COLA: Optimizing Stream Processing Applications via Graph Partitioning

Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf,
Kun-Lung Wu, Henrique Andrade, and Buğra Gedik

IBM T.J. Watson Research Center, Hawthorne, NY 10532, USA
{rohitk,hildrum,sujay,drajan,jlwolf,klwu,hcma,bgedik}@us.ibm.com

Abstract. In this paper, we describe an optimization scheme for fusing compile-time operators into reasonably-sized run-time software units called processing elements (PEs). Such PEs are the basic deployable units in System S, a highly scalable distributed stream processing middleware system. Finding a high quality fusion significantly benefits the performance of streaming jobs. In order to maximize throughput, our solution approach attempts to minimize the processing cost associated with inter-PE stream traffic while simultaneously balancing load across the processing hosts. Our algorithm computes a hierarchical partitioning of the operator graph based on a *minimum-ratio cut* subroutine. We also incorporate several fusion constraints in order to support real-world System S jobs. We experimentally compare our algorithm with several other reasonable alternative schemes, highlighting the effectiveness of our approach.

Keywords: stream processing, operator fusion, graph partitioning, optimization, scheduling.

1 Introduction

We live in an increasingly data-intensive age. By some estimates [1], roughly 15 petabytes of new data are generated every day. It is becoming an ever more mission critical goal for corporations and other organizations to process, analyze and make real-time operational decisions based on immense quantities of data being dynamically generated at high rates. Distributed systems built to handle such requirements, called stream processing systems, are now becoming extremely important. Such systems have been extensively studied in academic settings [2,3,4,5,6,7,8], and are also being implemented in industrial environments [9,10]. The authors of this paper are involved in one such stream processing project, known as System S [11,12,13,14,15,16,17,18,19,20], which is highly scalable distributed computer system middleware designed to support complex analytical processing. It has been evolving for the past six years.

1.1 Operator Graphs and the Fusion Problem

Application development in System S is facilitated by the spade development environment. Among other things, spade defines a programming model based on

J.M. Bacon and B.F. Cooper (Eds.): Middleware 2009, LNCS 5896, pp. 308–327, 2009.

type-generic streaming *operators*, as well as a stream-centric language to compose these operators into parameterizable, distributed stream processing applications. In this model, depicted in Figure 1(a), an application is organized as a *data flow graph* consisting of operators at the nodes connected by directed edges called *streams* which carry data from the source to destination operators. Examples of streaming operators include functors (such as projections, windowed aggregators, filters), stream punctuation markers, and windowed joins. SPADE also allows flexible integration of complex user-defined constructs into the application.

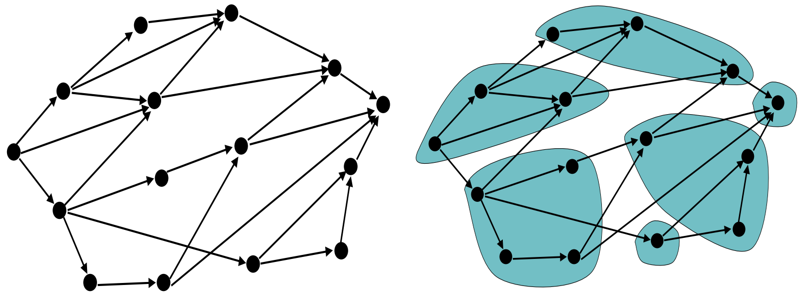

(a) Operator-level Data Flow Graph (b) PE-level Data Flow Graph

Fig. 1. Operators, PEs and Data Flow Graphs

The operator-level graph of Figure 1(a) represents a *logical* view of the application. When the application is executed on a System S cluster, these operators must be distributed onto the compute nodes for execution. From the point of view of the Operating System (OS) on a node, the unit of execution is a process. One of the main tasks of the SPADE compiler is to convert the logical operator-level data flow graph into a set of executables that can be run on the cluster nodes. A System S application executable is called a *processing element* (PE), which serves as a container for one or more operators, and maps to a process at the OS level. Coalescing several operators into a single PE is called *fusion* (described in detail in [13]). A compiled SPADE-generated application becomes a *physical* data flow graph consisting of PEs with data streams flowing between them. This is depicted in Figure 1(b), with the shaded regions representing PEs. One can think of the PEs as essentially *supernodes* in the original operator data flow graph.

When data is sent on a stream between two operators in the same PE, the SPADE compiler converts it into a function call invocation of the "downstream" operator by the "upstream" operator. Sending data on streams between PEs is performed by inter-process communication, such as TCP sockets. Thus, only the inter-PE (physical) streams remain as edges in the PE-level graph. As a result of this transformation, passing data on an intra-PE stream has almost no processing cost compared to an inter-PE stream, which requires inter-process communication.

The goal of this paper is to tackle the *fusion problem*: how to map a logical operator-level graph into an optimal physical PE-level graph. A good fusion algorithm is critical to enable high-performance distributed stream processing applications that can be flexibly deployed on heterogenous hardware. But there are tradeoffs involved. To see this, consider the two extreme solutions to the operator fusion problem. On one end of the spectrum, suppose all operators were fused into a *single* PE. This solution eliminates all communication cost because all downstream operators are invoked via function calls. However, the resulting PE is a single process which is limited to one node, and does not exploit the available hardware parallelism of multiple nodes. On the other end of the spectrum, suppose *no* operators were fused, each operator corresponding to its own PE. If the PEs were well distributed on the nodes, they would exploit the available hardware resources. However, they would incur significant communication cost. The ideal solution would be someplace in between, so that fusion is used to reduce communication cost while providing flexibility to exploit the available compute capacity across multiple nodes. Our experiments show that relative to the unfused case, a good fusion algorithm allows the application to achieve over 3 times higher throughput on the same set of resources.

1.2 Our Contributions

In this paper, we describe COLA, a profile-driven fusion optimizer, which operates as part of the SPADE compilation process. COLA supports an environment with heterogeneous hosts, while allowing the user to specify a variety of important real-world constraints about the fusion. Its input is information about an application being compiled by SPADE, along with some attributes of a set of representative SYSTEM S hosts. We say COLA is profile-driven since it relies on application information in the form of performance metrics indicating the CPU demands of the operators and data rates of each stream.

