SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems

Joel Wolf[1], Nikhil Bansal[1], Kirsten Hildrum[1], Sujay Parekh[1], Deepak Rajan[1], Rohit Wagle[1], Kun-Lung Wu[1], and Lisa Fleischer[2]

[1] IBM T.J. Watson Research Center, Hawthorne, NY 10532, USA
[2] Dartmouth College, Hanover, NH 03755, USA
{jlwolf,nikhil,hildrum,sujay,drajan,rwagle,klwu}@us.ibm.com,
lkf@dartmouth.edu
http://www.ibm.com

Abstract. This paper describes the *SODA* scheduler for *System S*, a highly scalable distributed stream processing system. Unlike traditional batch applications, streaming applications are open-ended. The system cannot typically delay the processing of the data. The scheduler must be able to shift resource allocation dynamically in response to changes to resource availability, job arrivals and departures, incoming data rates and so on. The design assumptions of *System S*, in particular, pose additional scheduling challenges. *SODA* must deal with a highly complex optimization problem, which must be solved in real-time while maintaining scalability. *SODA* relies on a careful problem decomposition, and intelligent use of both heuristic and exact algorithms. We describe the design and functionality of *SODA*, outline the mathematical components, and describe experiments to show the performance of the scheduler.

Key words: stream processing, scheduling, admission control, flow balancing

1 Introduction

The authors of this paper are involved in an ambitious project, started in 2003, known as *System S* [1–6]. *System S* is highly scalable distributed computer system middleware designed to handle complex jobs involving enormous quantities of streaming data. A prototype of this system has been built and continues to evolve.

Early examples of distributed stream processing systems have mostly involved relational databases augmented with streaming operations [7–10]. In contrast, *System S* supports arbitrarily complex processing, both in terms of the design of the basic units of computational software, known as *processing elements* (PEs), and the way in which these PEs are interconnected via streams. Additionally, when designing *System S* a key assumption was that the offered load would far exceed system capacity much of the time. Therefore it is expected that the processing nodes in *System S* will need to be utilized as close to fully

as possible. Scheduling in such a complex, overloaded streaming environment is challenging problem and requires novel solution techniques.

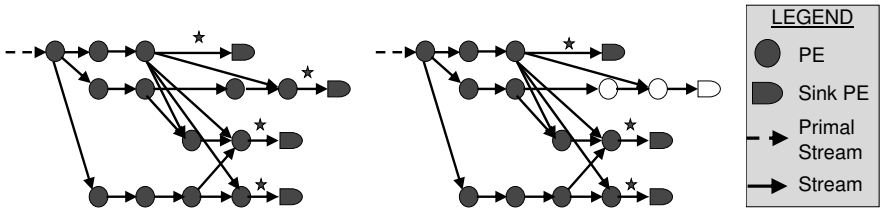
This overview paper describes the *System S* scheduler, known as *SODA*. (*SODA* stands for *Scheduling Optimizer for Distributed Applications*.) We motivate the design of the scheduler, emphasizing its objectives and functionality. We describe the four major mathematical components of *SODA* at a relatively high level: Space considerations prevent us from giving complete details, but we refer the interested reader to [11]. We sketch three infrastructure components which provide critical *SODA* input. Finally, we describe a number of experiments which illustrate *SODA* performance.

1.1 *SODA* Objectives and Functionality

In contrast with more traditional jobs, stream processing jobs are typically open-ended. A stream job could in theory continue to execute as long as input data are available. As a result, standard scheduling metrics involving completion times and/or makespan are no longer relevant. Figure 1(a) shows the *data flow graph* of a typical *System S* stream processing job. The PEs of the job are the nodes in the digraph, and streams, in turn, correspond to the directed arcs. Such digraphs are typically acyclic, as is the case in this figure. Thus, given a topological ordering, processing of data by the job will proceed from left to right. One can make a reasonable analogy to a factory assembly line. Raw packets enter in *primal* streams at the left. (Primal streams originate externally to systems.) Processing proceeds through the various PEs along the way, with streams carrying the progressively more "finished" packets. The final product or products is produced at the right of the data flow graph, where *sink PEs* consume the final products and possibly interface with the external (non-*System S*) world to deliver these results. The final streams flowing into the sink PEs are called *terminal streams*, and denoted with a star in Figure 1(a). This motivates our choice of objective function: *SODA* schedules to maximize a utility-theoretic function based on the "importance" measured at the terminal streams of the data flow graphs. We will give a formal definition of importance in Section 3, but it is typically based on a quantity or quality measure of the stream.

Each PE can only be expected to run satisfactorily if the processing power allocated to it is within some acceptable range. Thus the overloaded nature of *System S* motivates an important scheduler function: *SODA* must be prepared to reject some jobs. Otherwise some PEs may not be given their minimum acceptable allocations. Some distributed stream processing systems employ load shedding to deal with momentary processing node overload conditions [12], but we know of no other actual systems which consider job admission. When the system is frequently rather than rarely overloaded, shedding load is not enough. Job admission is essential.

Technically, *System S* may provide *SODA* with more than one data flow graph per job. Each such alternative data flow graph is known as a *template*. There might, for example, be a higher and a lower quality template. The natural tradeoff is that the higher quality template would require more processing



(a) Original (b) Weight change

Fig. 1. Job Data Flow Graph

resources than the lower quality one. The templates themselves could be very similar or very different. So an additional and novel function of *SODA* is the decision of which template to choose for each admitted job.

Optimizing the allocation of processing resources to the PEs in the chosen templates of the accepted jobs is extremely difficult for two reasons. The first reason is the highly interconnected (producer/consumer) nature of the PEs, potentially even across jobs. These PEs are *not* independent. The resources allocated to a PE which produces a stream affects the resources required for the PE(s) that consume that stream. Flow imbalances can lead on one hand to buffer overflows (and loss of data), and on the other to under-utilization of processing nodes. The second reason is again the overloaded nature of *System S*. In an underloaded system, flow imbalances simply matter less. A PE can use more of its allotted resources if needed because the resources are likely to be available. But in an overloaded system, there is no margin for error. Thus the scheduler must be parsimonious and carefully balance the allocated resources. We know of no other schedulers for distributed stream processing systems which perform this flow balanced resource allocation optimization.

