RC24911 (W0912-015) December 2, 2009
Computer Science

IBM Research Report

The *SODA* Optimizing Scheduler for *System S*

Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Lisa Fleischer
Dartmouth College

The *SODA* Optimizing Scheduler for *System S*

JOEL WOLF
IBM T.J. Watson Research Center
NIKHIL BANSAL
IBM T.J. Watson Research Center
KIRSTEN HILDRUM
IBM T.J. Watson Research Center
SUJAY PAREKH
IBM T.J. Watson Research Center
DEEPAK RAJAN
IBM T.J. Watson Research Center
ROHIT WAGLE
IBM T.J. Watson Research Center
KUN-LUNG WU
IBM T.J. Watson Research Center
LISA FLEISCHER
Dartmouth College

This paper describes *SODA*, a scheduler for *System S*. *System S* is a highly scalable distributed computer system designed to handle complex applications processing enormous quantities of streaming data. Unlike traditional batch applications, streaming applications are open-ended. The system cannot typically delay the processing of the data. The scheduler must be able admit or reject jobs. It must be able to assign and reassign resource allocations dynamically in response to changes in resource availability, incoming data rates, the relative importance of the work, as well as job arrivals and departures, and so on. The design assumptions of *System S*, in particular, pose additional scheduling challenges. *SODA* must deal with a highly complex optimization problem involving numerous real-world constraints, 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. This paper is intended to be as complete a description of *SODA* as is reasonably practical. We describe the design and functionality of *SODA*, give overviews and extensive details of the four major mathematical components, as well as three key input data infrastructure components. We present experiments to show the performance of the scheduler.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Distributed Applications*; G.1.6 [**Numerical Analysis**]: Optimization—*Constrained Optimization*

General Terms: Distributed processing, Stream processing, Optimization, Scheduling

Additional Key Words and Phrases: Job Admission, Resource allocation

1. INTRODUCTION

We consider distributed computer systems designed to handle very large-scale data stream processing jobs. This area of research is quite new. Early examples include relational databases augmented with streaming operations [Abadi et al. 2005; Chandrasekaran et al. 2003; Arasu et al. 2003; Zdonik et al. 2003]. These systems process voluminous quantities of incoming stream data, performing relational op-

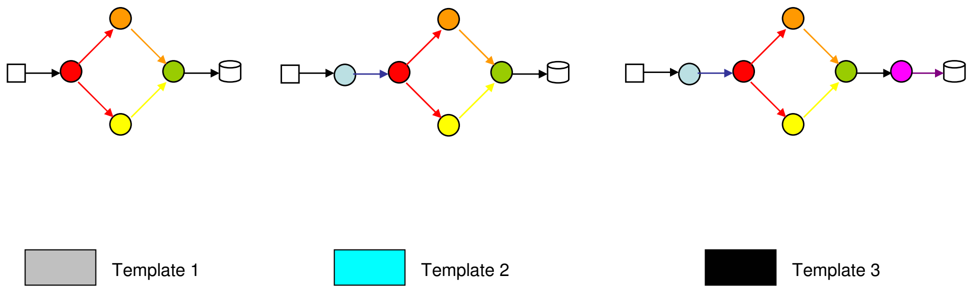


Fig. 1. Templates and Resources

erations such as database joins on them.

We believe that distributed stream processing systems are becoming quite important. The authors of this paper are involved in an ambitious project, started in 2003, known as *System S* [Amini et al. 2006; Douglis et al. 2004; Hildrum et al. 2008; Jain et al. 2006; Jacques-Silva et al. 2007; Wolf et al. 2008; Wolf et al. 2009]. *System S* is a highly scalable distributed computer system designed to handle complex jobs involving enormous quantities of streaming data. The system is intended to be vastly more general than a database environment. *System S* continues to evolve.

This paper describes the *SODA* scheduler for *System S*. (*SODA* stands for *Scheduling Optimizer for Distributed Applications*.)

The basic unit of computational work in *System S* is called a *processing element (PE)*. These PEs can be very general software. The PEs are connected via *streams* and grouped into *jobs*, which represent the basic unit of schedulable work in the system. Design goals reflect the ambitious nature of the project: when completed, we expect the system could consist of tens of thousands of *processing nodes* (*PN*s), to be able to concurrently support hundreds of thousands of *primal* (originating from outside the system) and *derived* (created within the system) streams, and to have a storage subsystem with a capacity of multiple petabytes. Even at these sizes, we expect the system to be swamped almost all of the time. Processors will be nearly fully utilized, since the offered load (in terms of jobs) will far exceed the prodigious processing capabilities of the system. The network will run at very high bandwidth rates. The storage subsystem will always be virtually full, because there will inevitably be more potentially useful data than can actually be stored. Such goals make the design of the system enormously challenging and substantially different from prior stream processing systems.

The composition of PEs and streams into data flow graphs seems natural and scalable. PEs consume and produce streams and are the *atomic units* to be composed, distributed and re-used. Streams consist of multiple *stream data objects (SDOs)*. In essence, *System S* is middleware which provides the PE and stream services. Technically, jobs consist of one or more data flow graphs, each of which represents an alternative way of performing a specific function. The alternative choices are called *templates*.

Figure 1 shows a job with three such templates. The PEs are shown as nodes in the data flow graphs. (The left-hand node in each case is a dummy node, which

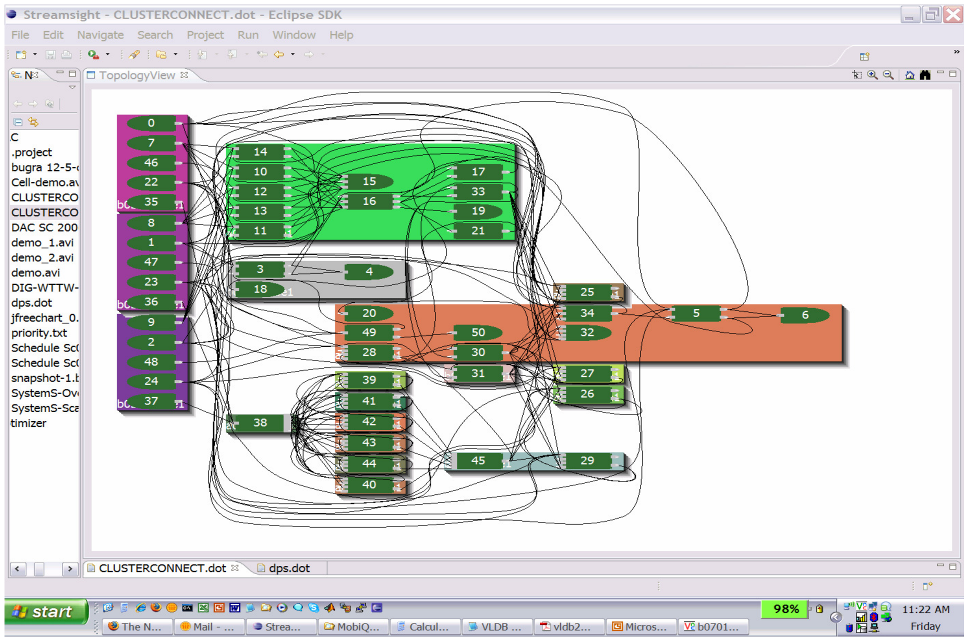

Streamsight - CLUSTERCONNECT.dot - Eclipse SDK
File Edit Navigate Search Project Run Window Help
TopologyView
CLUSTERCONNECT.dot
dps.dot
start
11:22 AM
Friday

Fig. 2. Streamsight: Data-flow graph

may be used to “glue” this job to another. The right-hand node in each template represents disk storage, which can be regarded as a second type of dummy node. Both the left- and right-hand side streams are required by *SODA* to “match” in all templates, as are the dummy nodes, so that the gluing may take place regardless of the template chosen.) The first template provides the basic functionality. The second template adds a preprocessing PE (perhaps for enhanced quality), and the third adds a postprocessing PE, presumably for the same reason.

