range of performance metrics can be used to define a cost space. We examine latency and load, but other resource measures, such as availability, bandwidth, memory, or processing power could have been included instead. Our SBON implementation uses a combined *latency/load space* with three latency dimensions and one load dimension.

Latency. The latency dimensions of a cost space form a *latency space* [16], where the distance between two coordinates is a reasonable prediction of the latency between the nodes. Several instances of synthetic network coordinates to estimate Internet latencies exist [16, 17]. The overhead of maintaining a latency space is small because a node can calculate its network coordinate after probing the latency to only a small subset of nodes. That subset consists of either well-known landmark nodes [17] or is randomly chosen [11]. Since Internet latencies often violate the triangle inequality and change dynamically over time, network coordinates can only provide an estimate of true latency. However, simulation results suggest that network coordinates, even with low dimensionality, have a small prediction error with a median of 11% [12].

Figure 3 shows a 3-dimensional latency space as calculated by the *Vivaldi* algorithm [11] with 115 North American nodes on PlanetLab. As annotated, three geographic clusters of nodes can be identified. Each node running Vivaldi keeps track of its 3-dimensional coordinate and its confidence in that coordinate. The algorithm works by each node successively refining its coordinates through periodic measurements to random other nodes. Each update consists of two nodes measuring the current latency to one another, plus an exchange of their coordinates and confidences. With this information, nodes develop a metric space where two nodes can approximate their true latency even if they have never exchanged measurements directly.

The latency space dynamically adapts to changing network conditions as nodes continuously refine their coordi-

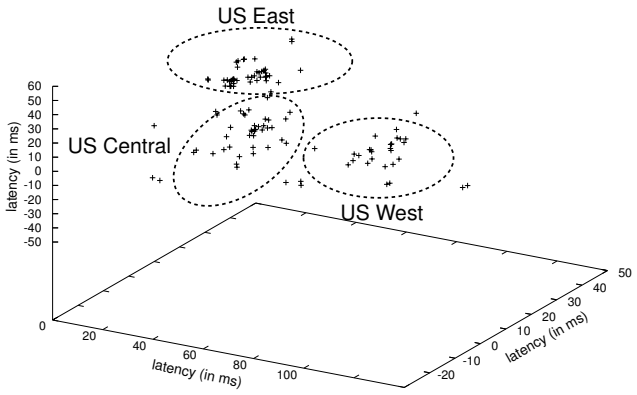


Figure 3. 3-D latency space on PlanetLab

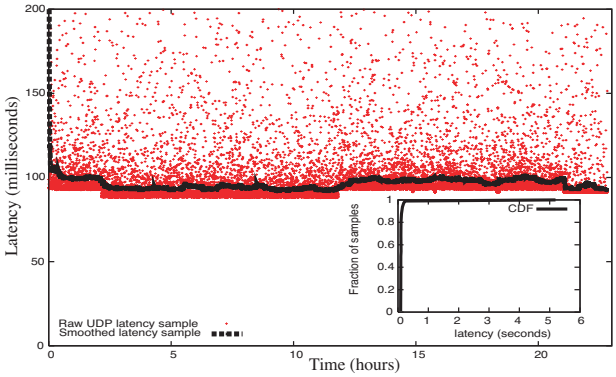


Figure 4. Latency samples on PlanetLab

nates. This raises the issue of stability of the latency space: if latency measurement variance is high, the latency space might not converge. Especially on nodes with high CPU load, application-level latency measurements may lead to skewed results [23]. On the other hand, a certain amount of dynamism is desired for the latency space to adapt to changing network conditions.

As part of developing the latency space, we found a simple mechanism to create stable, adaptive coordinates. We feed each latency observation into a moving percentile filter, an instance of a low-pass non-linear filter. Each link maintains its own filter and, even though latency observations have significant variance and their distribution is heavily skewed, the filter captures the consistent baseline of each link. Figure 4 shows the raw and smoothed application-level UDP latency samples gathered between two SBON nodes on PlanetLab over the course of 23 hours together with the CDF of all samples. The observed baseline latency varies due to Internet route changes. The variance in measurements is caused by the load on the nodes, which averaged 40 during our sampling, and congestion on the network paths. We found the filter dampens measurement variance on the latency space without losing sensitivity to changes in Internet routing paths. We determined empirically that the 20th percentile from a history of 10 samples

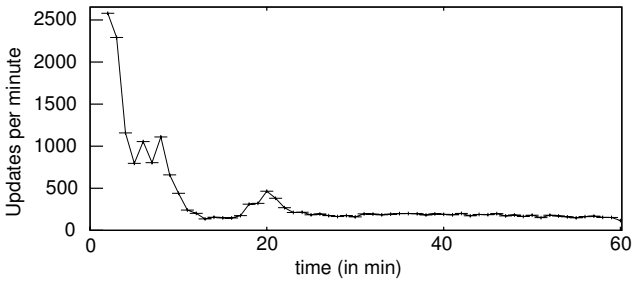


Figure 5. Rate of coordinate updates

leads to a stable latency space on PlanetLab [18].