Finally, *SODA* assigns the PEs in the chosen templates of the accepted jobs to processing nodes. Here there is a tradeoff between the load on the processing nodes and stream traffic on the network. Assigning two PEs connected by a stream to the same processing node eliminates the contribution of that stream to network traffic, but may contribute instead to overloading the processing node. So *SODA* attempts to achieve a balanced placement that does not overload either network links or node capacities. In fact, it attempts to minimize a weighted average of six separate metrics associated with processing loads on the nodes and traffic on the network links. The assignment problem is made more complex by the addition of many special constraints imposed by *System S*. These include, among many others, hardware constraints for certain PEs and nodes (*resource matching*), security and license constraints, constraints that pairs of PEs be placed together (*colocation*), or that pairs of PEs be placed on distinct nodes (*exlocation*). Of course, many PEs may share a node. *SODA* attempts to provide each PE with a fraction of the processing power of any node to which it is

assigned, matching as closely as possible the overall PE flow balancing goals already computed.

To summarize, the functionality of the *SODA* scheduler for *System S* includes:

- **Job admission**. *SODA* determines which jobs should be accepted and which jobs should be rejected.
- **Choice of templates**. For those jobs which are accepted *SODA* chooses the template alternative which is most appropriate for the amount of resources available.
- **PE resource allocation**. *SODA* determines how many resources to allocate to each PE in the chosen template of an accepted job.
- **PE fractional assignment**. *SODA* assigns each PE in the chosen template of an accepted job to fractions of one or more processing nodes.

Additionally, *SODA* optimizes two key metrics as it makes its scheduling decisions:

- **Importance**. *SODA* attempts to maximize a utility-theoretic measure of the "goodness" of the work in the system.
- **Resource utilization**. *SODA* attempts to balance the load across all resources (node processing capacity, network bandwidth) in the system by minimizing a weighted average of metrics that model resource utilization.

1.2 *SODA* Design Overview

Another original design requirement for *System S* was the ability to react quickly in a highly dynamic environment. Data rates may change suddenly and dramatically; new jobs may be submitted; jobs may be canceled. The available resources may also change: Processing nodes may go offline; new nodes may go online. Even the notion of what is important may change. The scheduler must be able to incrementally adjust the set of admitted jobs, the PE resource allocations and the fractional assignments to accommodate these changes.

For this reason *SODA* is an *epoch*-based scheduler. At the beginning of each epoch, *SODA* obtains as input a snapshot of the current system state, including the jobs running on the system and the jobs waiting to be admitted. It then computes for most of an epoch, finally outputting its scheduling decisions at the end of the epoch. That is, it produces a list of accepted and rejected jobs. For the accepted jobs it produces a choice of templates and a set of fractional allocations of the PEs to processing nodes. Those decisions are enforced by *System S* during the following epoch, and the entire process repeats indefinitely. Epoch lengths are a *SODA* settable parameter, but epochs on the order of a minute are typical. This is a reasonable compromise between the staleness of the input data and the time required for the mathematical components of *SODA* to make high quality decisions.

To make the scheduling problem tractable, each *SODA* epoch is divided into four mathematical phases. For reasonably sized *System S* installations they are

solved sequentially. Each of the four phases corresponds to a mathematical optimization module. The first two phases are known collectively as the *macro* model, while the second two are known as the *micro* model.

- The *macro model* chooses the jobs that will be admitted, the templates for those jobs, and the so-called *candidate* nodes to which the PEs in those jobs and templates can be assigned. These candidate nodes are a subset of the resource matched nodes, chosen to balance system load and simultaneously respect various constraints (security, licensing, colocation, exlocation and so on). The point is that candidate node choices made in the macro model are respected by the subsequent micro model, and this makes the decisions of the micro model easier and more effective.
- The *micro model* chooses the fractional allocations of the PEs in the jobs and templates that have been chosen by the macro model. Fractional allocations of PEs are 0 for a particular node *unless* that node has been chosen as a candidate node by the macro model. For this reason the *micro* model does not have to consider the difficult constraints handled in the macro model: They are satisfied automatically.

Within both the *macro* and *micro* model, the first (*quantity*) phase computes the resource allocation goals, and the second (*where*) phase computes the actual assignments. For this reason the four mathematical phases in *SODA* are known as *macroQ*, *macroW*, *microQ*, and *microW*, respectively. These decouplings make solving the individual optimization problems more efficient.

The remainder of this paper is organized as follows. In Section 2 we give an overview of *System S*. Section 3 contains a glossary of key new terms used by *SODA*. Understanding these terms is critical to following the overviews of the four mathematical components in Section 4. In Section 5 we describe experiments showing the performance benefits of the *SODA* scheduler. Section 6 contains a brief review of related work. Conclusions and future work are given in Section 7.

2 Overview of *System S*

We briefly describe some key components of *System S*. Readers are referred to [4,6] for more details. *System S* is distributed stream processing middleware, and its components provide efficient services to enable the simultaneous execution of multiple stream processing jobs on a large cluster of machines. A functional prototype of *System S* exists on a Linux cluster consisting of about 125 nodes interconnected by a Gigabit switched Ethernet network.