Figure 2 shows a particular *System S* application known as *DAC* [Wu et al. 2007], using a specially built visualization tool known as *Streamsight*. (*DAC* stands for Disaster Assistance Claim monitoring.) This figure shows the PN assignments for the PEs of the job, as determined by *SODA*. Each colored rectangle represents a different PN. The PEs are placed in these PNs and the intra- and inter-PN streams are visible.

Like *System S* itself, *SODA* is highly ambitious. The environment makes producing high quality schedules very difficult. Below, we list some of the challenges for this scheduler.

—The offered load will frequently exceed system capacity. Thus *System S* components should run at nearly full capacity much of the time. This includes the PNs, the network, and storage. A lack of spare capacity means no room for error. So the scheduler needs to be quite accurate.

—These are stream-based jobs. We have essentially one shot at most primal streams, so the decision on which jobs to run is crucial to get right. This also means that the scheduler must react intelligently in real-time.

—The jobs involve multiple PEs which are interconnected in complex, changeable configurations via bursty streams, just as multiple jobs are themselves glued together. Flow imbalances can lead to buffer overflows (and loss of data), or to under-utilization of PEs. We must not lose much data and we must not leave processors idle frequently.

—We must be able to dynamically rebalance resources for jobs whose "importance" changes frequently and dramatically. (We shall shortly define *importance* in a rigorous but general manner. For now the reader can think of importance subjectively.) Discoveries, queries and the like can require major shifts in resource allocation which must be made quickly. Even primal streams come and go.

—There are many special purpose additional constraints which must be respected by the scheduler.

The word "scheduler" has many distinct connotations in the literature [Blazewicz et al. 1993; Pinedo 1995]. To be precise, the *SODA* scheduler we designed performs the following four major functions.

—*Job admission:* It chooses a subset of jobs to execute from a huge collection of jobs submitted. In the process it attempts to maximize the total importance of all the work in the system subject to a variety of constraints.

—*Template selection:* For those jobs that will be executed it chooses one alternative PE/stream template from among several given to it. Each alternative template will presumably provide the greatest importance within some range or ranges of allocated resources. (Figure 3 illustrates importance as a function of resource for the three job templates. At medium allocated resource levels, for example, the second template dominates.) *SODA* chooses the template based on the relative importance and resources required by the other, competing jobs.

—*Candidate Nodes:* Internally, *SODA* computes a set of of *candidate* PNs for each executing PE from a distributed system of heterogeneous PNs. These are the PNs on which the PE will be allowed to be run. It does so in a manner which load balances the nodes and network traffic, minimizes the inter-node traffic while respecting a host of constraints.

—*Fractional allocations:* Finally, *SODA* chooses flow-balanced fractional allocations of PEs to PNs while respecting the candidate node decisions and still more constraints.

In summary, the scheduling problem we are given involves the simultaneous objectives of maximizing importance, ensuring flow balancing amongst the PEs, load balancing the PNs, as well as minimizing and load balancing the traffic on the network. And the solution must respect an extensive array of constraints. These include a notion of job *rank* legality, minimum and maximum allocations to each PE, required jobs, resource matching, licensing, security, fairness, incremental change, cardinality, mutual PE colocation, mutual PE exlocation, PE isolation, and legal fractional allocation constraints. Moreover, we must solve these complex optimization algorithms in real-time while maintaining scalability to huge problem sizes. This is a huge juggling act.

Because of the difficulty of this problem we have resorted to a number of mathematical "tricks":

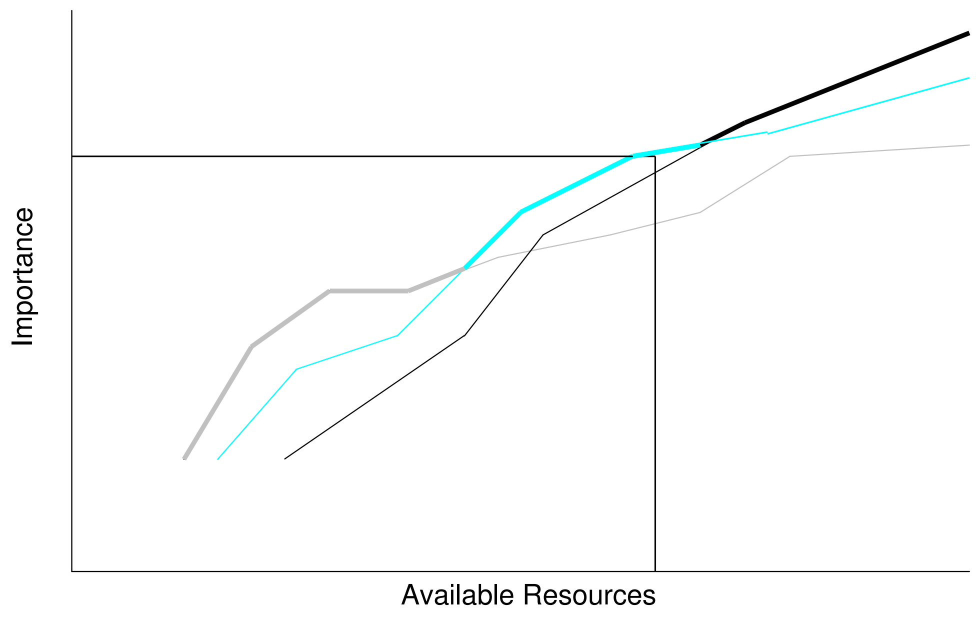


Fig. 3. Template Alternatives

—Hierarchical (multi-level) problem decomposition. The idea here is to use a two-level problem decomposition. The higher level component solves very hard mathematical optimization problems, but takes a relatively longer time to do so. This component is known as the *macro* model, and the time allowed for solving this problem is known as a *macro epoch.* The lower level component solves easier mathematical optimization problems using the output of the macro model as input. Its decisions are easier *because* it uses the higher level decisions as a guide. This lower level component is known as the *micro* model, and the time allowed for solving this problem is known as a *micro epoch.* There will be many micro model epochs in one macro model epoch, and each such micro model run will use the same macro model output. (To be precise, the input data for each micro epoch is computed as the output of the *previous* macro epoch.) Figure 4 shows a typical macro epoch and multiple micro epochs within it. At the top level *SODA* "thinks long and hard". At the lower level it "thinks fast".