We also found empirically that the latency space converges quickly. As shown in Figure 5, we observed that it takes around 30 minutes for a latency space with 116 simultaneously-added North American PlanetLab nodes to reach stability. The median relative prediction error for latencies was 9% for these nodes. In addition, the latency space can gracefully handle node churn because new nodes can learn their coordinates after a small number of measurements.

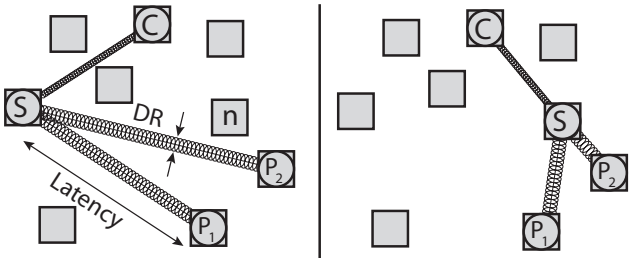
Load. The performance of a placed operator may be dominated by the CPU load of the hosting node. Therefore, an SBON node monitors its own load average and includes this as another dimension in its coordinate. A scaling function determines the weight given to load with respect to latency. As a result, nodes with a high load will move away in the cost space making them less likely candidates for placement decision. If particularly load-sensitive operators were to be run on the system, a multiplicative or non-linear additive function could be used to further penalize nodes with high load. For simplicity, we adopt a linear mapping in our experiments.

3.1.2 Relaxation Placement

Our placement algorithm, called RELAXATION, makes efficient operator placement decisions within the cost space. The main idea behind RELAXATION is to partition the placement problem into two phases. First, an unpinned operator in a query is *placed* using a spring relaxation technique in the latency dimensions of the virtual cost space, and then the solution is *mapped* to the closest physical SBON node in the cost space.

Using spring relaxation to place services in the cost space has several advantages. First, its iterative nature allows placement decisions to adapt to changing network, node, and query conditions. Second, spring relaxation is decentralized and does not require coordination between nodes. Finally, it naturally supports cross-query optimization, which considers the impact of shared placement decisions.

RELAXATION models the overlay network of queries as a collection of massless bodies (operators) connected by



(a) Before Relaxation (b) After Relaxation

Figure 6. Example of RELAXATION

springs (operator links). Pinned operators have a fixed location, whereas unpinned operators can move freely. The goal is to compute the rest lengths of this system of interconnected springs. The average force $\vec{F}_i$ experienced by a spring i is $\vec{F}_i = \frac{1}{2} k_i \vec{s}_i$ where k_i is the spring constant and $\vec{s}_i$ is the extension vector. To minimize network usage in the latency space with spring relaxation, we set the spring extension to the latency, $s_l = \mathrm{Lat}(l)$, and the spring constant to the data rate transfered over that link, $k_l = \mathrm{DR}(l)$.

Optimizing placement then becomes solving for the low energy state of the system, minimizing the sum of the potential energies E_i stored in the springs:

$$\arg\min_{\vec{s}_i} \sum_i E_i = \sum_i \vec{F}_i \vec{s}_i = \sum_i \frac{1}{2} k_i \vec{s}_i^{\,2} \qquad (2)$$

Thus, spring relaxation minimizes

$$\sum_{l \in L} \mathrm{DR}(l)\ \mathrm{Lat}(l)^2, \qquad (3)$$

which includes the network usage metric $u(q)$. The additional squared exponent in the function ensures that there is a unique solution from a set of placements with equal network usage.

We illustrate the placement of a query in latency space in Figure 6. The thickness of each link is proportional to its data rate and length to link latency. In (a), the location of the unpinned operator S has become sub-optimal due to network dynamics and "stretches" the operator links to producers P_1 and P_2. This causes the operator to migrate to a better node n, which is shown in (b). In general, a strong force pulling an unpinned operator in a particular direction can either be caused by multiple operator links or a single operator link that has a high data rate.

Spring relaxation can be implemented efficiently with a decentralized algorithm. Each spring is relaxed independently by moving it a small amount, potentially affecting the extensions of other springs in the system. After a number of such relaxation iterations, the system of springs converges towards a low energy state because each iteration decreases the total force in the system. Another advantage of this approach is that it naturally supports placement decisions when queries are interconnected with shared operators. By viewing the entire query graph as a network of springs, the placement of an operator can then potentially affect the placements of other operators with transitively shared operator links. In practice, placement effects will be more localized because we factor in the operator migration cost to ensure that placement decisions do not create oscillation in the system (see Section 3.2).

```
VIRTUAL-PLACE(S)
1  repeat
2        F⃗ ← 0⃗
3        for each S_parent in Parents(S)
4        do   F⃗ ← F⃗ + (S⃗ − S⃗_parent) × DR(S, S_parent)
5        for each S_child in Children(S)
6        do   F⃗ ← F⃗ + (S⃗ − S⃗_child) × DR(S_child, S)
7        S⃗ ← S⃗ + F⃗ × δ
8    until ‖F⃗‖ < F_t
```

Figure 7. RELAXATION **algorithm**

3.1.3 Algorithm

The SBON layer employs a two-step placement algorithm that has a straightforward decentralized implementation. Consider an SBON with its queries. For each unpinned operator, the following two steps are executed: the operator is first *placed* using RELAXATION in the latency dimensions of the cost space with respect to the location of its neighbors in the query; next, the computed cost space coordinate is *mapped* to a physical SBON node. To make the placement decision adaptable, the placement and mapping steps are repeated continuously for all unpinned operators. This does not cause a large communication overhead because each SBON node can perform placement decisions with local knowledge after learning only the cost space coordinates of its direct query neighbors. Because all nodes are a part of the cost space, any node can perform the virtual operator placement and physical operator mapping stages. We now describe the two steps in more detail.