Aside from *SODA*, key run-time components of *System S* include the *Job Manager (JMN)* and the *Stream Processing Core (SPC)*. The JMN is a framework upon which job management, dispatching and node control are built. The JMN consists of a central orchestrator, a *Master Node Controller (MNC)*, and a *Resource Manager (RMN)*. Providing the execution and communication substrate for System S, the SPC consists of four major components: the *Dataflow Graph Manager (DGM)*, the *Data Fabric (DF)*, the *Node Controller (NC)*, and

the *PE Execution Container (PEC)*. The DGM determines stream connections among PEs and matches descriptions of output ports with the flow specifications of input ports. The DF is the distributed data transport component, consisting of a set of daemons, one on each node. The NC manages PEC agents, and each PEC manages one or more PEs. The PEC provides a run-time context and acts as a security barrier, preventing the user-written applications from corrupting the *System S* middleware as well as each other.

Each job can be described by one or more data flow graphs, as shown in Figure 1(a). The nodes in the directed graph correspond to the PEs, and the arcs to the streams. (One stream may show up as several arcs with the same source.) The PEs consume and produce data streams through their input and output ports, respectively. These data flow graphs are defined in a *job configuration* file, and specify how different PEs are to be connected via *flow specifications*. Stream connections are created between input and output ports based on a publish-subscribe model. *System S* dynamically determines the PE connections at run-time by matching stream data types with flow specifications. This allows PEs to discover new streams that match their flow specifications whenever such streams become available, allowing an application designer to avoid hard-wiring PE connections.

3 Glossary of Key New *SODA* Terms

SODA employs a number of terms that have very specific meanings to the scheduler. We list these below, with explicit definitions. The first two items, the *value function* and *weight*, are the key components of the third item, *importance*. Importance, in turn, is the metric that *SODA* tries to maximize. The fourth item, the *resource function (RF)*, is the atomic unit by which we iteratively compute this notion of importance. Finally, *rank*, the fifth item, is an orthogonal notion to importance: It is a priority metric assigned to each job; the lower the better. Jobs which produce little importance but have a low rank may get done *instead* of jobs which have more importance but have a higher rank.

Each derived stream produced by a potential *System S* job has a *value function* associated with it. This is an arbitrary non-negative real-valued function. The domain of this function might typically be the projected rate of the stream. Or it might instead be a stream quality measure, such as projected goodput. In theory it could be a cross product of a variety of quantity, quality and even other "goodness" measures. The definition is intentionally general, though early *SODA* instances have employed rate-based value functions. Also note that value functions which are 0 everywhere will typically predominate: Although the notion is also intentionally general we expect to see non-trivial value functions mostly on terminal streams of various jobs. These are, of course, the "end products" of *System S* work, and one would thus naturally want to measure goodness there.

Each derived stream produced by a potential *System S* job also has a *weight* associated with it. This is a non-negative real number. Non-trivial weights will also typically be quite sparse, as the weight may as well be 0 unless the stream

also has a non-zero value function. Weights are automatically assigned based on job topology unless explicitly set by the user.

Each derived stream produced by a potential *System S* job has an *importance* which is the product of the weight and the value function. Importance is therefore a function of the rate or quality of the stream, which in turn depends on the resources allocated to all the upstream PEs – those PEs which helped to produce the stream. The summation of this importance over all derived streams is the *overall importance* being produced by *System S*, and this is what *SODA* attempts to maximize. (Again, a large majority of streams will typically not contribute to this importance metric.) Consider Figure 1 again. In Figure 1(a), all starred streams have positive weights. But in Figure 1(b) the second weight has been changed to 0. It follows that the 2 PEs immediately upstream of that weight cannot do work which contributes to overall importance. *SODA* will therefore not allocate resources to them. (Other PEs, further upstream, do useful work in support of streams with positive weights. They may get fewer resources than they would in the previous figure, but not necessarily none.) Weights are thus an easy "knob" to turn on and off portions of a job and also a way to adjust relative importance.

If importance is the metric to be maximized, the natural question is how to compute it. The first part of the answer is as follows: Each derived stream s in *System S* (and by approximate terminology the PE that produces that stream) has an *RF* associated with it. The *RF* is multidimensional. If there are n input streams to the producer PE, then the *RF* has $n + 1$ input parameters. There is one parameter for each of the input streams, each with the same domain as the value function. These measure the goodness of the respective input streams. The final input dimension is the (computational) resources which may be allocated to the PE, in *millions of instructions per second (*mips*)*. The output of this function is again in terms of the same domain, and measures the goodness of stream s. See, for example, Figure 2. Assuming the domain to be rate-based, the *RF* for stream s_4 takes 4 parameters as input. The first three are the rates of streams s_1 through s_3, and the fourth is the *mips* allocated to PE 4. The output is the rate of stream s_4. The *RF* needs to be learned over time by a *SODA* infrastructure component known as the *Resource Function Learner (RFL)*. The *RFL* component provides crucial input data for *SODA* and is the subject of continuing research.

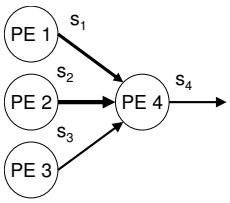
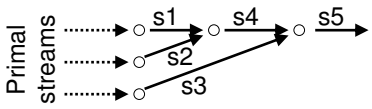
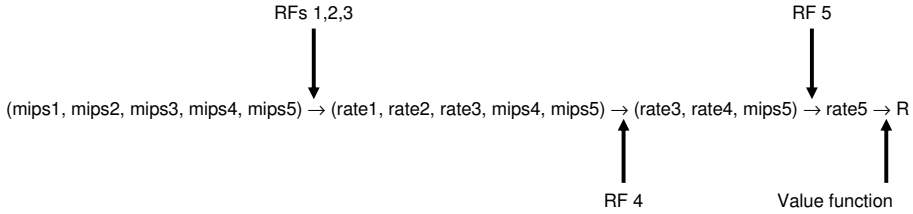


Fig. 2. The resource function for s_4 takes the *mips* of PE 4 and rates of s_1, s_2, and s_3 as input.