—Sequentially decoupling the problems. In both the macro and micro models the original problem is decoupled into two sequentially solved problems. The first problem computes quantity related output and the second computes placement related output using the quantity output as input. Thus we have the *macroQ*/*microQ* (*Q* for *quantity*) models and the *macroW*/*microW* (*W* for *where*) models. Decoupling an optimization problem cannot improve the quality of the solution, but the advantages of greater tractability can often outweigh the possible loss of accuracy. We optimize slightly different objective functions in each of the models.

—Employing both heuristic and exact algorithms. We use heuristics for at least

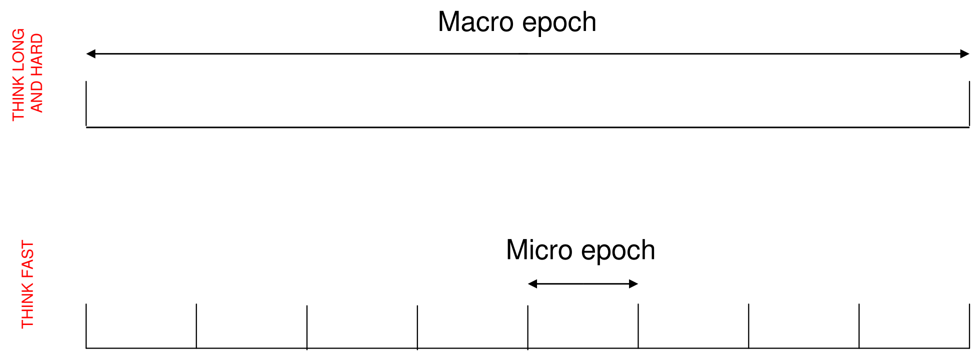


Fig. 4. *SODA* Epochs

three distinct purposes. First, we get a good starting solution for the exact scheme. Second, we get a solution which will be satisfactory even if the rest of the exact problem is too difficult to solve in the allotted time. Third, we may have to simplify the formulation of the exact problem in order to solve it in the allotted time. So there are trade-offs between exactness and the correctness of the formulation. And fourth, we can use heuristics to "true up" exact solutions to these simplified exact problems.

—Multi-granularity problem decompositions. We can solve a particular optimization problem at a coarse resolution. If there is sufficient time, we may solve it at a finer level of resolution. The latter will be more accurate than the former, but by performing this trick we are essentially trying to optimize the use of time in the scheduler itself. We also get at least *some* solution relatively quickly.

—Partitioning the problem into smaller problems. If the optimization problem is truly huge we create a number of smaller problems, solving each of these in parallel. Then we "glue" the results together. This is again a trade-off of computation time and accuracy.

We will give more details on each of these tricks in subsequent sections. The underlying mathematics of *SODA* is the key contribution of this paper.

The remainder of this paper is organized as follows. In Section 2 we give an overview of *System S*, stressing the other major components. Section 3 contains a glossary of new terms used by *SODA*, and by *macroQ* in particular. A description of each of the four major mathematical components is given in Section 4. As noted, these components are named *macroQ*, *macroW*, *microQ* and *microW*. Section 5 contains more mathematical details on each of these components. In Section 6 we briefly describe three components that provide key input data to SODA. These inputs are the so-called *resource functions*, job *ranks* and stream *weights*. The components themselves are known as the *Resource Function Learner (RFL)*, the *Rank Manager (RM)* and the *Weight Manager (WM)*. Section 7 describes *SODA* job admission and placement experiments. Section 8 outlines related work. Finally, Section 9 lists conclusions.

2. OVERVIEW OF *SYSTEM S*

System S is a large-scale distributed stream processing middleware. It is designed for supporting complex analytics on large volumes of streaming data, both structured and unstructured. *System S* has two main components: *SPADE* and the *System S* runtime. *SPADE* is a rapid application development front-end for *System S* [Gedik et al. 2008]. It consists of a language, a compiler, and auxiliary support for building distributed stream processing applications. The *SPADE* language provides a stream-centric, operator-level programming model, and composes these operators into logical data-flow graphs.

The operator logic can optionally be implemented in a lower-level language like C++. Generally, a *SPADE* operator implementing a simple logic, like filtering, would be too "small" to be efficiently deployed to a compute node at runtime. The *SPADE* compiler can fuse multiple operators into a single PE. In the process of code generation, *SPADE* creates these PEs by replacing all intra-PE streams with more efficient function call invocations of "downstream" operators by their "upstream" operators. Only inter-PE streams remain as actual streams after this fusion process. Thus, the logical (operator-based) data-flow graphs are coalesced into physical (PE-based) graphs that are more appropriate for deployment.

At runtime, the processing of stream applications is organized in terms of one or more *jobs* that consist of PEs organized into data-flow graphs. Physically, a PE is a process, and may contain of one or more threads of execution. The PEs of an application are distributed across the PNs. Each PN can run multiple PEs and divides its CPU resource between them according to fractions dictated by the *SODA* scheduler. The PE can be a generic program that uses the streaming API or it may be composed from several fused operators, as above. In the latter case, its behavior depends on that of the individual operators and the manner in which they are connected inside the PE. PEs consume and produce streams which consist of a series of strictly-typed tuples. A PE receives and sends data through *ports*, which represent attachment points for streams. A PE can read from multiple ports, write to multiple ports, and multiple streams may originate or end in a single port.

A functional prototype of *System S* exists on a Linux cluster consisting of hundreds of PNs interconnected by a Gigabit switched Ethernet network.

3. GLOSSARY OF KEY NEW *SODA* TERMS

SODA (and *macroQ* in particular) employs a number of terms that have very specific meanings to the scheduler. We list these below, with explicit definitions. Understanding these concepts is critical to the discussions that follow. 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 essentially the basic building block 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. Jobs which produce little importance but have a better rank may get done *instead* of jobs which have more importance but have a worse rank.

—*Value function:* 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 "finished products" of *System S* work, and one would thus naturally want to measure goodness there.

—*Weight:* 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, since we will see that the weight may as well be 0 unless the stream also has a non-zero value function.

—*Importance:* 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 – in other words, 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 5. There are 8 jobs. For the last job the figure displays two alternative templates. Several jobs are connected to others. Initially there are positive weights at all the terminal, "starred" streams. All of the PEs will be given resources (assuming their job is admitted). But suppose that the second weight for job 7 is changed to 0. It follows that the 2 PEs immediately upstream of that weight will not 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 have before the change. Weights are thus an easy "knob" to turn on and off portions of a job and, more generally, a way to adjust relative importance.

Stream weights are supplied by a *System S* infrastructure component known as the *Weight Manager (WM)*. We will give a brief overview of the *WM* later on in this paper.