Virtual Operator Placement. Figure 7 shows the pseudocode used by RELAXATION to find the virtual placement for an unpinned operator in the cost space. The VIRTUAL-PLACE function is executed by the SBON optimizers on every SBON node. The current cost space coordinate of the operator is $\vec{S}$. Only the latency dimensions of the cost space coordinate are considered because, ideally, the operator should be placed at a node with lowest possible load. A new unpinned operator without a coordinate is assigned a provisional location close to the coordinate origin at random. The total force $\vec{F}$ that this operator experiences is calculated by iterating over its parent and child operators connected through input and output operator links and retrieving their current cost space coordinates over the network. The force $\vec{F}$ is updated with the distance in cost space between the current coordinate $\vec{S}$ and the remote coordinates scaled by the data rate used by the query link (lines 3–6).

The data rates of a new query are based on estimates provided by DSPS, which are refined through network measurements once stream data is flowing. The movement of the operator through latency space is dampened by a factor δ to avoid unnecessary oscillation around the optimal location. The cost space coordinate of the operator is updated iteratively (line 7) until the magnitude of the force $\vec{F}$ is larger than a force threshold F_t (line 8). The force threshold sets the desired precision of the virtual operator placement. After the operator coordinate has been computed, the operator is mapped to a physical SBON node. We determined empirically that $\delta = 0.1$ and $F_t = 1$ work well on PlanetLab and used these values in our experiments.

Physical Operator Mapping. The second stage of the placement algorithm maps the target cost space coordinate to a physical node. It uses the virtual placement coordinate (with the load dimension equal to zero) to identify a region of the cost space and then proceeds to actually select a node (with potentially non-zero load) within that region based on application requirements.

The stage falls into five steps:

1. Find k nodes whose coordinates are near the target cost space coordinate.
2. Contact this small set of nodes directly to discover their current operators and resources.
3. Sort the list by distance to the target coordinate.
4. Walk the sorted list, returning the first node already running the operator.
5. Failing that, return the nearest node that meets the application's resource criteria

Step (1) is done by a k-nearest neighbor search in the cost space. We used $k = 10$ in our experiments. This problem has been well-studied in the literature and our prototype currently solves it centrally by performing a simple brute-force search. Several distributed solutions have been deployed with success and are directly applicable. The two primary approaches have been based on space-filling curves [3, 22] and on greedy geographic routing in a multi-dimensional metric space [21]. We have developed initial implementations based on each of these approaches [19] and comparing them is part of on-going work. If no nodes are found, we expand the search parameter k.

Step (5) ensures that operators are only placed on nodes that have sufficient resources to support the operator. Resources include node resources, such as CPU, memory, and disk space, and also network resources, such as available network bandwidth. Due to a lack of resources operator sometimes are placed sub-optimally in terms of our network usage metric. However, this distributes processing load of operators across nodes, avoiding localized hotspots.

3.2. Placement Optimization

Operators are initially placed by a *full placement optimizer* and the placements are periodically re-evaluated by *local placement optimizers*.

Full Placement Optimizer. To create a new stream, a DSPS passes a query plan to the SBON layer on any node. The *full placement optimizer* finds an initial placement for all unpinned operators in the new query using RELAXATION. The full placement optimizer runs only once and has complete knowledge of the entire query plan. After receiving physical placements for all unpinned operators from the full placement optimizer, the SBON layer instructs the DSPS to instantiate the new operators and create the stream.

When a new query is added that has the same structure and operators as an existing query, network and node resources can be saved by reusing operators between queries. This is related to multi-query optimization in traditional database systems [15]. The cost space can guide the search for reusable operators and reduce the complexity of having to consider all operators in the system: when the full placement optimizer performs the physical operator mapping for a desired placement coordinate, it also retrieves the currently hosted operators at nodes in that *region* of the cost space. If it finds a reusable operator that produces the same data as the operator to be placed, it reuses this operator and the corresponding sub-query instead of instantiating a new instance.