The second part of computing importance involves iteratively traversing the data flow graphs from "left" to "right", ending in a final value function calculation. Consider Figure 3. By topologically sorting [13] a directed acyclic graph, we can apply ready list scheduling [14] to compute the importance for stream s_5. In the figure three RFs are initially ready because they are fed by primal streams. So we obtain the rates at streams s_1 through s_3. Then an additional RF becomes ready (because its inputs have been computed), and we obtain the rate at stream s_4. Next we compute the rate at stream s_5. Finally we apply the weighted value function at s_5 to obtain importance. (*SODA* can also handle data flow graphs with cycles, but we omit details.)



(a) PE layout



(b) Calculating importance

Fig. 3. Calculation of Importance

Each job in *System S* has a *rank*, a positive integer which is used to determine whether the job should be run at all. The importance, on the other hand, determines the amount of resources to be allocated to each job that will be run. A lower job rank is better than a higher one. *SODA* admits jobs such that there is a specific job rank for which the following holds: All jobs with lower ranks are admitted, and all jobs with higher ranks are not admitted. Jobs with that rank may or may not be admitted, depending on the available resources and the importance associated with the (streams of the) jobs themselves. We call this property *rank-legality*. (This statement is a slight simplification, since one needs to account for inter-job dependencies.) Figure 4 shows job admission in two different load conditions. Each of the alternatives is rank-legal.

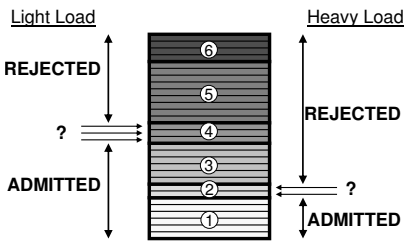


Fig. 4. Rank-Legal Job Admission

4 *SODA* Mathematical Components

In this section we describe, at a relatively high (qualitative) level, the four major mathematical components of *SODA*. Space limitations prevent a full exposition, but the interested reader is referred to [11] for complete details. The basic functionality of the components is as follows.

- *macroQ* decides which jobs to admit, which templates to choose, and the processing power goals for each PE in those jobs and templates.
- *macroW* computes the candidate processing nodes for the PEs given to it by *macroQ*.
- *microQ*, revises the processing power goals for the PEs in light of the candidate node decisions made by *macroW*.
- *microW* computes the fractional allocations of the PEs to the processing nodes based on the output of *macroW* and *microQ*.

Each *SODA* component has an internal *deadline*. Remember that *SODA* has slightly less than one epoch to solve the *macroQ*, *macroW*, *microQ* and *microW* problems. So the *SODA* scheduler itself has a scheduler.

4.1 *macroQ*

The *macro quantity* model, *macroQ*, finds a set of jobs to admit during the next epoch. For each admitted job it chooses a template from among the alternatives given to it. The jobs have ranks, and the jobs that are chosen by *macroQ* must respect the rank-legality constraint. Required jobs must be admitted. Minimum and maximum PE *mips* constraints must also be respected. The goal of the *macroQ* model is to maximize the projected importance of the streams produced by the admitted jobs and chosen templates. There is a total amount of processing power in the system, namely the sum of the power of all the processing nodes.

Thus *macroQ* becomes a resource allocation problem (RAP) [15]. We solve a discrete version of the RAP. So we divide the total processing power of the system into units of equal sized resolution. Also assume a specific rank-legal set of jobs and templates. The data flow graphs of these jobs and templates may be interconnected, and we form a digraph by gluing them together appropriately.

macroQ is a divide and conquer algorithm, and the division is based on the partitioning of this digraph into weak components.

So consider for the moment one such component. The corresponding discrete RAP within the PEs of that component can then be solved. Note that the objective function can be regarded as a "black box", calculated by iterative *RF* compositions followed by a weighted value function calculation, as noted in Section 3. This RAP can be solved by a scheme known as *Non-Serial Dynamic Programming (NSDP)* [15]. As part of the solution we obtain the optimal importance for each level of resolution up to the total resources in the system.

Having performed this NSDP on *each* component we now consider the problem of optimizing over *all* components. The good news here is that the problem is a *separable* RAP. Separability here means that each summand is a function of a single decision variable, and such resource allocation problems are inherently easier to solve. In fact, if the component importance functions happen to be *concave* the problem can be solved by fast algorithms due to Fox or Galil and Megiddo. If the component importance functions, on the other hand, are not concave, the problem may still be solved by dynamic programming. See [15] for details on all of these algorithms. It turns out that concavity is not an uncommon condition for our component importance functions. So *macroQ* tests each component for concavity and employs the fastest combinations of these algorithms depending on the results.

At the end of this step we have computed the optimal *mips* allocations for each PE. But this can be regarded as just the inner loop of a three step nested process. In the central loop we evaluate all rank-legal templates. In the outer loop we evaluate successively finer resolution granularities.

The evaluation of all rank-legal templates is obviously exponential [13] in nature, though most jobs, in fact, only have a single template. The rank-legality constraints adds another exponential term, but these calculations can be streamlined, depending on the *macroQ* deadline.

The resolution granularity loop is simple in nature: *macroQ* starts with a coarse resolution to obtain a quick solution. Then it uses the time already spent to estimate the finest resolution it believes it can solve in the time remaining, subject to a reasonable minimum *mips* value. It outputs the best importance found, typically the finer resolution.

4.2 *macroW*

The *macro where* model, *macroW*, inputs from *macroQ* the set of resource allocation goals for PEs in chosen templates of admitted jobs, as well as the estimates of stream traffic between pairs of those PEs. The goal of *macroW* is to find a balanced allocation of these PEs to candidate nodes. Recall that these candidate nodes are a subset of the resource matched nodes, and that these choices will be respected by the *micro* model. These candidate nodes need to respect a large number of constraints, including several types of security and licensing constraints, memory, colocation and exlocation, limits on the maximum PEs per

node, maximum degrees of parallelism for each PE, fixed PEs, and incremental constraints.