—*Resource function:* 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 *resource function (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 6. Assuming the

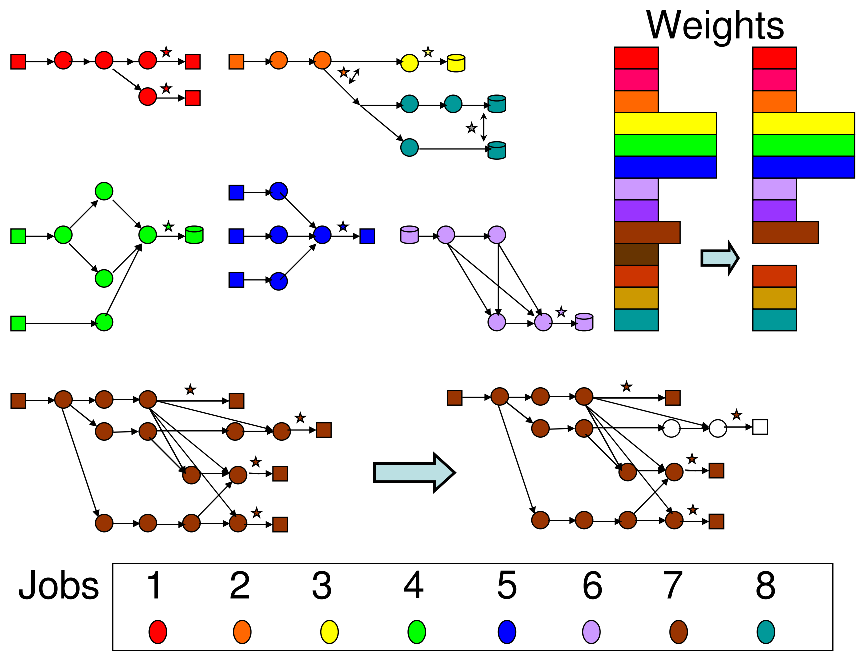


Fig. 5. *SODA* Jobs and Weights

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. (Some details are hinted at in the figure. Output ports filter the streams, and the output from PEs 1 and 2 are aggregated into the first input port, effectively decreasing the dimensionality of this *RF* by one.) The *RF* needs to be "learned" over time by a *SODA* infrastructure component known as the *Resource Function Learner (RFL)*. We will give a brief overview of the *RFL* in Section 6.1. 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 7. By topologically sorting [Cormen et al. 1985] a directed acyclic graph, we can apply ready list scheduling [Blazewicz et al. 1993; Coffman 1976] to compute the importance for stream s_5. In the figure three *RF*s are initially ready because they are fed by primal streams. So we obtain the rates at streams s_1 through s_3. Then additional *RF*s become ready (because their inputs have been computed), and we obtain the rate at streams s_4 and s_5 in succession. 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.)

—*Rank:* 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

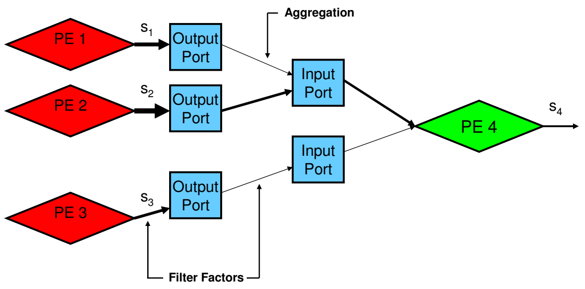
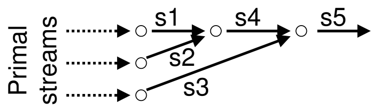
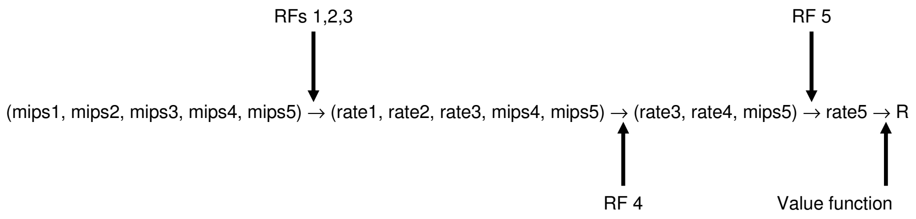


Fig. 6. Resource Function



(a) PE layout


RFs 1,2,3
RF 5
(mips1, mips2, mips3, mips4, mips5) → (rate1, rate2, rate3, mips4, mips5) → (rate3, rate4, mips5) → rate5 → R
RF 4
Value function

(b) Calculating importance

Fig. 7. Calculation of Importance

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. (There are two seemingly irreconcilable camps on the issue of whether rank should improve with value or the reverse. Our motivation in using the convention we chose is twofold: First, it is common to say that something is "priority one", meaning it is most important. Second, and certainly less arguably, one is the smallest positive integer, and thus we definitively will know that a job with rank one is most essential. On the other hand, it is certainly true that adopting this definition causes rank to be inversely related to the assigned rank number. We apologize to members of the other camp.)

Job ranks are supplied by a *System S* infrastructure component known as the *Rank Manager (RM)*. We will give a brief overview of the *RM* later on in this paper.

We now describe the possible subsets of jobs that can be admitted into *System S*. The key is that there will be a specific job rank for which the following holds: All jobs with lower ranks will be admitted, and all jobs with higher ranks will not be 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 is actually a very slight simplification, since one needs to account for inter-job dependencies.

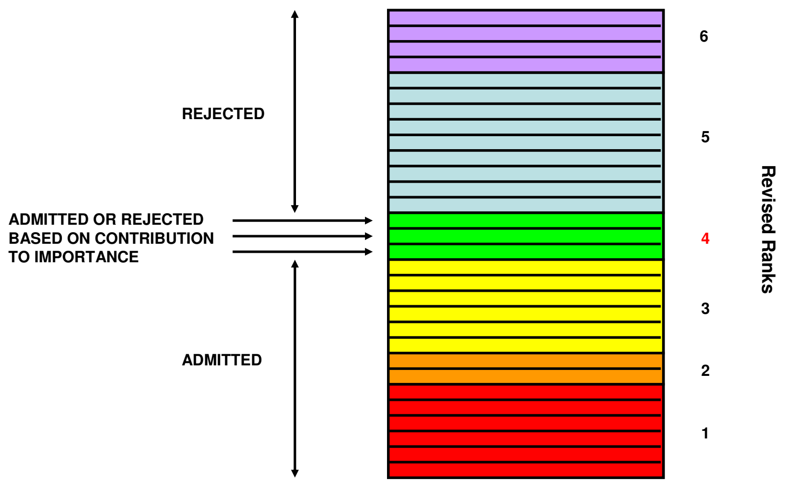


Fig. 8. Job Admission as a Function of Rank: Medium Load

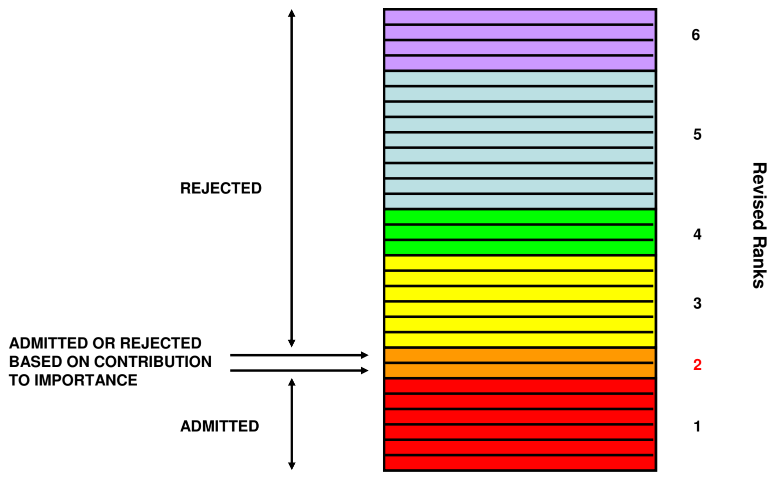


Fig. 9. Job Admission as a Function of Rank: Heavy Load

We will address this issue more formally in the next section.) Figures 8 and 9 show job admission in a heavy and medium load conditions (relative to available system capacity). As the load goes up the *waterline* (that is, the cardinality of the highest admitted job rank) goes down. In a light load condition perhaps all jobs would be admitted. Each of these alternatives is rank-legal.