Although we use the phrase *fusion*, COLA works from the top down rather than from the bottom up. Starting with all operators fused together into a single PE, the COLA algorithm iteratively splits "large" PEs into two separate "smaller" PEs by solving a specially formulated *graph partitioning* scheme. Then a *PE*

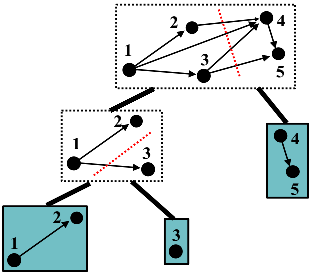


Fig. 2. Iterative graph partitioning in COLA

scheduler, serving as a compile time surrogate for the run time scheduler, hypothetically assigns the resulting PEs to potential hosts in an attempt to balance the load. If the combined solution to the graph partitiong and PE scheduling problems is not satisfactory, an *oracle* chooses one of the current PEs to split next, and the process iterates. Finally, the best solution found is chosen as COLA output, and the fused PEs are compiled. Several sample COLA iterations are shown in the format of a binary tree in Figure 2. At the root of the tree all operators are in one PE. The PEs created by solving the first graph partitioning problem are shown at depth 2. The oracle then picks one of these PEs to partition further (in this example, the first), and these PEs are shown at depth 3. And so the process continues. At the end of the last iteration the leaf nodes of the binary tree (shown shaded) represent the output COLA PEs.

In this paper, we describe two variants of COLA. The first version, called Basic COLA, generates a set of partitions that minimizes the communication cost, while ensuring that the fused PEs will still "fit" within the CPU capacities of the available nodes. While this basic version is not as sophisticated as the second version, it does illuminate several key ideas and components of the overall design. It also suffices for many applications. The second version is called Advanced COLA. In addition to the goals of Basic COLA, it (a) attempts to balance the load across the nodes, and (b) enables the user to restrict the fusion via a set of real-world constraints on the operators and PEs. We support six different types of constraints, three of which are *resource matching* (an operator requires a host with specific attributes such as CPU type), *PE exlocation* (operators cannot be fused together), and *host colocation* (operators must go on the same host). We retain and enhance the graph partitioning scheme used for Basic COLA, and introduce an integer programming formulation and solver to handle the PE scheduling problem in a more precise manner. The basic operator fusion problem has been tackled [13] using a greedy heuristic. We will show in this paper that COLA can provide a significant improvement over this heuristic. No other work to our knowledge has attempted to address the full version of the fusion problem with additional constraints.

Note that although COLA does consider the actual target system during its operation, it is not the same as the SYSTEM S runtime scheduler called SODA [18]. When an application (job) is actually executed on the system, it is SODA that manages the job. In particular, SODA provides functionality such as admission control (based on available resources), as well as load-balancing the cluster among multiple jobs and dynamically adapting to changing load conditions. For its evaluation purposes, COLA, which is invoked only at compile time, includes a simpler PE placement mechanism that mimics the full SODA algorithm.

Our contributions in this paper include the following.

- A new scheme for fusing SPADE operators into nearly optimal PEs in SYSTEM S, appropriate for general, heterogeneous processing host environments, and working synergistically with the scheduler.
- Support within the scheme for a wide variety of additional real-world constraints.

- A new and practical generalization of a classic graph partitioning problem, and a novel solution scheme.
- An effective compile-time PE scheduler which mimics the SODA SYSTEM S run-time scheduler.
- Experimental evidence suggesting that the COLA scheme has a major impact on the performance of SYSTEM S.

The remainder of this paper is organized as follows. Section 2 describes the Basic COLA problem formulation and the solution approach. In Section 3, we describe the formulation and solution strategy for Advanced COLA. Experiments showing the performance of the Basic COLA variant are described in Section 4. (The infrastructure to support the constraints is not yet available in SYSTEM S, so we defer experiments involving the Advanced COLA scheme for now.) Finally, in Section 5 we give conclusions and list future work.

2 Basic COLA

2.1 Problem Formulation

Consider a directed graph $G = (V, E)$ in which the vertices V represent the SPADE operators and the directed edges E represent the streams flowing between the operators. Assume that we are given *operator costs* $w_v \geq 0$ for $v \in V$ that represent the CPU costs of the corresponding operators, and *communication costs* $w_e \geq 0$ for $e \in E$ that represent the CPU costs due to sending and receiving tuples associated with the corresponding streams. (We will measure all CPU costs in terms of *millions of instructions per second*, or *mips*.) This input data is computed in SPADE via the use of efficient profiling methodology [13]. For a subset $S \subseteq V$, let $\delta(S)$ denote the set of edges with exactly one end-point in S. Let the *size* of a subset $S \subseteq V$ be defined as

$$\text{SIZE}(S) = \sum_{v \in S} w_v + \sum_{e \in \delta(S)} w_e. \tag{1}$$

Intuitively speaking, SIZE(S) denotes the total CPU utilization that a PE consisting of the subset of operators S would incur. Recall that the streams contained completely inside a PE are converted into function calls during compilation and incur negligible CPU cost. For two sets S and T, we denote the set difference (set of elements of S that are not elements of T) by $S \ominus T$. To simplify the notation, we denote $w(S) = \sum_{v \in S} w_v$ and $w(\delta(S)) = \sum_{e \in \delta(S)} w_e$. Thus, SIZE$(S) = w(S) + w(\delta(S))$.

Assume that we are also given a list of *hosts* $\mathcal{H} = \{h_1, \ldots, h_k\}$ with their CPU speed *capacities* $B_1, \ldots, B_k$, also in *mips*. The COLA fusion optimization problem can be stated as follows: find a fusion of the operators into PEs and an assignment of the PEs to hosts such that the total CPU cost of a host is at most its capacity and the total communication cost across the PEs is minimized. More formally, the problem is to partition V into PEs $\mathcal{S} = \{S_1, \ldots, S_t\}$ and compute an assignment function $\pi : \mathcal{S} \rightarrow \mathcal{H}$ such that

(i). for any $h_i \in \mathcal{H}$, we have $\sum_{S \in \mathcal{S}:\pi(S)=h_i} \textsc{size}(S) \leq B_i$, and
(ii). $\sum_{S \in \mathcal{S}} w(\delta(S))$ is minimized.

Expression (i) describes scheduling feasibility: The assigned operators must fit on the hosts. Recall that in this Basic COLA variant we do not require that the load be balanced *well*, just acceptably. The expression (ii) measures the total communication cost across the PEs. Technically, it is twice the total communication cost across the PEs, since each edge going between different PEs is counted coming and going. This multiplicative factor does not, of course, affect the graph partitioning optimization in any way. This problem can be shown to be NP-hard by a reduction from the balanced cut problem [21].