Local Placement Optimizer. The *local placement optimizer* re-evaluate the placement decisions of locally running operators and initiates migrations of operators when necessary. It runs periodically on every SBON node and iterates over all local unpinned operators, re-placing them in the latency space. It then maps the new virtual placement coordinate to a physical node only when the displacement from the previous coordinate is larger than a cost space threshold to avoid unnecessary nearest neighbor lookups. This threshold value depends on the perceived cost of a lookup. After mapping the coordinate to a physical node in the latency/load space, the local optimizer calculates the saving in network usage for this new placement. To prevent needless migrations, an operator is migrated only if the saving in network usage is higher than a *minimum migration threshold*. This threshold depends on the cost of operator migrations and ensures that migrations are amortized over the lifetime of a query.

One might think that two local optimizers running on different nodes could potentially lead to oscillations of operator placements. Since all local optimizers have access to the same cost space, however, their placement decisions agree with each other because they are based on the same information. In rare cases, rounding errors determine the outcome of a placement that is exactly in the middle be-

tween physical nodes in the cost space. Then the minimum migration threshold will ensure that an operator remains at its current location.

Since local optimizers are running concurrently, it is important that they do not interfere with each other. Therefore, migrations are done atomically in the SBON. When a local optimizer decides to migrate an operator, it first notifies the DSPS to stop the data flow through the query. While a query is stopped, local optimizers running on other SBON nodes do not attempt optimizations for the same query. After an optimizer has finished running (and carried out any migrations), the flow of data is restarted. An additional benefit is that no data is lost during the migration because operators are explicitly notified to stop or buffer data production.

4. Evaluation

We evaluated our implementation of the SBON layer in simulation and with experiments on PlanetLab. Our results fall into three groups: we show in simulation that RELAXATION minimizes network usage while providing low delay to applications (4.1); we investigate how the SBON layer migrates and reuses operators on PlanetLab (4.2); and we link the SBON layer to a current DSPS that does not perform network-aware operator placement (4.3). Our experiments show how DSPS can benefit from our placement optimization techniques with little implementation effort.

4.1. Placement Efficiency in Simulation

We first wanted to compare the performance of RELAXATION to other placement algorithms. We implemented the algorithms in a discrete-event simulator. Using simulation has the advantage of giving us complete control over the network topology and the experiment, so we could ensure that the placement algorithms were compared under identical conditions.

We examined five alternative placement approaches: OPTIMAL chooses the best possible placement with the lowest network usage based on an exhaustive search over all possible placements. This requires global knowledge of the entire network and is not feasible in practice, but it gives a baseline for the performance of the algorithms. IP MULTICAST places unpinned operators at nodes that would be routers hosting branches of an IP multicast tree rooted at each producer. For each query, we calculate the IP multicast tree by taking the union of all the IP unicast routes from the producers to the consumers. PRODUCER randomly picks one of the nodes hosting producers for placement of the unpinned operator. CONSUMER places the operator at the node hosting the consumer. Finally, RANDOM picks a physical node for each operator at random and serves as a worst case comparison.

We simulated 1550 nodes in a transit-stub topology generated by the Georgia Tech topology generator [25]. The topology has 10 transit domains with 5 nodes, each connected to 150 stub domains with 10 nodes on average. Routing tables for the topology were calculated using the routing policy weights assigned by the topology generator to reflect Internet routing policy. The network diameter of the topology was 878 ms. Producers and consumers were randomly distributed across the network. Only a single producer operator was hosted at any physical node. A query consisted of four producers, an intermediary join and select operator, and a consumer. We refer to the join and select as a single *aggregator* operator. For simplicyt, all producers sent data at a rate of 2 KB/s and the aggregator had a selectivity of 8:1. We placed the aggregators on routers in the transit-stub topology; in a real deployment, they would be part of low-latency networks connected to the routers. The placement of operators in transit domains reflects that some nodes in a large-scale DSPS might be hosted at Internet exchanges.

Table 1. Increase in network usage and delay

Algorithm	*Network Usage Penalty*	*Delay Penalty*
OPTIMAL	0%	13%
RELAXATION	15%	24%
IP MULTICAST	27%	0%
PRODUCER	43%	75%
CONSUMER	60%	0%
RANDOM	81%	76%

Network Usage. We show the average increase in network usage after placing 1000 queries in Table 1, where OPTIMAL minimizes network usage. RELAXATION performs well, which is expected because network usage is the metric that it optimizes for in its cost space. IP MULTICAST is less efficient because it minimizes routing hops and not necessarily routing latency. Therefore, RELAXATION manages to find better placement nodes that are not on the direct IP routing path from the producer to consumer. PRODUCER does better than CONSUMER because, by placing the selective operators on one of the producers, data from only three producers needs to be sent through the network to the operator. RANDOM exhibits poor network usage and CONSUMER is only marginally better.

The distribution of network usage over all queries is shown in Figure 8. It illustrates the clear division between the evaluated placement algorithms. Looking at the 80th percentile, RELAXATION manages to cause only 14% more network traffic than OPTIMAL, whereas PRODUCER results in 48% more traffic.