4. MATHEMATICAL COMPONENT OVERVIEW

In this section we describe the four major mathematical components of *SODA*. We describe the solution approaches and motivate them from a practical standpoint, emphasizing how the solutions are dictated and/or guided by the *SODA* design philosophy. The two temporally hierarchical levels and their goals are:

—*The macro model*, which chooses the jobs that will be admitted, the templates for those jobs, and the candidate nodes to which the PEs in those jobs and templates can be assigned. The choices made in the macro model are respected by the micro model during the micro epochs of the *next* macro epoch, and this makes the decisions of the micro model easier and more effective.

—*The micro model*, which 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 PN *unless* that PN has been chosen as a candidate node by the macro model. The micro model handles dynamic variability in the relative importance of work (via revised weights), and changes in the state of the system (via PNs and PEs that go up or down), without having to consider the difficult constraints handled in the macro model.

This decomposition is not perfect. Periodically there could be solutions from the macro model which are inconsistent with the constraints of the micro model. A "micro to macro" feedback loop would seem to be useful, but we have not seen examples where it is needed in practice.

Now we describe the individual decoupled quantity and where components for both the macro and micro models:

—*macroQ*, the *macro quantity* model, maximizes projected importance by deciding which jobs to do, which templates to choose, and by computing flow balanced PE processing allocation goals, in *mips*, subject to job rank legality, required jobs, minimum and maximum *mips* constraints.

—*macroW*, the *macro where* model, minimizes projected network traffic and makes the task of load balancing the PNs easier by allocating uniformly more candidate nodes than PEs goals dictate, all subject to resource matching; security; licensing; fairness; incremental movement; PE colocation, exlocation and isolation; minimum and maximum numbers of candidate nodes per PE; and maximum numbers of PEs per PN. It actually optimizes a weighted average of four separate metrics, two of which are averages of the utilization of the PNs and the traffic in the network links. The other two are maximum values of these same metrics. The overallocation allows more flexibility in handling micro epoch dynamics.

—*microQ*, the *micro quantity* model, revises the maximimum projected importance, computing more accurate *mips* allocation goals for the PEs than those of *macroQ* by explicitly taking the candidate nodes into account. (Recall that *macroQ* does not know this information.) It also deals with revisions due to changes in PN states, PE states and the like. Thus, to some extent *microQ* acts as a "where to quantity" feedback loop.

—*microW*, the *micro where* model, attempts to implement all these optimization decisions: Its output is a set of fractional assignments of the PEs to the PNs such that for each PE the sum of the allocated *mips* across all candidate nodes is as close to the allocation goal from *microQ* as possible. There are constraints on incremental movement, acceptable fractional allocations, fixed PE to PN assignments and PN utilization.

5. MATHEMATICAL COMPONENT DETAILS

5.1 *macroQ*

The *macro quantity* model, *macroQ*, finds a set of jobs to admit during the next macro epoch. For each job it chooses a template from among the options given to it. Each template represents an alternative plan for performing the job. The jobs have ranks, and the jobs that are chosen by *macroQ* must respect a rank-legality constraint. Required jobs must be admitted. (Without loss of generality we can assume required jobs have rank 1.) 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 templates. In the process of solving the problem *macroQ* computes the optimal importance, the list of job and template choices, and finally the set of processing power goals (measured in *mips*) for each of the PEs within the chosen list. We formalize this below.

```
 1: Set $OPT = 0$
 2: Set $OK$ = false
 3: while $OK$=false do
 4:   Pick resolution granularity $\bar{G}$
 5:   for $r = R$ to 1 by -1 do
 6:     Create list of $L_r$ rank-legal job/templates with waterline $r$
 7:     for $l = 1$ to $l = L_r$ do
 8:       Compute $C_{\mathcal{J},T,l}$ weak components
 9:       for $c = 0$ to $c = C_{\mathcal{J},T,l} - 1$ do
10:         NSDP scheme to solve component $c$ RAP with granularity $\bar{G}$
11:       end for
12:       Compute number of components with concave importance functions
13:       if all are concave then
14:         Galil-Megiddo scheme to solve inter-component RAP with granularity $\bar{G}$
15:       else if none are concave then
16:         DP scheme to solve inter-component RAP with granularity $\bar{G}$
17:       else
18:         Fox/DP scheme to solve inter-component RAP with granularity $\bar{G}$
19:       end if
20:       if $\mathcal{I} > \mathcal{OPT}$ then
21:         OPT=$\mathcal{I}$
22:       end if
23:     end for
24:   end for
25:   Evaluate $OK$
26: end while
27: Output $OPT$
```

Fig. 10. *macroQ* Pseudocode

The problem formulation and the algorithm in *macroQ* are fairly elaborate. For the reader's convenience, Table I provides a summary of notation used, in order of appearance. And Figure 10 provides an outline of the *macroQ* pseudo-code. Note that there are basically three nested loops.

—The outer loop, from line 3 to line 26, considers different levels of resolution granularity for the resource allocation problems that will be solved. A coarse level

Variable	Definition
J	Number of jobs offered
Π_j	Original rank of job j
N_j	Number of templates for job j
$\mathcal{J}$	Job list
T	Template list
$\mathcal{T}_{\mathcal{J}}$	Set of all template lists for job list $\mathcal{J}$
$\mathcal{L}$	Job/template list
$D_{\mathcal{J},T}$	Directed acyclic graph associated with job/template pair $(\mathcal{J}, T)$
$\mathcal{P}_{\mathcal{J},T}$	Set of nodes (PEs) in $D_{\mathcal{J},T}$
$d_{\mathcal{J},T}$	Asymmetric distance function
$\mathcal{D}_{\mathcal{J},T,p}$	Set of PEs which depend on PE p
$\tilde{\mathcal{D}}_{\mathcal{J},T,j}$	Set of jobs which depend on job j
$\Pi_{\mathcal{J},T,j}$	Revised rank for job j
$\tilde{\mathcal{L}}$	Rank-legal job/template list
g_p	*mips* allotted to PE p
$\mathcal{V}_s$	Composite value function for stream s
w_s	Weight for stream s
I_s	Importance function for stream s
H	Total *mips* in system
m_p	Minimum *mips* for PE p if admitted
M_p	Maximum *mips* for PE p if admitted
$\tilde{\mathcal{J}}$	Jobs that must be admitted
$\bar{G}$	Number of resource units in discrete RAP
$\bar{m}_p$	Minimum resource units for PE p if admitted
$\bar{M}_p$	Maximum resource units for PE p if admitted
$C_{\mathcal{J},T,l}$	Number of weak components for lth job/template list $(\mathcal{J}, T)$
$\mathcal{P}_c$	Set of PEs in weak component c
$\mathcal{I}_c$	Importance function for weak component c
$\breve{m}_c$	Minimum resource units for weak component c
$\breve{M}_c$	Maximum resource units for weak component c
R_{max}	Worst revised rank
L_r	Number of job/template alternatives examined of revised rank r

Table I. Key *macroQ* Notation

of granularity provides a quick solution, while a fine level provides an accurate solution. Because *SODA* is a real-time scheduler, *macroQ* must have a solution by the time the macro epoch completes. The quick, coarse solution serves this purpose.