As an application developer, one is primarily interested in maximizing the amount of data that is processed by the job. This can be measured as the aggregate data rate at the source (input) operators of the job, and is commonly referred to as *ingest rate* or *throughput*. Since this metric is hard to model as a function of the operator fusion, COLA attempts to minimize the total inter-PE communication as a surrogate.

2.2 Solution Approach

```
1. Run Pre-processor
2. Use PE scheduler to compute an LPT schedule
3. Repeat until the schedule is feasible:
   (a) Use Oracle to identify PE p to split next
   (b) Use Graph Partitioner to split p into two PEs
   (c) Use PE scheduler to compute an LPT schedule
4. Run Post-processor
```

The pseudocode of the Basic COLA algorithm is given above. We will go into the key components in more detail below, but we first describe a high-level view of the scheme. To begin with, a *pre-processor* is used to "glue" certain adjacent operators together provided doing so would not affect the optimality of the final PE solution. Once these operators are identified we will simply revise the problem definition and treat the glued operators from then on as *super operators*. Then the main body of the scheme begins. At any point, the COLA algorithm maintains a current partitioning $\mathcal{S} = \{S_1, \ldots, S_t\}$ of the given graph into PEs. Initially, it places all the operators into a single PE, so that $t = 1$ and $S_1 = V$. The *PE scheduler* then finds an assignment π of PEs to hosts. Next, COLA checks to see if expression (i) is satisfied. If not, the oracle picks the next PE to split, the *graph partitioner* performs the split, attempting to minimize expression (ii), and the process iterates. At some point the PE assignments should become feasible. (For ease of exposition we will not discuss the handling of pathological cases, for example, one in which a single operator is too large to schedule.) Finally, a post-processor is employed in an effort to improve the solution slightly before the PEs are output.

Pre-processor. The pre-processor performs certain immediate fusions of adjacent operators into super operators, motivated by the following lemma. Essentially, the lemma proves that if, for any vertex v, the communication cost of one its edges (say, $e = (u, v)$) is larger than the sum of the operator cost of the vertex and the communication costs of all its other incident edges, then the edge e can be collapsed by fusing vertices u and v. Thus, the pre-processor fuses adjacent operators by collapsing edges with sufficiently large communication costs.

Lemma 1. *Consider a directed edge $e = (u, v) \in E$ from operator u to v and suppose $w_e \geq \min\{w_u + w(\delta(\{u\}) \ominus \{e\}), w_v + w(\delta(\{v\}) \ominus \{e\})\}$ holds. There exists an optimum solution in which u and v belong to the same PE. Here, for two sets X and Y, $X \ominus Y$ denotes the set with the elements from X that are not in Y.*

Proof. Consider any feasible solution in which u and v belong to distinct PEs S_1 and S_2 respectively. It is enough to show how to modify this solution so that u and v belong to the same PE without increasing its cost or violating its feasibility. Assume without loss of generality that $w_e \geq w_u + w(\delta(\{u\}) \ominus \{e\})$. In this case, we move u from S_1 to S_2. That is, we let $S'_1 \leftarrow S_1 \ominus \{u\}$ and $S'_2 \leftarrow S_2 \cup \{u\}$. It is easy to see that $\text{SIZE}(S'_1) \leq \text{SIZE}(S_1) - w_u + w(\delta(\{u\}) \ominus \{e\}) - w_e \leq \text{SIZE}(S_1)$ and $\text{SIZE}(S'_2) \leq \text{SIZE}(S_2) + w_u + w(\delta(\{u\}) \ominus \{e\}) - w_e \leq \text{SIZE}(S_2)$. Furthermore, the new objective value is at most the old objective value plus $w(\delta(\{u\}) \ominus \{e\}) - w_e \leq 0$. Thus the proof is complete.

The pre-processor iteratively fuses pairs of operators $\{u, v\}$ for which the condition in the above lemma holds. Once we fuse $\{u, v\}$ into a super operator U, we update its weight as $w_U = w_u + w_v$ and the weight of the edges incident to U as $w_{Ux} = \sum_{x \in V \ominus \{u,v\}} (w_{ux} + w_{vx})$ and $w_{xU} = \sum_{x \in V \ominus \{u,v\}} (w_{xu} + w_{xv})$. The super operators are simply treated operators in the following iterations. Our experiments show that this pre-processing step, while employed rather rarely, helps improve the quality of the final solution.

The resulting graph with all (super and other) operators placed in a single PE is then employed in the first iteration of the main body of the scheme.

PE Scheduler. Given a current set of PEs $\mathcal{S} = \{S_1, \ldots, S_t\}$, the role of the scheduler in the Basic COLA scheme is to determine if these PEs can be feasibly scheduled on the given hosts $\mathcal{H}$. That is, it tries to find an assignment function $\pi : \mathcal{S} \rightarrow \mathcal{H}$ such that expression (i) is satisfied. To find a relatively good assignment quickly we borrow and modify for our needs the well-known *Longest Processing Time first (LPT)* scheduling scheme [22]. The LPT scheme enjoys several near-optimality properties [22] and is simple to implement. As its name hints, *LPT* processes the PEs in order of decreasing size. The intuition is that by doing so this greedy scheme will dispense with the largest PEs in the beginning, and then "recover" the load balance by dealing with the smallest PEs in the end. So we order the PEs by size, and reindex so that $\text{SIZE}(S_1) \geq \cdots \geq \text{SIZE}(S_t)$. *LPT* initializes the "current used capacity" B'_i of each host h_i to be $B'_i \leftarrow 0$. At any point, it processes the next PE, say S_i, and assigns it to a host, say h_j, that would have the minimum resulting utilization if assigned there. More formally, it assigns S_i to a host $\pi(S_i) = h_j$ such that

$$h_j = \text{argmin}_{h_k \in \mathcal{H}} \frac{B'_k + \textsc{size}(S_i)}{B_k}.$$

It then updates the current used capacity of host h_j by setting $B'_j \leftarrow B'_j + \textsc{size}(S_i)$. So at each stage the current used capacity is simply the sum of the sizes of the PEs assigned to it. The tentative assignment is feasible if, after the last iteration, $\sum_{S \in \mathcal{S}:\pi(S)=h_j} \textsc{size}(S) = B'_j \leq B_j$ holds for all $h_j \in \mathcal{H}$. If the assignment is feasible, COLA outputs that the current PEs and passes the control to the post-processor. Otherwise, COLA must split another PE. This involves the oracle and the graph partitioner.

Oracle. The oracle decides the next PE to split, and it is very simple. It simply returns that PE with more than one operator which has the largest size. A reasonable alternative would be to split the largest size multi-operator PE assigned to the most over-utilized host. Splitting large PEs is obviously an intuitively good strategy. As a side benefit it will tend to minimize the number of calls to the graph partitioner, helpful because each such call adds to the overall communication cost.