Delay Penalty. We examined the delay penalty that an application experiences due to each placement strategy. We define delay penalty as the increase in delay when compared to the IP routing delay on the longest path from any producer to the consumer. Placing all unpinned operators on the consumer node (as done by CONSUMER) achieves the

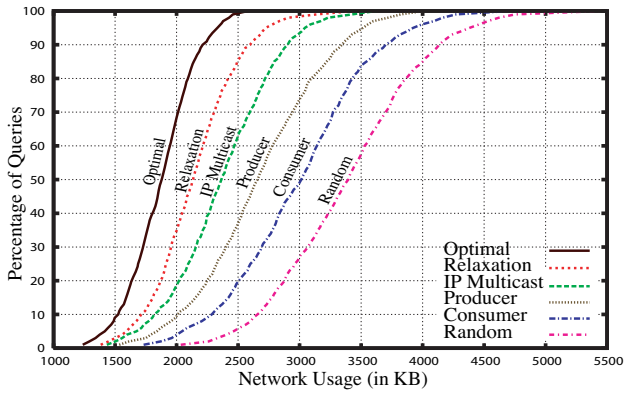


Figure 8. Network usage

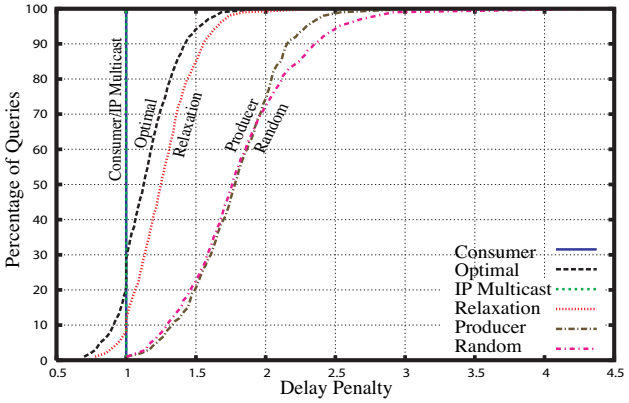


Figure 9. Delay penalty

lowest delay penalty of 0%.

Table 1 shows the average delay penalty after placing 1000 queries. As expected, CONSUMER and IP MULTICAST have no delay penalty because they place operators only along IP routing paths. RELAXATION and OPTIMAL introduce only a small delay penalty of 24% and 13%, respectively. PRODUCER and RANDOM perform worst because they add a random delay overhead.

Figure 9 shows the distribution of delay penalty. A interesting result is that both OPTIMAL and RELAXATION achieve lower delay penalties than CONSUMER for about 15% of the queries. This is due to the fact that the routing weights in the transit-stub topology reflect the fact that IP routes sometimes have sub-optimal latencies. OPTIMAL and RELAXATION are then able to reduce the delay by choosing different overlay routes with a higher hop count but lower delay. Note that PRODUCER and RANDOM have a long tail of bad placement decisions with a large delay penalty. The results from the simulations portray that, while RELAXATION is optimizing for network usage, its impact on delay penalty, an application-oriented metric, remains low.

4.2. Placement Optimization on PlanetLab

Our second set of experiments examine operator migration and reuse on PlanetLab. We first wanted to monitor the behavior of a single pair of queries over a long duration. We wanted to see if and to what extent migration, driven by a changing cost space, would change a single continuous query in detail.

We started two simple queries and monitored them for 20 hours. Both queries had the same pair of producers `P1` and `P2`, a single operator that perfomed a select and a join, and the same consumer. The producers produced monotonically numbered tuples at a constant rate. The SQL for the query was `SELECT * FROM P1,P2 WHERE P1.data=P2.data AND mod(P1.data,4)=0`, effectively selecting $\frac{1}{4}$ of the tuples for forwarding.

We used 130 PlanetLab nodes from diverse geographic locations as our experimental testbed. We used a latency/load cost space, including load as a fourth dimension with equal-weight. We measured application-perceived tuple delay by having the recipient send a small reply packet for each tuple sent. This practical measure approximates "true" delay, given that the error in local clocks on PlanetLab can exceed tuple delay. The producers sent a data packet once a second for each query.

The pair of single queries had two producers on the US West Coast (*berkeley* and *ucsc*) and had the consumer in Europe (*upc.es*). Starting at the same time, each query ran for 20 hours. Figure 10 shows the network usage for each query. Each point depicts the instantaneous network usage of the query. Vertical bars denote operator migrations. The results show that, after 4 hours, the migrating query reduces its network usage by 30% from migration C onward by placing the operator next to one of the producers. Further migrations keep the operator near the producers, which occur due to changes in the trans-Atlantic link and the load on the West Coast nodes.

A more detailed look at the effect of each placement decision (A, B, ...) is illustrated in Figure 11. It illustrates the effect of the migrations A through F on the query. The numbers on top of each bar graph give the network usage normalized by the network usage of the first placement, A. This placement is the same as for the non-migratable query and, therefore provides a comparison point for how migration has improved network usage while also reducing delay by a small margin. The bar graphs denote each link's contribution to total network usage over time and the length of the lines denotes latency. In column A, the data initially travels from two producers (*ucsc* and *berkeley*), through an aggregator (*uoregon*), where it is joined and selected, and sent across the Atlantic to *upc.es*. Over 20 hours, the operator migrates to *washington* (B), then *berkeley* (C), then *ucsc* (D), then back to *berkeley* (E), and then *ucsc* (F). Operator migration minimizes the length of the portions of the query before the selection, reducing network usage.