—The middle loop, from line 5 to line 24, decrements the possible revised rank waterlines, considering fewer and fewer jobs as it goes.

—The inner loop, from line 7 to line 23, is a divide and conquer approach based on the number of so-called *weak components* of the relevant data flow graphs. The overall resource allocation problem to be solved can be handled by solving an elaborate problem on each weak component, and then combining the solutions via a simpler problem across *all* components. We will describe these in more detail later in this section.

Ultimately we output the best solution discovered, on line 27.

5.1.1 *Notation.* Let J denote the number of offered jobs, indexed by j. Each job j has a postive integer *rank* Π_j. As noted previously, we adopt the convention that lower numbers indicate higher ranks. So the best possible rank is 1. Each job j comes with a small number of possible job *templates.* This number may be 1. It *will* be 1 if the job has already been instantiated, because we assume that the choice of a template is fixed throughout the "lifetime" of a job. It is, however, the role of the *macroQ* model to make this choice for jobs that are newly admitted. Let N_j denote the number of templates for job j, indexed by t.

Any subset $\mathcal{J} \in 2^J$ will be called a *job list.* For each job list $\mathcal{J}$ a function $T : \mathcal{J} \rightarrow \{1, ..., N_j\}$ will be called a *template list.* Denote the set of all template lists for $\mathcal{J}$ by $\mathcal{T}_{\mathcal{J}}$. Finally, define the *job/template list* to be the set $\mathcal{L} = \{(\mathcal{J}, T) | \mathcal{J} \in 2^J, T \in \mathcal{T}_{\mathcal{J}}\}$. A major function of *macroQ* is to make a "legal and optimal" choice of a job/template list.

We will make the assumption, for ease of exposition, that no cycles exist in the directed flow graphs for a job and template choice. *SODA* can actually handle intra- and inter-job cycles, but the details are somewhat complex.

So each job/template list $(\mathcal{J}, T)$ gives rise to a directed acyclic graph $D_{\mathcal{J},T}$ whose nodes $\mathcal{P}_{\mathcal{J},T}$ are the PEs in the template and whose directed arcs are the streams. (This digraph is "glued" together from the templates of the various jobs in the list, and we omit the exact details. These PE nodes may come from multiple jobs.) Assigning length one to each of the directed arcs, there is an obvious notion of an asymmetric distance function $d_{\mathcal{J},T}$ between pairs of relevant PEs. Note that $d_{\mathcal{J},T}(p, q) < \infty$ means that PE p precedes PE q, or, equivalently, that q depends on p. Let $\mathcal{D}_{\mathcal{J},T,p}$ denote the set of PEs $q \in D_{\mathcal{J},T}$ for which q depends on p. This notion of dependence gives rise, in turn, to the notion of dependence between the relevant jobs: Given jobs $j, j' \in \mathcal{J}$, we will say that j' depends on j provided there exist PEs q and p, belonging to j' and j, respectively, for which $d_{\mathcal{J},T}(p, q) < \infty$. Let $\tilde{\mathcal{D}}_{\mathcal{J},T,j}$ denote the set of jobs $j' \in \mathcal{J}$ for which j' depends on j.

We now define a revised job rank notion based on a particular job/template list $(\mathcal{J}, T)$ by setting

$$\Pi_{\mathcal{J},T,j} = \begin{cases} \min_{j' \in \tilde{\mathcal{D}}_{\mathcal{J},T,j}} \Pi_{j'} & \text{if } j \in \mathcal{J} \\ \Pi_j & \text{otherwise.} \end{cases}$$

This is well-defined. Now we can define the notion of a *rank-legal* job/template list $(\mathcal{J}, T)$. For such a list we insist that $j \in \mathcal{J}$ and $j' \notin \mathcal{J}$ implies that $\Pi_{\mathcal{J},T,j} \leq \Pi_{\mathcal{J},T,j'}$. (This is equivalent to the statement that there is a value for which all jobs with lower revised ranks will be admitted and all jobs with higher revised ranks will not be admitted.) Let $\tilde{\mathcal{L}}$ denote the set of rank-legal job/template lists.

Define the decision variable g_p to be the resource allocation, in *mips*, given to PE p. As noted, any derived stream s associated with job/template list $(\mathcal{J}, T)$ has a *value function.* The stream, in turn, is created by a unique PE p associated with $(\mathcal{J}, T)$. The PE p gives rise to a set $\{q_1, ..., q_{k_p}\}$ of k_p PEs q_i for which $p \in \mathcal{D}_{\mathcal{J},T,q_i}$.

This set includes p itself. We have also introduced the notions of learned *RFs* which can be iteratively composed to create a function from the processing power tuple $(g_{q_1}, ..., g_{q_{k_p}})$ to the domain of the value function. And so the composition of these recursively unfolded functions with the value function yields a mapping $\mathcal{V}_s$ from the tuple $(g_{q_1}, ..., g_{q_{k_p}})$ to the non-negative real numbers for stream s. This function is called the *composite value function* for s. Multiplied by a weight w_s for stream s it becomes a *stream importance* function I_s mapping $(g_{q_1}, ..., g_{q_{k_p}})$ to the non-negative real numbers $[0, \infty)$. Finally, aggregating all the stream importance functions together for all streams which are created by a given PE p yields a *PE importance function* $\mathcal{I}_p$.

Let H denote the total amount of *System S* processing power, in *mips*. Let m_p denote the minimum amount of processing power which can be given to PE p if it is admitted, and M_p denote the maximum amount of processing power which can be given to PE p if it is admitted. Suppose that the set $\tilde{\mathcal{J}}$ represents the jobs that must be admitted.

5.1.2 *Mathematical Formulation.* We seek to maximize the overall importance, which is the sum of the PE importance functions across all possible rank-legal job/template lists. The objective is therefore to find

$$\max_{(\mathcal{J},T)\in\tilde{\mathcal{L}}} \sum_{p\in\mathcal{P}_{\mathcal{J},T}} \mathcal{I}_p(g_{q_1}, ..., g_{q_{k_p}})$$

subject to the following constraints:

$$\sum_{p\in\mathcal{P}_{\mathcal{J},T}} g_p \le G, \tag{1}$$

$$m_p \le g_p \le M_p \qquad \forall p \in \mathcal{P}_{\mathcal{J},T}, \tag{2}$$

$$\tilde{\mathcal{J}} \subseteq \mathcal{J} \tag{3}$$

Constraint 1 is the resource allocation constraint. It ensures that all of the resource is used if it is useful and possible to do so. Constraint 2 requires a PE p to be within some minimum and maximum range if it is admitted. Constraint 3 insists that required jobs are admitted.