To balance between the processing node and the bandwidth usage, *macroW* minimizes a weighted average of six separate metrics: These consist of the average and maximum estimated utilizations of the processing nodes, the average and maximum projected bandwidth of any network link, and the average and maximum projected utilization of any processing node's network interface.

macroW uses a two-pronged approach. First, the problem is modeled as a mixed-integer optimization program [16], and solved using a state-of-the-art commercial software CPLEX [17]. But the structure of the problem lends itself to a local search heuristic. So we have also developed a submodule of *macroW*, known as *miniW*, to do local search on the space of PE candidate node assignments. This serves as a back-up to the *macroW* solution, and as a post-processing heuristic to the "exact" solution provided by CPLEX.

In fact this heuristic has several advantages:

- Fault-tolerance: In case the CPLEX-based solution fails, the heuristic provides a backup solution.
- Robustness: For large problem instances, integer programming may be slow and not converge by the *macroW* deadline. Thus, an alternative that always produces a (possibly sub-optimal) solution quickly is crucial.
- Accuracy: Traffic components of the linear programming (LP) formulation are inherently quadratic in nature, and this results in weak LP relaxations being used by our CPLEX-based *macroW*. For large problem instances some of these non-linearities are ignored for smaller streams. A good solution to a more accurate model may be better than an exact solution to a less accurate one.

We describe the phases of *miniW* briefly.

First, a *preprocessing* phase shrinks the problem size. In particular, PEs with fixed candidate node assignments are removed, and appropriate bookkeeping is performed to reduce the remaining processing power of the relevant nodes. Likewise, streams whose PEs are fixed are removed, and the bandwidth on the relevant network links are reduced. Processing nodes which are down or fully utilized are removed from the problem as well. The reduced problem is often of much smaller size than the original, yielding significant time savings.

Next, the *initialization* phase provides a first feasible solution. There are several algorithms implemented here. In one example, streams are sorted based on traffic, and processing nodes sorted in terms of available load. Then, the PEs in these streams are mapped to the nodes, while ensuring feasibility. Another example is a round-robin approach: First, PEs from previous epochs are assigned to their previous candidate nodes. (This avoids incremental movement constraints as much as possible.) The remaining PEs are assigned to candidate nodes in round-robin fashion, again ensuring feasibility. A round-robin approach attempts to ensures that no processing node is overly loaded in terms of number of allocated PEs. All these solutions are compared with the solution obtained via CPLEX, and the best solution is used as a starting point for the next phase.

In the *local improvement* phase, *miniW* attempts to iteratively improve the solution by a variety of techniques. It may move a single PE from one candidate node to another, provided that move is feasible and the objective function decreases. (In the neighborhood search literature this is traditionally called a *1-opt* move.) The algorithm may try swapping the candidate nodes of two PEs. (This is a 2-opt move.) It may assign two PEs connected by a stream to the same candidate node. (This is also a 2-opt move.) This reduces traffic, but increases node utilization. Finally, it may swap all the PEs on a pair of candidate nodes. Each of these techniques can be helped by judicious orderings of the PEs, streams and processing nodes. The idea is to calculate how important each is to the overall solution, and sort by those metrics. For instance, PEs are ordered by decreasing *mips* requirements, decreasing traffic requirements, or exclusivity. Processing nodes are ordered by decreasing load. Streams are ordered by decreasing traffic requirements, or by decreasing allocation goals of the corresponding PEs.

Finally, there may be a *perturbation* phase. *miniW* is designed to run until it reaches its deadline or cannot improve the solution. So if the local improvement phase reaches a locally optimal solution, *miniW* will perturb that solution, insisting on feasibility but ignoring the fact that the solution does not improve. The same techniques as the local improvement phase are employed, with the hope of escaping the local minimum. The process then continues until the *macroW* deadline is reached.

4.3 *microQ*

The role of *microQ*, the *micro quantity* model, is to adjust the PE processing allocation goals from *macroQ* based on the PE candidate nodes determined in *macroW*. Recall that *macroQ* knows only the total resources available in *System S*, not information on the individual processing nodes. Only *macroW* considers the processing node information. So *microQ* effectively corrects problems that may arise from the decoupling of the *macro* model into two sequential problems.

The PEs are grouped into (weak) components, as per *macroQ*. The desired resource allocation for a particular PE depends on the overall allocation of *mips* to the component that contains it. This connection is described via *pacing constraints* that specify, for each level of allocation of *mips* to the component, the proportion of these *mips* that should be allocated to each PE. For each component, we use *macroQ* to determine a piecewise-linear, concave function which approximately maps the resources allocated to the component to *importance*. The goal is to allocate resources to components to maximize total importance, satisfying the component-PE pacing constraints.

Since this problem is nonlinear, we do not solve it directly. Instead, we take an iterative approach, as follows: We estimate the resource allocation for each component. This determines a set of *linear* pacing constraints to enforce. Now, the problem can be solved as an LP that is actually a network flow problem [16] with these additional pacing constraints. If any component in the solution falls into a linear segments other than the one assumed, we impose the "revised"

pacing constraints, and re-solve. The final solution is obtained when the process converges, or when the time allotted to *microQ* runs out.

4.4 *microW*

The goal of *microW*, the *micro where* model, is to make actual fractional assignments of PEs to processing nodes. The idea is to match as closely as possible the overall processing power goals computed for each PE by the *microQ* model, while meeting various constraints on incremental movement and node changes, fixed PEs, legal fractional allocations and so on. One constraint, for example, limits the cumulative amount change in fractional assignment values from the previous epoch. Another does so on a per PE basis, and a third on the number of processing nodes that can be modified during the current epoch.

The *microW* problem is solved via suitably modified techniques borrowed from the network flow literature [16]. We build and maintain a directed graph with three types of nodes:

- On the left side the nodes are the under-allocated PEs, ordered from most under-allocated to least under-allocated.
- In the middle the nodes are the processing nodes themselves.
- On the right side the nodes are the over-allocated PEs, ordered from least over-allocated to most over-allocated.