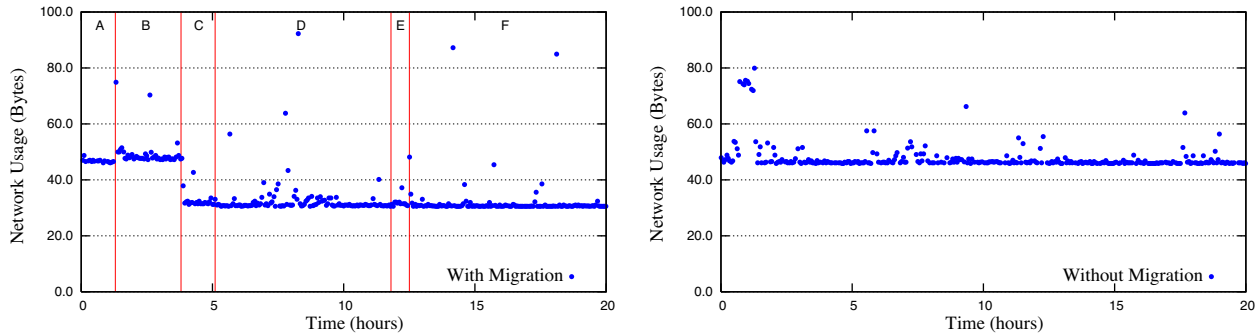


Figure 10. Network usage of a pair of queries, one with a migration (left) and one without (right)

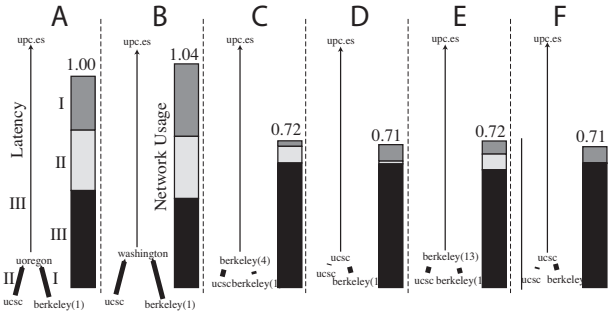


Figure 11. Effect of operator migration

4.2.1 Operator Migration

The third experiment examines the aggregate effect of migrations due to changes in network and node conditions over an extended period of time. Given that PlanetLab was heavily loaded when we ran our experiments, we also wanted to quantify the effect incorporating node load into the cost space would have on network usage. Like the previous experiment, each query streamed data from two producers through an aggregator/selector, and out to a consumer. Data rates, tuple sizes, and constituent nodes were the same as in the previous experiment. Because of potentially high variation between one set of ⟨producer,producer,consumer⟩ triples and another, we created our migratable and non-migratable queries in pairs, each with the same set of endpoints. We created 24 pairs of queries, which ran for 5 hours, and recorded the network usage of each.

We show the change in the relative network usage of each migratable query in Figure 12, compared to the query with the same producers and consumer without migration enabled. Values below 0 indicate a *decrease* in network usage when migration is enabled. Permitting migration often leads to lower mean network usage, although in a small number of cases the usage increases. We find that migrations were effective in decreasing network usage for 75% of the query pairs. Data from the same experiment also show that the set of queries that are permitted to migrate use 16.9% less of the total network capacity compared to non-migratable queries. The migratable queries performed an average of 3.5 migrations each, suggesting that migrations are occurring at a moderate rate. Allowing queries to migrate also reduced aggregate query delay by 10.5%.

We also ran the same experiment with a pure latency space. The results show that including load resulted in a modest improvement in per-query latency and aggregate network usage with similar numbers of migrations. We have omitted the results due to space constraints.

4.2.2 Operator Reuse

Another advantage of the SBON model is that the processing of operators can be reused across different queries. In this way, the SBON layer avoids transmitting multiple redundant copies of the same data across the network. This is especially important when transporting data over long-haul trans-oceanic links, which may experience high congestion and long latencies. The SBON automatically detects opportunities for data sharing across queries and combines operators with the same inputs whenever those operators are placed in a similar region of the cost space by RELAXATION.

To demonstrate this, we created four pairs of queries. Each query contained a consumer on the US West Coast requesting aggregated data from the same two producers in Europe. In the first case, we disabled operator reuse, requiring each query to instantiate its own aggregation operator. In the second case, operator reuse was enabled, and aggregation was shared across three of the four queries (as determined by RELAXATION). Because the shared aggregation operator was generating only a single copy of its output data, the query also included a *multicast* operator which redistributed the shared data to the individual consumers. Figure 13 shows the logical topology of the two sets of queries.

As expected, the overall network usage with operator reuse enabled (119.55 bytes) was lower than when reuse was disabled (152.32 bytes), a savings of 21%. We expect the savings to be greater with a larger number of queries sharing the same data. In many stream applications, we anticipate data and operator popularity to follow Zipf-like distributions, as has been demonstrated for other types of Internet traffic [6], further highlighting the importance of operator reuse in DSPSs.