Graph Partitioner. The graph partitioner is the central component of the COLA algorithm. Given a PE S, its role is to determine how to split it into two non-empty PEs, say S_1 and S_2. It bases its decision on two objectives:

1. to minimize the communication cost between the resulting PEs S_1 and S_2, and
2. to avoid highly unbalanced splits such that $\textsc{size}(S_1)$ is either very large or very small as compared to $\textsc{size}(S_2)$.

To achieve this, we use the following well-studied problem, called the *minimum-ratio cut* or *sparsest cut* problem. Given a graph $H = (V_H, E_H)$ with vertex-weights $w_v \geq 0$ and edge-weights $w_e \geq 0$, find a cut (S_1, S_2) where $S_2 = V_H \ominus S_1$ such that the following ratio, also called the *sparsity*, is minimized:

$$\frac{w(\delta(S_1))}{\min\{w(S_1), w(S_2)\}}. \tag{2}$$

This objective minimizes the weight of the cut $w(\delta(S_1))$ while favoring the "balanced" cuts for which $\min\{w(S_1), w(S_2)\}$ is large.

Since the sparsest cut problem is NP-hard [23], we use an algorithm of Leighton and Rao [24] to find an approximate solution. We choose their algorithm since it is efficient to implement and provably finds a cut with sparsity within a factor that is logarithmic in the number of operators of the optimum sparsity. We outline their approach here. They first set up a linear programming (LP) formulation of the sparsest cut problem as follows. One can think of the graph H as a flow network where vertices are sources and sinks and the edges $e \in E_H$ are "pipes" that have flow capacity w_e. The LP encodes the following flow problem. Route a demand of $w_u \cdot w_v$ between each pair of vertices $u, v \in V_H$, possibly split along several paths, and minimize the maximum "congestion" on any edge. In other words, minimize $\max_{e \in E_H} f_e/w_e$, where f_e denotes the flow sent on edge $e \in E$. Intuitively, a cut (S_1, S_2) with a small ratio (2) will have edges with high congestion, since the capacity $w(\delta(S_1))$ of the cut is small compared to the total demand $w(S_1) \cdot w(S_2)$ that needs to be routed across the cut. The cut is then identified from the fractional solution of the LP using the above intuition. We omit the details from here and refer the reader to Leighton and Rao [24].

Because finding the solution to the LP can be slow even with the best linear solver packages, we implement this step with a well-known combinatorial algorithm [25] that approximates the solution to the multicommodity flow LP.

Post-processor. The post-processor performs certain "greedy" PE merges in order to improve the solution quality without violating the property that the partitioning has a feasible assignment to hosts. The idea is to partly correct for the possibly less than perfect ordering of the graph partitioning iterations. It first determines if a pair of PEs, say S_i and S_j, *can* be merged, as follows. It tentatively merges S_i and S_j into a single PE $S_i \cup S_j$. If the resulting partitioning has a feasible host-assignment using the LPT scheme, it marks this pair of PEs as "mergeable". It then greedily merges that pair of mergeable PEs which gives the maximum reduction in the total communication cost. This process is repeated until there are no pairs that can be merged, and the resulting PEs are the output of the Basic COLA scheme.

3 Advanced COLA

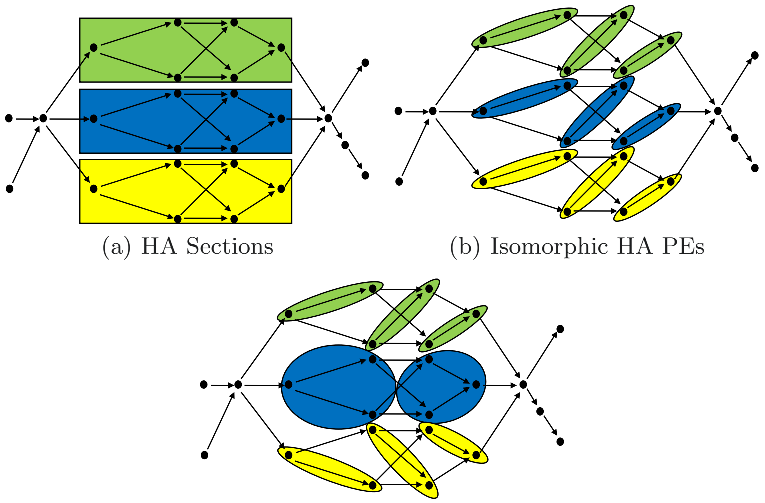
In order to make COLA useful for a wide variety of scenarios, it should allow the user to guide or constrain the fusion process. This version supports six such types of constraints, and it also considers a more complex objective function.

3.1 User-Defined Fusion Constraints

We have incorporated the following six types of constraints, and for each we offer motivating examples.

1. *Resource matching*: An operator may be allowed to be assigned to only a subset of the hosts. The rationale here is that some operators may need a resource or a performance capability not present on all hosts.
2. *PE colocation*: Two operators may be required to be fused into the same PE. Motivation includes the sharing of some per-process resource, such as a JVM instance or some other language-binding runtime.
3. *Host colocation*: Two operators may be required to be assigned to the same host. Clearly, PE colocation implies host colocation, but the reverse need not be true. As motivation, two operators may wish to share a host license, local files, or have shared memory segments.

4. *PE exlocation*: Two operators may be required to be fused into separate PEs. This may allow some work to continue if a PE crashes.
5. *Host exlocation*: Two operators may be required to be assigned to separate hosts. In this case, host exlocation implies PE exlocation, but not the reverse. Motivation for host exlocation includes a common per-process resource requirement for which a single host would be insufficient.
6. *High availability*: In order to support the notion of hot standbys a subgraph of the overall operator data flow graph may be identically replicated several times. See Figure 3(a), where there are three subgraph replicas. The constraint requires that the fused PEs respect this subgraph in the sense that they are either entirely contained within a single replica or do not intersect with any replicas. Figures 3(b) and 3(c) present two feasible PE fusion solutions; each shaded subsection corresponds to a PE. High availability constraints must also ensure that any PE contained within one replica will not be assigned to the same host as a PE contained within another replica. Additionally, one may optionally insist that the PEs within one replica have the identical structures as those within the other replicas. An example of PEs chosen with this *isomorphic* condition turned on is shown in Figure 3(b). An example of PEs chosen with the isomorphic condition switched off is shown in Figure 3(c). In either case, there are implied host exlocation constraints for all pairs of differently shaded PEs. The motivation for all of this is, as the name implies, high availability: If the work in one replica cannot be done, perhaps because of a host failure, there will likely be immediate backups available on disjoint hosts.