5.1.3 *Solution Approach.* We discretize the above continuous resource allocation problem by dividing the total amount of resource H into $\bar{H}$ equal size atomic units of "resolution" $H/\bar{H}$ *mips* each. Assume that this value $\bar{G}$ is given. For each PE p let $\bar{m}_p = \lfloor m_p\bar{G}/G\rfloor$ and $\bar{M}_p = \lceil M_p\bar{G}/G\rceil$ represent the discrete analogues of the minimum and maximum *mips* constraint terms. Also assume a fixed rank-legal job/template list $(\mathcal{J}, T) \in \tilde{\mathcal{L}}$ containing all the required jobs $\mathcal{J}$. Partition the PEs and streams into $C_{\mathcal{J},T}$ weak components and fix one such component c. Let $\mathcal{P}_c$ denote the PEs in component c.

We consider, using the natural change in notation, the corresponding discrete resource allocation problem of maximizing $\sum_{p\in\mathcal{P}_c}\mathcal{I}_p(\bar{g}_{q_1}, ..., \bar{g}_{q_{k_p}})$ subject to the constraints $\sum_{p\in\mathcal{P}_c} \bar{g}_p \le \bar{G}$ and $\bar{m}_p \le \bar{g}_p \le \bar{M}_p$ for all $p \in \mathcal{P}_c$. This problem can be solved by a scheme known as *Non-Serial Dynamic Programming (NSDP)* [Ibaraki and Katoh 1988]. NSDP is a complex dynamic programming scheme designed specifically to handle difficult (non-separable) resource allocation problems. (See line 10 of Figure 10.) As part of the solution methodology we obtain the optimal values $\mathcal{I}_c(\bar{g}_c)$ for every $\bar{g}_c$ between 1 and $\bar{G}$, *as well as* the PE *mips* allocations that constitute this optimal solution. We can thus regard $\mathcal{I}_c$ as a *component importance function* of the resources $\bar{g}_c$ allotted to component c. Set $\breve{m}_c = \sum_{p \in \mathcal{P}_c} \bar{m}_p$ and $\breve{M}_c = \sum_{p \in \mathcal{P}_c} \bar{M}_p$.

Note that the objective function can be regarded as a "black box", calculated by iterative *RF* compositions followed by a weighted value function calculation. To make this as efficient as possible the *macroQ* code has itself been carefully optimized. Careful analyses are performed to determine which sub-graph calculations are strictly necessary and which are redundant. A cache of previous results is also employed. Also, *macroQ* code is aware of time and is given a deadline by *SODA*. So it occasionally takes "shortcuts", using a partially greedy scheme instead of a full NSDP algorithm. This fits the design philosophy: *SODA* is a *real-time* scheduler.

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* resource allocation problem: We wish to maximize $\sum_c \mathcal{I}_c(\bar{g}_c)$ such that $\sum_c \bar{g}_c \leq \bar{G}$ and $\breve{m}_c \leq \bar{g}_c \leq \breve{M}_c$ for all c. 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 one of three algorithms: These are the schemes by Fox, Galil and Megiddo, and Frederickson and Johnson, which can be regarded as fast, faster and (theoretically) fastest, respectively. If the component importance functions, on the other hand, are not concave, the problem may still be solved by dynamic programming (DP). See [Ibaraki and Katoh 1988] for details on all of these algorithms. Also see lines 14, 16 and 18 of Figure 10.

As one would expect from a "law of diminishing returns" argument, it is a common condition that our component importance functions is concave, or nearly so. So we test each component for concavity and adopt one of three approaches, depending on the results.

—If all component importance functions are concave we solve the resource allocation problem by the Galil and Megiddo algorithm. This algorithm is quite fast in practice and much easier to code than the Fredrickson and Johnson scheme.

—If all the component importance functions are not concave we solve the resource allocation problem by dynamic programming.

—In other cases we solve the concave portion of the problem by the Fox algorithm (because it provides the needed intermediate values) and then solve the remainder of the problem by dynamic programming.

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. Again, see Figure 10.

The evaluation of all rank-legal templates is obviously exponential [Cormen et al. 1985] in nature, though the problem is generally not large: *SODA* only evaluates alternative for *new* jobs. Once a template decision has been reached it lasts for the remaining epochs of the job. And most jobs, in fact, only have a single template.

The rank-legality constraints adds another exponential term, but this process can also be streamlined if time is an issue. The code loops through each revised rank value, working from the worst revised rank R_{max} to the best revised rank 1: For any given revised rank value it assumes all poorer revised rank jobs will not be admitted, all better revised rank jobs will be admitted, and has to decide which jobs of that revised rank will be admitted. For all but the poorest revised rank these jobs were admitted in the previous calculation. The code computes their importance divided by their resource allocations and orders the jobs accordingly. If a full exponential evaluation will not complete in time the code admits jobs of that revised rank based on this ordering. The case where there are a large number of jobs of the highest revised rank is obviously less satisfactory. And this case includes the case where all jobs have the same revised rank. The code performs a greedy scheme if pressed for time, but the results may be less than optimal. The philosophy is that an imperfect *macroQ* solution is better than no solution at all. In the pseudo-code we let L_r the be the number of job/template alternatives examined of revised rank r, whether linear or exponential.

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 remaining time, subject to a reasonable minimum *mips* value (100 *mips* is typically used). It reports the best importance found, and this is typically based on the finer resolution.

5.2 *macroW*

The *macro where* model, *macroW*, employs two types of input from *macroQ*. First, it uses the set of resource allocation goals for PEs in the chosen templates of admitted jobs. Second, it uses the estimates of stream traffic rates between relevant pairs of those PEs. (As noted in Section 3, value functions are typically rate-based. Even if they are not, *macroQ* automatically computes the rates between PEs in the optimal solution by the same type of black box calculations described in Figure 7 and Section 5.1.3.) The goal of *macroW* is to find a "balanced" assignment of candidate PNs for these PEs. Additionally, the *macroW* assignments must respect a large number of practical constraints. The constraint types include resource matching; security; licensing; fairness; incremental movement; PE colocation, exlocation and isolation; minimum and maximum numbers of candidate nodes per PE; and maximum numbers of PEs per PN. Recall that the candidate node choices made in *macroW* will be respected during multiple *micro* model epochs.

To balance between the PN and the bandwidth usage, the objective function in *macroW* minimizes a weighted average of four separate metrics: These consist of the average and maximum estimated utilizations of the PNs, and the average and maximum projected utilizations of the various network links. (We point out that we are using the word utilization somewhat loosely here. We may *theoretically* overload both the PNs and the links, yielding projected utilizations in *macroW* that are over 100%. The reason for this is that we wish to over-allocate each PE, at least in theory, to have the flexibility of handling fluctuations in *actual* requirements. The implicit underlying assumption is that while some PEs will wind up being *hot*, others will simultaneously be *cold*. In *macroW* the goal is to build for the uniformly hot case. But typically, a running PE will not use all the resources it is assigned.