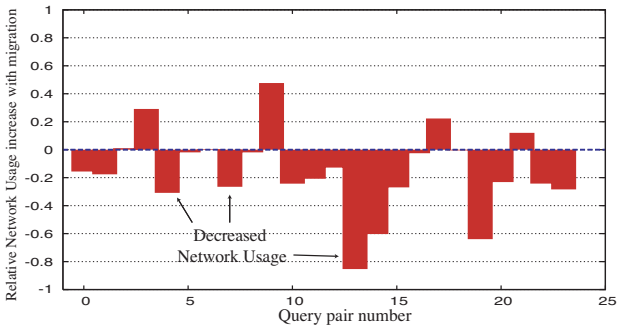


Figure 12. Aggregate effect of migration

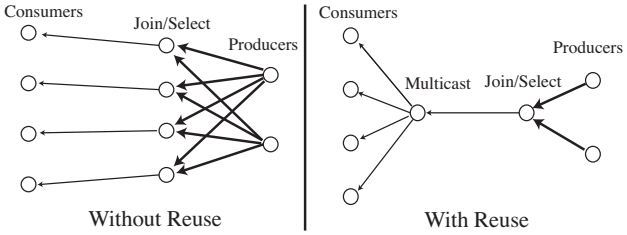


Figure 13. Reuse Topology

4.3. Borealis Extension

We believe that current DSPSs can benefit from an SBON layer to achieve network-aware operator placement. To test this claim, we extended *Borealis* [7], an existing distributed stream processing engine, with our SBON implementation. This section describes how, with few code changes, the resulting system retains the full query functionality of Borealis while achieving the network usage and delay benefits shown in the previous experiments.

In Borealis, part of a running query, or *chunk* of *boxes*, can be migrated to different nodes. This migration capability is used by an agoric contract-based optimizer [4] that decides to buy or sell units of work based on internal queue usage. Borealis can also handle manual operator reuse, in which a single query can serve multiple consumers. This means that Borealis can take full advantage of the adaptive placement decisions made by the SBON.

In an attempt to obtain a useful measure of integration effort, the Borealis extension was performed with no prior code-level knowledge of the Borealis code base. Once the Borealis codebase was understood, the actual code changes were minimal and were completed and tested over the span of several days. The integration involved fewer than 400 lines of code, mostly for communication between our SBON layer (written in Java) and Borealis (written in C++).

Our extension is structured as follows. When a user creates a new query, Borealis creates a query plan that is passed to the SBON layer. Since Borealis requires a fixed initial location for operators, the initial placement result of the SBON full query optimizer is used as the starting point for the query passed to Borealis. This starting location is automatically sent to Borealis, which in turn starts the Borealis data flow. From this point onwards, the SBON optimizer sends migrate commands to the Borealis operator execution environment via their operator interface. This interface directly invokes the migration code implemented in Borealis.

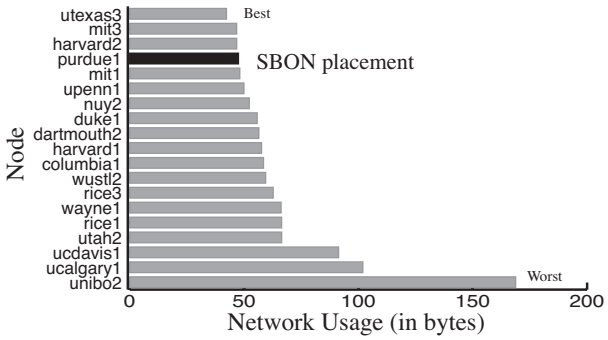


Figure 14. Operator placement with SBON/Borealis.

On PlanetLab, we used Borealis to deploy a set of identical queries that had two pinned producers, one consumer, and an unpinned windowing aggregate operator. The pinned producers were at *utexas3* and *harvard1*, and the pinned consumer was at *ucdavis1*.[1] The producers published data at the same rate, while the operator had a measured selectivity of 12%. One query was placed by the SBON optimizer on one of 19 PlanetLab nodes. To compare against the efficiency of manual placement, we simultaneously added queries that pinned their operator on each of the PlanetLab nodes in turn, which is akin to conducting an exhaustive search for the best placement in terms of network usage.

As can be seen in Figure 14, the query placed by the SBON optimizer has one of the lowest network usages of any of the placement possibilities. While there are a number of nodes that are relatively good choices, RELAXATION favors nodes farther from the query endpoints (because of spring pull) when there are good mid-cost space candidates. This figure is representative of how the SBON performs a good initial placement decision without needing to perform the exhaustive search, and how it can help Borealis with the initial placement.

5. Future Work

As future work, we plan to explore extensions of our cost space approach to include other metrics, such as available bandwidth and node reliability. We also intend to analyze the convergence properties of relaxation placement in more detail with the placement of more complex queries. Especially with bursty workloads, additional dampening of placement decisions may be necessary.