(a) HA Sections

(b) Isomorphic HA PEs

(c) Non-Isomorphic HA PEs

Fig. 3. High Availability

One could also think of two additional constraints, called *PE dedication* and *host dedication*. PE dedication would mean that an operator must be its own PE. Host dedication would mean that an operator must be its own PE *and* assigned alone on a host. Thus, host dedication implies PE dedication. Both these constraints can be easily incorporated via a small change to the COLA preprocessor and the addition of PE exlocation and host exlocation constraints. As a result, we do not treat these as separate constraints in COLA, even though we could expose them as constraints to the user.

3.2 Problem Formulation

The two somewhat competing goals are to ensure that

(iii). the maximum utilization $U = \max_{h_i \in \mathcal{H}} \sum_{S \in \mathcal{S}: \pi(S) = h_i} \text{SIZE}(S)/B_i$ is minimized, and

(iv). the overall communication cost $C = \sum_{S \in \mathcal{S}} w(\delta(S))$ is minimized.

Assuming the maximum utilization in expression (iii) is less than or equal to 1, which we will require, this expression is simply a more quantifiable version of the scheduling feasibility condition employed in the Basic COLA scheme. As before we will omit a discussion of how we handle pathological cases in which this scheduling feasibility is not possible.

We will handle both goals simultaneously by minimizing an arbitrary user-defined function $f(U, C)$ of U and C. This function can (and typically will) be as simple as a weighted average of the two metrics. It represents the tradeoff of the scheduling flexibility measured in expression (iii) with the efficiency measure in expression (iv).

Our final solution will obey:

- the six types of constraints, namely resource matching, PE colocation, host colocation, PE exlocation, host exlocation and high availability.
- the scheduling feasibility constraint.

We will call a solution which meets the six types of constraints *valid*, regardless of whether the solution satisfies the scheduling feasibility constraint. A valid solution which also satisfies the scheduling constraint will be known as *feasible*, as is standard.

3.3 Solution Approach

The pseudocode for the Advanced COLA scheme is given in Figure 4. First we give a high-level overview of our approach. We build upon the algorithm for Basic COLA, though we must add and modify many steps. There is a pre-processor, as before. It is augmented to resolve the PE colocation constraints. It also partially handles the HA constraints. Depending on whether or not HA constraints exist there may be multiple PEs rather than a single PE by the end of the pre-processing stage.

```
– Run Pre-processor
– Phase 1. Repeat
    • Compute the communication cost $c$ of the current partitioning
    • If PE exlocation constraints are satisfied, go to Phase 2
    • Use Oracle for phase 1 to find a PE $p$ to split next
    • Use Graph Partitioner for phase 1 to split $p$ into two PEs
– Phase 2. Repeat
    • Use PE scheduler to compute a schedule with utilization $u$
    • If the schedule is valid, go to Phase 3
    • Use Oracle for phase 2 to find a PE $p$ to split next
    • Use Graph Partitioner for phase 2 to split $p$ into two PEs
    • Compute the communication cost $c$ of the current partitioning
– Phase 3. Repeat
    • Let $s \leftarrow f(u, c)$
    • If the schedule is feasible, go to Phase 4
    • Use Oracle for phase 3 to find a PE $p$ to split next
    • Use Graph Partitioner for phase 3 to split $p$ into two PEs
    • Compute the communication cost $c$ of the current partitioning
    • Use PE scheduler to compute a schedule with utilization $u$
– Phase 4. Repeat
    • Use Oracle for phase 4 to find a PE $p$ to split next
    • Use Graph Partitioner for phase 4 to split $p$ into two PEs
    • Compute the communication cost $C$ of the current partitioning
    • If $C > (1 + T)c$, go to Post-processor
    • Use PE scheduler to compute a schedule with utilization $U$
    • Let $S \leftarrow f(U, C)$
    • $s \leftarrow \min\{s, S\}$
– Run Post-processor
```

Fig. 4. Advanced COLA Pseudocode

In the main body of our algorithm we solve the problem iteratively, as we did in the basic scheme. In each iteration we employ, as needed, a PE scheduler, an oracle to determine which PE to split next, and a graph partitioner to split that PE. However, the main body of the Advanced COLA scheme is composed of four successive phases of the iterative process. These phases are similar to each other, but not quite identical. During phase 1 the PE exlocation constraints are resolved. During phase 2 the host colocation, host exlocation and high availability constraints are resolved, which means that the solution at this point will be valid. Alternatively, COLA will have shown that there is no valid solution, because the graph partitioner will have split the operator flow graph all the way down into singleton operators without reaching validity. The user will be notified of this, and COLA will terminate. An important property of validity is that it will *persist* as we continue the graph partitioning process. (To see this, consider a single split of a PE in a valid solution, and consider the scheduling assignment in which the two new PEs are assigned to their previous host, and all other PEs are assigned to their previous hosts as well. This assignment also satisfies the six types of constraints. A corollary of this persistence property is that the existence of a valid solution can be determined by employing our iterative partitioning scheme.) In the normal case that a valid solution exists, the scheme continues. During phase 3 the scheduling feasibility constraints will be resolved. This means that we do have a feasible solution to the COLA problem. Denote the utilization at the end of phase 3 by u, and the overall communication cost by c. We can compute the objective function as $s = f(u, c)$. Note that the overall communication cost is monotonic: It increases with every new graph partitioning. We continue the iterative process *past* this point, into phase 4, and at each stage we will compute a new utilization U and a new overall communication cost C. Now scheduling feasibility does *not* necessarily persist as we split PEs, because the sizes of the PEs increase. The new solution is *likely* to be scheduling feasible, because of the the increased sizes should be counterbalanced by increased scheduling flexibilty. If the solution is scheduling feasible, that is, if $U \leq 1$, we check to see if $S = f(U, C) < s$. If so, we replace s by S, and we have found an improved solution. When do we stop the iterative process? The answer is that we will constrain the overall communication cost to be within a multiplicative user-input threshold T of the cost c of our first feasible solution: $C \leq (1 + T)c$. We stop when this condition fails, or when we have reached the bottom of the binary tree, so that all PEs are single operators. The value of T determines how much the algorithm is willing to compromise on overall communication cost in an attempt to find more scheduling flexible solutions. For instance, if $T = 1$, then the algorithm will continue to find more scheduling flexible solutions until the communication cost of the current solution (C) is *twice* the cost of the first feasible solution (c). On the other hand, if $T = 0$, then the algorithm skips phase 4 completely. Finally there is a post-processor to greedily improve the solution.