Directed arcs in this digraph exist if it is possible to push flow for a particular PE from one node of the digraph to another. The *microW* algorithm can be described as a doubly nested loop. The outer loop is performed on the under-allocated PEs, from most under-allocated to least under-allocated. The inner loop is performed on the over-allocated PEs, from most over-allocated to least over-allocated. A shortest path is chosen between the under-allocated PE and the over-allocated PE, and the maximal feasible flow is pushed along this path. After each successful flow push we perform the relevant bookkeeping and maintenance, adjusting the constraints, recomputing the under- and over-allocated PEs and incrementally reconstructing the directed graph. If there are no under- or over-allocated PEs *microW* ends with a perfect solution. The *microW* scheme also ends if flow push failures occur through an iteration of the entire doubly nested loops or if *microW* reaches its deadline.

5 Experimental Evaluation

5.1 Methodology

We evaluate *SODA* in the context of two qualitatively different *System S* applications: LSD [1], and DAC [6]. The LSD application is a large application intended to process high incoming data rates. It is composed of 104 jobs and 737 PEs. The LSD PEs are generally lightweight, but because the final job graph is large and highly connected, producing a flow-balanced schedule is difficult. The

DAC application is smaller but provides scheduling challenges because its PEs have a wide range of processing requirements. It consists of six jobs and 51 PEs. For the experiments, the jobs corresponding to each application are submitted to the *System S* cluster, where they are run for ten minutes to collect relevant data. For both these applications, SODA takes less than a minute to compute a solution.

We compare the *SODA* PE placement decisions to three other approaches:

- **Random (RAND)**: PEs are assigned to nodes uniformly at random. In expectation, each processing node hosts the same number of PEs, but in fact, the number of PEs hosted by a node may vary quite a bit.
- **Round-robin (RR)**: PEs are processed sequentially and each PE is assigned to a node with the minimum PEs assigned so far. This is a very naive load balancing of PEs across the nodes.
- **Expert (EXP)**: The application developers for LSD and DAC decide on the number of nodes and an allocation of PEs to nodes based on both their knowledge of the application as well as several trial-and-error runs where all PEs are resource matched to specific nodes. These placements are often tested in underloaded test environments, and cannot be expected to scale to overloaded environments. But they offer a reasonable measure of performance, one that must at least be matched, even in overloaded settings, by the scheduler.

These three schemes only perform PE placement–they do not address admission control, template choice or PE fractional allocations.

We evaluate each scheduler using the following metrics:

- *Ingest rate*: This is a measure of how much data (in Mbps) could be processed by the system. It is intended as a measure of the system's "effective capacity", and should be correlated to importance.
- *Importance*: The importance of a job is measured at the sink PEs as a quantity-based metric that depends on the data rates at the sink PEs. In our experiments, the streams into the sinks have unit weights and identity value functions, while all other streams have zero weights and value functions. As a result, the importance of a job is measured by the data rate flowing into its sink PEs.
- *Stream affinity*: One way to measure the quality of the placement is in terms of the traffic load on the system. We compute the amount of traffic that is sent between PEs on the same node divided by the total traffic. The higher this quantity, the better, since PEs which share a stream should be put on the same node (or nearby) to minimize network utilization.

These metrics are computed from the raw system metrics such as CPU usage per PE and traffic consumed and produced by each PE.

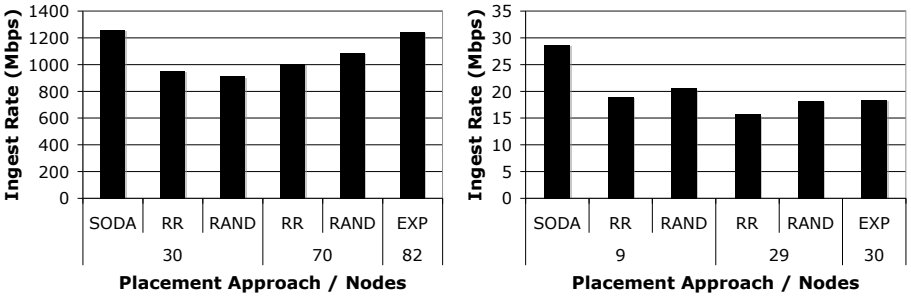
In the experiments below, we test the scheduler performance under different resource conditions ranging from under-provisioned to over-provisioned, which is achieved by varying the number of nodes made available to the scheduler.

This allows us to see how the performance will change as the *raw* system capacity changes, and also which scheduler is better at achieving higher system utilizations and better *effective* system capacity. We perform three runs for each combination of scheduler and node pool size, and analyze the average across these runs.

5.2 Results

The carefully constructed EXP placements use 82 nodes for LSD and 30 nodes for DAC. *SODA* uses far fewer nodes yet achieves a higher quality placement than EXP. In particular, *SODA* performs favorably with as few as 30 nodes for LSD and 9 nodes for DAC, 36% and 30% of the number of nodes used in the expert placement, respectively. To compare with these, we also present the results for RAND and RR for two scenarios: 30 and 70 nodes for LSD, and 9 and 29 nodes for DAC. These allow us to compare their performance with *SODA*'s placement at one end of the spectrum (less nodes), and with EXP at the other end (more nodes).

Figure 5 compares the ingestion rates of *SODA*, EXP, RR, and RAND. From the figure, we see that *SODA* is able to ingest as much traffic as EXP with far fewer nodes (30) for LSD. For DAC, *SODA* outperforms EXP by over 50% with just 9 nodes. This is largely because EXP seeks to ensure that all PEs receive sufficient MIPS. As a result, the PEs are spread across many more nodes than they need to be, while SODA recognizes that nine nodes is enough and so saves on traffic. For a given node pool size, the *SODA*-computed placement also consistently ingests more traffic than RAND or RR. The performance of both RR and RAND is, not surprisingly, poorer than EXP. For instance, with 70 nodes for LSD, RR is able to ingest 25% less traffic than EXP, and with 29 nodes for DAC, RR is able to ingest 15% less traffic than EXP.