[1] The Planetlab overlay involves machines placed at different universities and research institutions. The notation *utexas3* signifies the third machine that is placed at University of Texas at Austin.

Another area for future investigation is the applicability of classical database optimization techniques in the setting of a cost space. This introduces an interesting tension between the desire to make the SBON's placement optimizer agnostic to operator semantics and the need to give the SBON enough information to make good optimization decisions. For example, some stream operators, such as joins, can be decomposed, which would allow the SBON to place them closer to producers. Appropriate interfaces between the DSPS's query optimizer and the SBON's placement optimizer could be designed for a tighter integration.

6. Conclusions

We introduced an SBON layer as a tool for distributed stream-processing systems to assist with the operator placement problem. A DSPS generates a query plan with pinned and unpinned operators, and the SBON manages the task of positioning unpinned operators efficiently using its ongoing knowledge of network and node conditions. We solve the operator placement problem with a two step approach: (1) the placement decision is made in a virtual cost space, and (2) the cost space coordinate is mapped to a physical node. The cost space encodes network and node measurements from all nodes in a scalable and decentralized manner, enabling individual nodes to make adaptive placement decisions with local information.

Through PlanetLab-based experimentation and simulation we demonstrated that the SBON creates data streams that minimize overall network usage, a characteristic that is increasingly important as the number of data streams in the network increases. We showed that SBONs do so with a combination of intelligent placement, based on the characteristics of the stream, and of operator reuse. We believe that the relatively easy integration of our SBON layer with an existing stream processing system demonstrates its benefits. We hope this work will induce future research efforts into large-scale query optimization for DSPSs that leverage our approach.

References

[1] Y. Ahmad and U. Çetintemel. Network-Aware Query Processing for Stream-based Applications. In *Proc. of VLDB'04*, Toronto, Canada, Aug. 2004.

[2] Y. Ahmad, U. Cetintemel, J. Jannotti, A. Zgolinski, and S. Zdonik. Network Awareness in Internet-Scale Stream Processing. *IEEE Data Eng. Bulletin*, 28(1), Mar. 2005.

[3] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Proc. of P2P*, Linköping, Sweden, September 2002.

[4] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *Proc. of NSDI'04*, Mar. 2004.

[5] B. J. Bonfils and P. Bonnet. Adaptive and Decentralized Operator Placement for In-Network Query Processing. In *Proc. of IPSN*, Palo Alto, CA, Apr. 2003.

[6] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. of INFOCOM*, New York, NY, March 1999.

[7] U. Centintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of CIDR*, Asilomar, CA, Jan. 2005.

[8] L. Chen, K. Reddy, and G. Agrawal. GATES: A Grid-Based Middleware for Processing Distributed Data Streams. In *Proc. of HPDC*, Honolulu, HI, June 2004.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, et al. Scalable Distributed Stream Processing. In *Proc. of CIDR*, Asilomar, CA, Jan. 2003.

[10] Y.-H. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. In *Proc. of SIGMETRICS*, June 2000.

[11] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Practical, Distributed Network Coordinates. In *Proc. of HotNets-II*, Cambridge, MA, Nov. 2003.

[12] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *Proc. of SIGCOMM*, Portland, OR, Aug. 2004.

[13] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An Architecture for a World-Wide Sensor Web. *IEEE Pervasive Computing*, 2(4), Oct. 2003.

[14] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. of VLDB*, Berlin, Germany, Sept. 2003.

[15] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4), Dec. 2000.

[16] T. E. Ng and H. Zhan. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proc. of INFOCOM*, New York, NY, June 2002.

[17] M. Pias, J. Crowcroft, S. Wilbur, S. Bhatti, and T. Harris. Lighthouses for Scalable Distributed Location. In *Proc. of IPTPS*, Berkeley, CA, Feb. 2003.

[18] P. Pietzuch, J. Ledlie, and M. Seltzer. Supporting Network Coordinates on PlanetLab. In *Proc. of WORLDS'05*, San Francisco, CA, Nov. 2005.

[19] P. Pietzuch, J. Ledlie, J. Shneidman, M. Welsh, M. Roussoupolos, and M. Seltzer. Service Placement in Stream-Based Overlay Networks. Poster, NSDI'05, May 2005.

[20] P. Pietzuch, J. Shneidman, J. Ledlie, M. Welsh, M. Seltzer, and M. Roussopoulos. Evaluating DHT-Based Service Placement for Stream-Based Overlays. Proc. of IPTPS'05.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *Proc. of SIGCOMM*, San Diego, CA, Aug. 2001.

[22] C. Schmidt and M. Parashar. Flexible Information Discovery in Decentralized Distributed Systems. In *Proc. of HPDC*, Seattle, WA, June 2003.

[23] K. Shanahan and M. J. Freedman. Locality Prediction for Oblivious Clients. In *Proc. of IPTPS*, Ithaca, NY, Feb. 2005.

[24] The Planetlab Consortium. http://www.planetlab.org, 2004.

[25] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. of INFOCOM*, San Francisco, CA, Mar. 1996.

This material is based upon work supported by the National Science Foundation under Grant Number 0330244.