Figure 5 shows the four iterative phases of the Advanced COLA scheme. At the end of each phase we are further down the binary tree. The final solution, denoted in the figure with stars, occurs at some point in the tree between the end of phases 3 and 4.

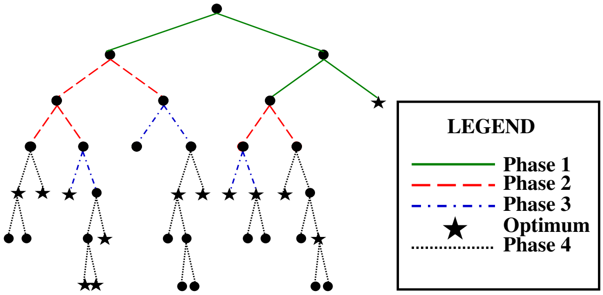


Fig. 5. Iterative Algorithmic Phases

Pre-processor. The pre-processor fuses unconstrained adjacent operators according to the conditions of Lemma 1. It also fuses PE colocated operators. And it separates HA replicas into separate PEs. So there will be one PE for each HA replica, plus potentially a catchall PE for all operators not part of any HA replicas. Recall Figure 3(a). If the isomorphic condition is turned on we will also replace each relevant operator cost with the average values of the corresponding operators across all the replicas. Similarly, we will replace each relevant communication cost with the average values of the corresponding streams across all the replicas. These values will probably be close in any case, but the reason for doing this is overall robustness, as will become clear below in the description of the graph partitioner. Finally, we will mark each relevant pair of PE replicas as host-exlocated, and continue to the main body of the scheme.

PE Scheduler. The component of the Basic COLA scheme that changes most relative to that of the basic algorithm is the PE scheduler. It is not needed in phase 1, but is used in phases 2 through 4. The *LPT* algorithm of the Basic COLA scheme is very fast, the complexity typically being dominated by the reordering of the PEs. It is an effective and robust scheme in the absense of additional constraints. *LPT* can certainly be adapted easily to handle resource matching, host colocation and host exlocation. But it will produce far lower quality solutions in a scenario with many such constraints, because it is a one-pass greedy scheme. We therefore formulate and solve the problem as a more computationally expensive integer program (IP). Specifically we define decision variable $x_{p,h}$ to be 1 if PE p is assigned to host h, and 0 otherwise. Let R_p denote the set of resource matched hosts for PE p. Host colocation defines an equivalence relation which we denote by $\equiv_{HC}$. Host exlocation does *not* determine an equivalence relation, but we define the set HE to be the set of pairs (p_1, p_2) of exlocated PEs. We then solve the following:

$$\text{Minimize} \quad \max_h \sum_p \textsc{size}(S_p) \cdot x_{p,h}/B_h \tag{3}$$

$$\begin{aligned} \text{subject to} \qquad x_{p,h} &= 0 && \text{if } \; h \notin R_p, && (4)\\ x_{p_1,h} &= x_{p_2,h} && \forall \; h, \; \text{ if } \; p_1 \equiv_{HC} p_2, && (5)\\ x_{p_1,h} + x_{p_2,h} &\leq 1 && \forall \; h, \; \text{ if } \; (p_1, p_2) \in HE, && (6)\\ \sum_h x_{p,h} &= 1 && \forall \; p, && (7)\\ x_{p,h} &\in \{0, 1\} && \forall \; p, h. && (8) \end{aligned}$$

The objective function 3 measures the maximum utilization of any host. Constraint 4 enforces the resource matching constraints. Constraint 5 enforces the host colocation constraints. Constraint 6 enforces the host exlocation constraints. Constraint 7 ensures that each PE is assigned to one host. Finally, constraint 8 ensures that the decision variables are binary.

Oracle. In phase 1 the oracle will return any PE which fails to meet a PE exlocation constraint. This means there are at least two operators in the PE which are supposed to be PE exlocated. The choice is otherwise irrelevant, since all such constraints will need to be satisfied by the end of the phase. In phases 2 through 4 the oracle is identical to that of the Basic COLA scheme.

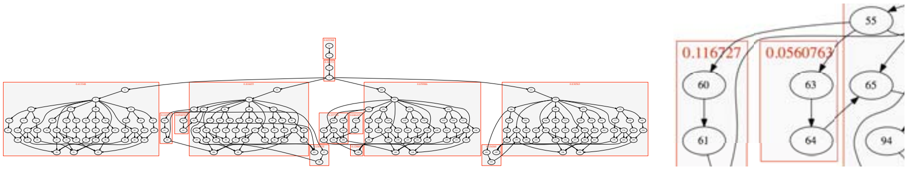
Graph Partitioner. Here there are two differences from Basic COLA. One is specific to all four phases, and relates to the HA constraints with isomorphic condition on. The other is relevant, as before, only to phase 1.

- If the isomorphic condition is on and the graph partitioner splits a PE that is part of one replica the scheme will force this solution immediately on all the other replicas. Since we have chosen averages it does not matter which replica is chosen to be split first. Furthermore, the graph partitioning solution should be relatively close to optimal for all replicas. If the isomorphic condition is off each replica can be split independently.
- The graph partitioner approach in phase 1 is again slightly modified. We wish to encourage the PE exlocated operators to be split by the graph partitioning process. So we add additional demand between all such operator pairs, which makes them more likely to be split, and solve the revised graph partitioning problem as before.

Post-Processor. The postprocessor in the Advanced COLA scheme is identical to that of the basic scheme.

4 COLA Experiments

To evaluate how COLA performs in practice, we use a job called VWAP that runs on System S. The job VWAP [26] represents a financial markets scenario in which a stream of real-time quotes is processed to detect bargains and trading opportunities. Figure 6 shows the directed graph G corresponding to this job, as well as a typical operator fusion computed by COLA. The boxes correspond to PEs and the numbers in the boxes correspond to the sizes of PEs in terms of CPU fractions. (Figure 6(b) is an enlarged view of a portion of Figure 6(a).) This job consists of 200 operators and 283 arcs.



(a) The VWAP Job (b) Enlarged view

Fig. 6. COLA Operator Fusion

The experiments discussed in this paper were performed using a SYSTEM S deployment on a cluster consisting of IBM BladeCenters running Linux 2.6.9. We employed between 4 and 7 blades, each having dual-CPU, dual-core 3.2GHz Intel Xeon processors with 4GB of RAM. The blades are in the same rack, and are inter-connected using a high-speed 20GB/s backplane. These homogeneous blades were reserved for these experiments. Thus, no other processes were allowed to use the blade resources.