(a) LSD (b) DAC

Fig. 5. Ingest rate: LSD and DAC

One of the metrics that SODA tries to maximize is the importance. Figure 6 presents the importance of DAC, as optimized by SODA in *macroQ*; recall from Section 3 that in our case this corresponds to the net traffic flowing into the sink PEs. Here, we see that *SODA* matches the performance of EXP in spite of using a third of the nodes. On the other hand, RR and RAND perform more than 10% worse than EXP, even when using 29 nodes. In particular, RR achieves only 84% of the traffic rates at the sinks attained by EXP and SODA.

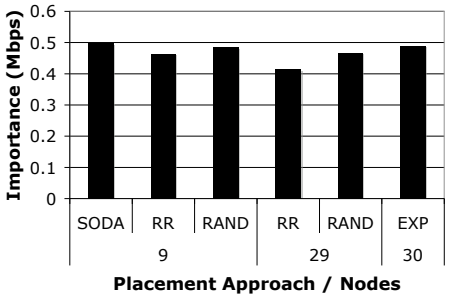
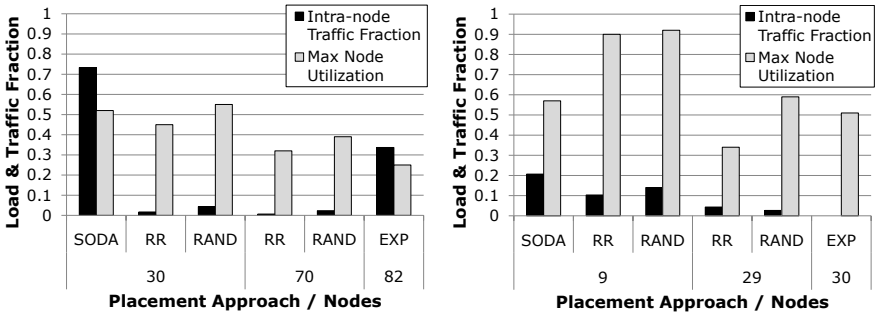


Fig. 6. Importance: DAC

Another goal of SODA is to ensure the network and nodes are not overloaded. The effect of the schedulers in terms of two system metrics is shown in Figure 7, which plots the stream affinity and maximum load for LSD and DAC. Stream affinity is the fraction of traffic that is sent on streams that have both source and destination PEs on the same node; higher is better. The load is indicated by the maximum CPU utilization across the nodes in the cluster; lower is better. From the figure, we see that *SODA* increases the intra-node traffic fraction without significantly increasing the maximum node load.

Considering traffic, with 30 nodes, the *SODA* placement for LSD sends less than 30% of the traffic over the network (over 70% on the same node); compared to 66% for EXP with 82 nodes. In addition to helping reduce network congestion, this also contributes to the stronger throughput results obtained by *SODA*, since the overhead of sending data to a PE on the same node is lower. Naturally, RAND and RR fare poorly on this metric since they do not use stream information in their placement algorithm, and in fact are susceptible to exceeding the network capacity. In particular, for the case of LSD with 30 nodes, *SODA* is able to achieve much higher stream affinity (70%) than RAND and RR (less than 10%) with the comparable maximum loads (around 50%). For DAC, with 9 nodes, *SODA* places 20% more of the traffic on the same node, even though the maximum load is comparable to EXP, resulting in a significantly larger ingest rate.

Now considering load, we see that with DAC, RR and RAND do rather poorly with 9 nodes, by causing some nodes to be highly loaded. This is because the DAC PEs have dramatically different CPU requirements. In contrast, *SODA*



(a) LSD (b) DAC

Fig. 7. Placement tradeoffs: LSD and DAC

balances the PEs across the nodes, thereby resulting in a much lower load, only slightly higher than the EXP with larger number of nodes. For LSD, the PEs are much more uniform, so RR and RAND perform satisfactorily in this metric. SODA results in a slightly higher load, but due to the higher intra-node traffic and more balanced placement, it is nevertheless able to achieve a higher ingest rate. Further, this maximum utilization of 52% is not in the problematic range.

In all our experiments, we observe that *SODA* requires significantly fewer nodes, and utilizes much less network capacity to perform as well, if not better than, a carefully constructed expert placement. Furthermore, we see that naive approaches like RAND and RR perform worse than *SODA* in general. This illustrates the strength of the scheduler, and its ability to schedule effectively in overloaded systems.

6 Related Work

Stream processing systems have been an active area of research in recent years. Example systems include Borealis [7], TelegraphCQ [8], STREAM [9], Aurora and Medusa [10]. These systems process voluminous quantities of incoming stream data, typically performing relational operations such as joins and selections on them. In contrast, *System S* is much more general, allowing arbitrarily complex operators, including relational ones.

Most of these stream processing systems are designed to be run on more than one node, and thus there has also been work on scheduling and load-balancing the operators. While these scheduling approaches have some of the flavor of the work we present here, none targets our problem exactly. We describe some of these related approaches here.

The FIT algorithm [12] is a load-shedding algorithm which intelligently drops load. Determining where best to drop load can be quite a complex problem, since

dropping at a particular operator has an effect on the downstream operators, sometimes an unintended one. In some cases, shedding load on a particular operator increases the resources for other operators on that node, and so could *increase* load at nodes downstream. FIT cleverly addresses this problem in a distributed way, but without a global notion of importance. The *SODA* scheduler provides this same functionality as part of its resource allocation and scheduling, and does so in a way that takes into account the processing graph for a job and the total system objectives.