The COLA fusion strategy was compared against two alternative fusion strategies:

- NONE: No fusion. Thus each operator lies in a distinct PE.
- FINT: This FINT fusion strategy, proposed in [13], also takes the operator and communication costs as input. It employs a *bottom-up* rather than a top-down approach to compute a fusion. It initially places all operators into distinct PEs and iteratively fuses them into larger PEs till some criteria is met.

Another natural fusion strategy is to fuse all operators into a single PE. However this typically results in a highly computationally intensive PE yielding very low throughput values. It is therefore not evaluated further in our experiments.

We evaluate a given fusion alternative by job *throughput*. This is a measure of how much data (in Mbps) is processed by the job. It is intended to be a measure of the job's "effective capacity". Each VWAP experiment is characterized by the fusion strategy employed and the number of blades used.

Both the COLA or FINT fusion strategies require operator and communications costs as input. The good news is that SYSTEM S incorporates efficient *profiling* methodology [13]. But there is bad news as well. Specifically, to use the profiling mode in an application run, we need to choose an operator fusion. To create an operator fusion, on the other hand, we need the profiling data. This is a chicken and egg problem. Since the NONE fusion strategy does not require operator or communication costs as input, we first run this fusion in the profiling mode. However, the NONE fusion is observed to yield low throughput values and hence only moderately useful profiling data. So we employ an iterative approach, as follows: Using the NONE profiling data, we compute the fusion, referred to as iteration 1, again in profiling mode. Then, using the new profiling data, we compute the fusion again, this time using the *new* profiling data. This becomes iteration 2, and we continue in this manner. This iterative approach is used for both FINT and COLA fusions.

Furthermore, we adopt one additional heuristic in the iterative process. Since the early fusions yield lower throughputs, we compensate for this in COLA as follows. In iteration 1, we scale the CPU capacities B_i of the hosts by a factor $\gamma < 1$. In the experiments described below, we set $\gamma = 0.5$ in iteration 1. Then we gradually increase γ from 0.5 to 1.0 by 0.1 through 6 subsequent iterations, since we expect to obtain more and more accurate estimates on the operator and communication costs.

Figure 7 demonstrates the benefit of multiple iterations, particularly for COLA. Fortunately, only a few seem to be required. The COLA fusion strategy shows

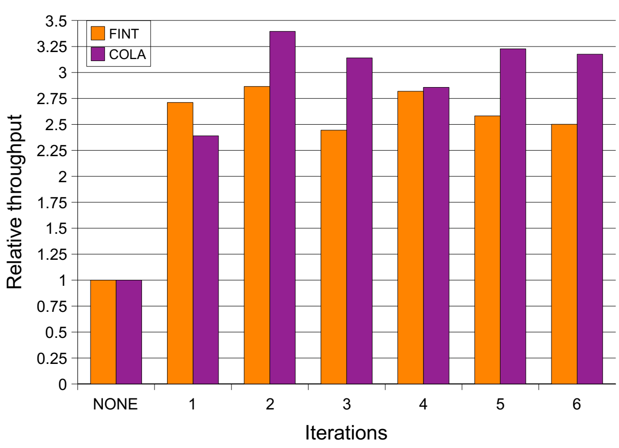


Fig. 7. Profiling iterations for FINT and COLA on 5 blades. The y-axis represents the relative throughput of the runs, scaled so that the throughput of NONE run is 1.

Table 1. Theoretical Comparison of FINT and COLA

VWAP1	NONE		FINT		COLA	
Blades	Cut size	PE size	Cut size	PE size	Cut size	PE size
4	0.3792	0.3218	0.0661	0.5353	0.0332	0.4990
5	0.3703	0.2980	0.0441	0.5476	0.0587	0.4676
6	0.7236	0.8169	0.2698	0.8169	0.1998	0.7560
7	0.6833	0.6697	0.2492	0.6697	0.1860	0.6666

VWAP2	NONE		FINT		COLA	
Blades	Cut size	PE size	Cut size	PE size	Cut size	PE size
4	0.3544	0.3677	0.0618	0.4185	0.0618	0.4185
5	0.5740	0.6030	0.2577	0.6030	0.1530	0.5547
6	0.5571	0.5764	0.3367	0.5764	0.1552	0.5294
7	0.5888	0.5813	0.2600	0.5813	0.1587	0.5813

a significant increase in the throughput in the first two iterations. We typically see that COLA reaches its maximum throughput value quickly.

In Table 1, we show how COLA compares to FINT and NONE in terms of both cut size and maximum PE size. We study two versions of the VWAP job: VWAP1 consists of 200 operators and 283 arcs, while VWAP2 consists of 217 operators and 315 arcs. The cut size is the measure of the total cost of sending traffic between PEs. The larger the cut size, the more CPU cycles are being devoted to sending data. The intuition is that a high quality fusion has a low cut size without making any PEs too big to fit on a processor. We can see that the cut sizes for COLA are signifcantly lower than the cut sizes for FINT, and COLA often maintains a maximum PE size which is smaller than that of FINT.

Table 2. Throughput Values for VWAP Runs with FINT and COLA

VWAP1 blades	NONE	FINT			COLA		
		1st	1st-local	Max	1st	1st-local	Max
4	99.4 (1)	273 (2.75)	295 (2.96)	295 (2.96)	284 (2.86)	286 (2.88)	303 (3.05)
5	97.2 (1)	263 (2.71)	279 (2.87)	279 (2.87)	232 (2.39)	330 (3.40)	330 (3.40)
6	189 (1)	286 (1.51)	286 (1.51)	293 (1.55)	280 (1.48)	379 (2.00)	391 (2.06)
7	179 (1)	268 (1.50)	322 (1.80)	336 (1.88)	267 (1.50)	349 (1.95)	363 (2.03)

VWAP2 blades	NONE	FINT			COLA		
		1st	1st-local	Max	1st	1st-local	Max
4	92.6 (1)	212 (2.29)	212 (2.29)	212 (2.29)	187 (2.02)	237 (2.56)	249 (2.69)
5	150 (1)	219 (1.47)	219 (1.47)	258 (1.73)	229 (1.53)	332 (2.22)	332 (2.22)
6	146 (1)	227 (1.55)	260 (1.78)	260 (1.78)	238 (1.63)	346 (2.37)	346 (2.37)
7	154 (1)	226 (1.46)	369 (2.39)	369 (2.39)	253 (1.64)	362 (2.35)	362 (2.35)

Next we discuss throughput, our prime practical metric. Table 2 presents the throughput values for NONE and various iterations of FINT, and COLA. Since the throughput values from different iterations can be quite different, a natural question to ask is which iteration one should finally chose. To this end, we evaluate several different choices: the first iteration (1st); the first iteration that achieves a local maximum throughput (1st-local); and the iteration (among the first few) that achieves the overall maximum throughput (Max).