Xing et al. [18, 19] addresses the problem of variance in stream rates. Both papers describe a way to distribute the load so that changes in input rate have a smaller chance of overloading the system. However, they do not address the case when the system is overloaded, and make no decisions about job admission.

Pietzuch et al. [20] provides a scheduling algorithm for a wide-area network that places operators so as to minimize network latency. In the local area network that we address, bandwidth, not network latency, is the main concern. In addition, their work does not address the problem of job admission. Lakshmanan et al. [21] also addresses scheduling to minimize latency.

The STREAM project [22] has goals somewhat similar to those presented in this paper. Their system handles queries in an SQL-like language. When resources are tight, they revise queries by dropping packets and/or changing internal parameters.

Xia et al. [23] address the admission control problem in a hypothetical stream processing system. Their model assumes a linear processing graph. In other words, the input stream is processed, successively, by a series of operators. Thus, no operator takes input from more than one source stream.

7 Conclusions and Future Work

In this paper we have introduced *SODA*, a scheduler for very large-scale distributed stream processing applications. This scheduler is implemented and running as a component in the *System S* project. We have shown that *SODA* is practical, novel, and effective, scheduling as well as or better than expert placement but using well under half the nodes. While schedulers of other stream processing systems have some features of *SODA*, *SODA* is unique in that in addition to allocating processing to nodes, it also controls job admission and weights the resources given to the admitted jobs. This overview paper provides an introduction to the problem, high level descriptions of the solution, and an experimental analysis which demonstrates *SODA*'s performance.

One of the more novel features of *SODA* scheduler is that it can schedule itself as a separate PE. The value function for *SODA* would measure the effect of additional processing resources on solution quality. Giving more resources to *SODA* would make the solution quality better at the possible expense of giving other work in the system more resources. We plan to create a *SODA* PE which can be scheduled in the near future.

Note that the notion of *SODA* scheduling itself is very different from the notion in Section 4 that *SODA has* a scheduler. We plan to improve this *SODA* scheduler as well.

Though *System S* is oriented towards streaming applications, traditional work will invariably be performed as well. So we have created (but not yet integrated) a scheduler for the more traditional sorts of jobs that invariably are needed in any system.

For very large problem instances we expect to design a variant of *SODA* in which epochs are arranged in a two level temporal hierarchy. In this case, the *macro* model will run in a *macro epoch*, and the *micro* model will run in a *micro epoch*. There will be a number of micro epochs in each macro epoch, allowing the computationally expensive *macro* models more time for their optimization. (We have not yet seen problem instances in which this approach would be necessary.) For truly large problems we have a design, not yet fully coded, to partition the work in *SODA*, allowing for vast scaling, though potentially at some loss of accuracy.

System S was built for a traditional packet-based network. But there is actually great affinity between *System S* and circuit switching architectures: Communication between PEs is long-lived, on the order of multiple minutes or more. *Optical Circuit Switches (OCS)* provide all of the benefits of circuit switching and make the bandwidth of the system more flexible. We have developed (and are continuing to refine) an extension to *SODA* that allows it to make *link assignments* (defining the network topology) at the same time it performs its traditional role of making PE candidate assignments. A lab prototype has been built.

References

1. 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: International Workshop on Data Mining Standards, Services and Platforms. (2006)
2. 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)
3. 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**(4) (2008)
4. 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: ACM SIGMOD International Conference on Management of Data. (2006)
5. Jacques-Silva, G., Challenger, J., Degenaro, L., Giles, J., Wagle, R.: Towards autonomic fault recovery in System-S. In: International Conference on Autonomic Computing. (2007)
6. 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: International Conference on Very Large Data Bases. (2007)

7. 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: Conference on Innovative Data Systems Research. (2005)
8. 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: Conference on Innovative Data Systems Research. (2003)
9. 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)
10. Zdonik, S., Stonebraker, M., Cherniack, M., Cetintemel, U., Balazinska, M., Balakrishnan, H.: The Aurora and Medusa projects. IEEE Data Engineering Bulletin **26**(1) (2003)
11. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.L., Fleischer, L.: Scheduling optimizer for distributed applications: A reference paper. Technical Report 24453, IBM Research Report (2007)
12. Tatbul, N., Çetintemel, U., Zdonik, S.: Staying fit: Efficient load shedding techniques for distributed stream processing. In: International Conference on Very Large Data Bases. (2007) 159–170
13. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. McGraw Hill (1985)
14. Blazewicz, J., Ecker, K., Schmidt, G., Weglarz, J.: Scheduling in Computer and Manufacturing Systems. Springer-Verlag (1993)
15. Ibaraki, T., Katoh, N.: Resource Allocation Problems. MIT Press (1988)
16. Nemhauser, G.L., Wolsey, L.A.: Integer and Combinatorial Optimization. John Wiley and Sons, New York (1988)
17. ILOG: CPLEX. http://www.ilog.com/products/cplex
18. Xing, Y., Hwang, J.H., Çetintemel, U., Zdonik, S.: Providing resiliency to load variations in distributed stream processing. In: International Conference on Very Large Data Bases, VLDB Endowment (2006) 775–786
19. Xing, Y., Zdonik, S., Hwang, J.H.: Dynamic load distribution in the Borealis stream processor. In: IEEE International Conference on Data Engineering, Washington, DC, USA, IEEE Computer Society (2005) 791–802
20. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: IEEE International Conference on Data Engineering, Washington, DC, USA, IEEE Computer Society (2006)
21. Lakshmanan, G., Strom, R.: Biologically-inspired distributed middleware management for stream processing systems. In: ACM/IFIP/Usenix International Middleware Conference. (2008)
22. Motwani, R., Widom, J., Arasu, A., Babcokc, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query processing, approximation, and resource management in a data stream management system. In: Conference on Innovative Data Systems Research. (2003)
23. Xia, C.H., Towsley, D., Zhang, C.: Distributed resource management and admission control of stream processing systems with max utility. In: ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems. (2007)