We again study two versions, VWAP1 and VWAP2, as described above. The rows represent distinct sets of experiments run on different number of reserved blades given in the first column. For each such set of experiments, 6 iterations were used for each of FINT and COLA. The throughput values are truncated to three significant digits. The numbers in parentheses represent the relative gain over the NONE fusion, i.e., throughput values scaled so that the throughput of the NONE fusion is 1.

The throughput values of both FINT and COLA are significantly higher than those of NONE, while the throughput values of COLA for 1st-local and Max are usually higher than those of FINT.

5 Conclusions and Future Work

In this paper we have described and solved an important operator fusion problem which arises naturally in System S. Our scheme works in heterogeneous processor environments, and supports a wide variety of real-world constraints. We believe that the COLA scheme is mathematically novel and interesting. Initial experiments support the value of our approach to the SPADE operator fusion problem. We list some future enhancements.

– As may be seen in Table 2, for example, adding more hosts to the COLA problem does not necessarily improve performance. We are thinking about approaches in which COLA might automatically choose fewer hosts than those offered to it. This issue is easier in the case of the basic scheme with homogeneous hosts.

- The function $f(U, C)$ used by the Advanced COLA scheme could be a weighted average of U and C. But we have not described how to pick these weights. We believe the appropriate weights could be learned by examining the effects of alternative choices on throughput.
- It appears to be useful to have the Advanced COLA scheme quickly decide if the six types of real-world constraints allow a feasible solution or not. (This would allow a user to modify inconsistent constraints and resubmit.) Our current scheme provides an answer this feasibility question, but not necessarily in the fastest timeframe. We plan to modify our approach to better handle this issue.
- We currently consider processing hosts as single units. But in today's environment they are often composed of several multi-core processors. Our LPT and IP PE scheduling scheme does not consider this processor hierarchy at present. We believe both schemes can be enhanced to do so, and we will be experimenting to see if such an approach is valuable.

References

1. ThomsonReuters, http://ar.thomsonreuters.com
2. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The design of the Borealis stream processing engine. In: Proceedings of Conference on Innovative Data Systems Research (2005)
3. Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R., Zdonik, S.: Retrospective on Aurora. VLDB Journal (2004)
4. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: Proceedings of Conference on Innovative Data Systems Research (2003)
5. Girod, L., Mei, Y., Newton, R., Rost, S., Thiagarajan, A., Balakrishnan, H., Madden, S.: XStream: A signal-oriented data stream management system. In: Proceedings of the International Conference on Data Engineering (2008)
6. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., Widom, J.: STREAM: The Stanford stream data manager. IEEE Data Engineering Bulletin 26 (2003)
7. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, p. 179. Springer, Heidelberg (2002)
8. Zdonik, S., Stonebraker, M., Cherniack, M., Cetintemel, U., Balazinska, M., Balakrishnan, H.: The Aurora and Medusa projects. IEEE Data Engineering Bulletin 26 (2003)
9. Coral8 (2007), http://www.coral8.com
10. StreamBaseSystems (2007), http://www.streambase.com/

11. Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Selo, P., Park, Y., Venkatramani, C.: SPC: A distributed, scalable platform for data mining. In: Proceedings of the Workshop on Data Mining Standards, Services and Platforms (2006)
12. Douglis, F., Palmer, J., Richards, E., Tao, D., Tetzlaff, W., Tracey, J., Yin, J.: Position: Short object lifetimes require a delete-optimized storage system. In: ACM SIGOPS European Workshop (2004)
13. Gedik, B., Andrade, H., Wu, K.L.: A code generation approach to optimizing high-performance distributed data stream processing. In: Proceedings of the ACM International Conference on Information and Knowledge Management (2009)
14. Gedik, B., Andrade, H., Wu, K.L., Yu, P.S., Doo, M.: SPADE: The System S declarative stream processing engine. In: Proceedings of the ACM International Conference on Management of Data (2008)
15. Hildrum, K., Douglis, F., Wolf, J., Yu, P.S., Fleischer, L., Katta, A.: Storage optimization for large-scale stream processing systems. ACM Transactions on Storage 3 (2008)
16. Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., Venkatramani, C.: Design, implementation and evaluation of the linear road benchmark on the stream processing core. In: Proceedings of the ACM International Conference on Management of Data (2006)
17. Jacques-Silva, G., Challenger, J., Degenaro, L., Giles, J., Wagle, R.: Towards autonomic fault recovery in System-S. In: Proceedings of Conference on Autonomic Computing (2007)
18. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.L., Fleischer, L.: SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 306–325. Springer, Heidelberg (2008)
19. Wu, K.L., Yu, P.S., Gedik, B., Hildrum, K.W., Aggarwal, C.C., Bouillet, E., Fan, W., George, D.A., Gu, X., Luo, G., Wang, H.: Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In: Proceedings of the International Conference on Very Large Data Bases Conference (2007)
20. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.L.: Job admission and resource allocation in distributed streaming systems. In: Workshop on Job Scheduling Strategies for Parallel Processing, IPDPS (2009)
21. Garey, M., Johnson, D.: Computers and Intractability. W.H. Freeman and Company, New York (1979)
22. Pinedo, M.: Scheduling: Theory, Algorithms and Systems. Prentice Hall, Englewood Cliffs (1995)
23. Síma, J., Schaeffer, S.E.: On the NP-completeness of some graph cluster measures. In: Wiedermann, J., Tel, G., Pokorný, J., Bieliková, M., Štuller, J. (eds.) SOFSEM 2006. LNCS, vol. 3831, pp. 530–537. Springer, Heidelberg (2006)
24. Leighton, F.T., Rao, S.: Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. J. ACM 46, 787–832 (1999)
25. Garg, N., Könemann, J.: Faster and simpler algorithms for multicommodity flow and other fractional packing problems. SIAM J. Comput. 37, 630–652 (2007)
26. Andrade, H., Gedik, B., Wu, K.L., Yu, P.S.: Scale-up strategies for processing high-rate data streams in System S. In: Proceedings of the International Conference on Data Engineering (2009)