(The dynamic reallocation of actual resources to the PEs is precisely the job of the *micro* model.)

We note that the *macroW* problem can be modeled as an integer linear program (ILP) [Nemhauser and Wolsey 1988], and solved using state-of-the-art commercial software such as CPLEX [ILOG]. Howerver, the problem is of potentially enormous size, and ILPs cannot be solved efficiently. (In fact, the *macroW* problem is inherently quadratic in nature, because of the pair of PEs associated with each stream. We are "forcing" a linear formulation.) We therefore solve *macroW* in three stages, the first two of which, taken together, represent a divide and conquer hierarchical approach. These stages solve the *TopLevelMILP* problem, multiple *BottomLevelILP* problems, and the *miniW* problem, respectively. It is this decomposition, not the problems themselves, that is the contribution in our *macroW* approach. For the reader's convenience, Table II provides a summary of notation used, except for the decision variables, in order of appearance. There are a large number of decision variables in the *macroW* formulation, so we separate these out in Table III, also in order of appearance. Figure 11 provides stage level pseudo-code for *macroW*.

—At the *top level* (line 1) of the hierarchy we are primarily concerned with making decisions for *clusters* of PNs. Assume as input a partition of the PNs into clusters, within each of which the network connectivity is sufficiently high to ignore link traffic entirely. (In *System S* these clusters will typically consist of PNs which are connected by a high speed backbone. They might also correspond to a single BladeCenter chassis [BladeCenters].) The top level optimization problem makes the decision of what (single) cluster will contain the candidate nodes for a given PE. The assumption of a single cluster per PE restricts the optimality of the overall solution, but it significantly reduces the size of the problem. The expectation is that the overall solution will not be too compromised. (As we shall see, this assumption is temporary in any case.) The PN (and hence the cluster) utilizations are balanced in the objective function, as are the network link utilizations. But the higher level problem is treated as a continuous relaxation of the "real" *macroW* problem. Some fractional rather than binary decision variables are allowed, speeding up the solution at the expense of fully modeling the exact original problem. A few constraints are also omitted in the top level problem, because they don't make sense in the context of fractional decision variables. The top level problem is specifically modeled as a *mixed-integer linear program (MILP)*[Nemhauser and Wolsey 1988], and solved via CPLEX.

—At the completion of *TopLevelMILP* we have partitioned the PEs into clusters. Moreover, we infer from the solution information about how many resources (such as licenses) are required by each cluster. This knowledge allows us to decompose the orginal problem into far smaller independent problems, one per cluster. We call these the *Inner-LevelILP*s. See lines 2 through 4 of Figure 11. We can also ignore the two network terms in the objective function, since they are now constants. Instead, we add the few additional constraints missing from the top level problem. We solve the lower level problems exactly, as ILPs, using binary decision variables. Again we employ CPLEX.

—At the completion of the *BottomLevelILP*s we should have a *macroW* solution. However, several potential problems may occur. For example, one of the CPLEX problems could fail to solve within time allotted to it by *SODA*. Second, feasible solutions may not actually be found, perhaps due to the hierarchical structure of our solution approach. And, finally, further improvements may be possible to the solution, because of the continuous relaxation assumption made, or because we now remove the assumption that all candidate nodes for a given PE be contained within a single cluster. The good news is that the structure of the *macroW* problem naturally lends itself to a local search heuristic. So we employ such a heuristic, *miniW*, that serves as a back-up to the first two CPLEX stages of *macroW*, and also as a post-processing heuristic to any solution CPLEX does provide. See line 5 of Figure 11.

Ultimately, we output the candidate node decisions on line 6.

```
1: Solve *TopLevelMILP*
2: **for** $\kappa = 0$ to $C-1$ **do**
3:     Solve *BottomLevelILP*($\kappa$)
4: **end for**
5: Solve *MiniW*
6: Output candidate node decisions $v_{p,n}$
```

Fig. 11. *macroW* Pseudocode

5.2.1 *Notation.* Let C denote the number of clusters, indexed by κ. The cluster which contains PN n will be denoted by κ_n. Let H_n denote the processing power of PN n, in *mips*. (If a PE p is assigned to a multi-core PN n and is single-threaded, it will not be able to use all of the processing power. To handle this and more general scenarios we define $H_{p,n}$ to be the maximum *mips* PE p can use on PN n.) Also let $\mathcal{E}$ denote the set of links (edges) in the network connecting the various clusters, indexed by e. Let K_e denote the bandwidth capacity of link e. Assuming that the network routing table forms a tree, the removal of any link partitions the clusters into two sets $k_{e,1}$ and $k_{e,2}$. (One can decide arbitrarily which is partition 1 and which is partition 2.)

We obtain from *macroQ* the *mips* goals g_p for each PE p. Let $\mathcal{S}$ denote the set of PE pairs (p_1, p_2) which are connected by a stream. As noted, we also obtain from *macroQ* the traffic rate estimate t_{p_1,p_2} for each $(p_1, p_2) \in \mathcal{S}$.

There will be a number of decision variables. We define the most important ones next. The key variable is $v_{p,n}$, which will be 1 if PN n is a candidate node for PE p, and 0 otherwise. (In solving *TopLevelMILP*, however, we will relax this to be a fractional rather than a binary variable.) All other decision variables will be computable once the candidate node decision variables are known. Specifically, we add an auxilliary binary decision variable $w_{p,\kappa}$, which we will turn out to be 1 if cluster κ contains the candidate nodes for PE p, and otherwise. To handle the traffic issues, we add a decision variable $y_{p_1,p_2,e}$, which we will force to be 1 if $(p_1, p_2) \in \mathcal{S}$ and uses link e, and 0 otherwise. Two auxilliary binary decision variables are needed for this purpose. The first, $y_{p_1,p_2,e,1}$, will be 1 if either PE p_1 or PE p_2 (or both) lies in the first cluster set, $k_{e,1}$, and 0 otherwise. The second, $y_{p_1,p_2,e,2}$, will serve the analogous purpose for the second cluster set $k_{e,2}$. A variety of constraints will force the decision variable behavior we have described.

Variable	Definition
C	Number of clusters
κ_n	Cluster containing PN n
H_n	*mips* of PN n
$H_{p,n}$	Maximum usable *mips* for PE p on PN n
$\mathcal{E}$	Set of links between clusters in network
K_e	Bandwidth capacity of link e
$k_{e,1}$	Cluster set 1 formed by link e
$k_{e,2}$	Cluster set 2 formed by link e
g_p	*mips* allotted to PE p
$\mathcal{S}$	Set of PE pairs connected by streams
t_{p_1,p_2}	Traffic rate on stream between p_1 and p_2
N	Number of PNs
E	Number of links
$A_{p,n}$	1 if PN n is resource matched to PE p, 0 otherwise
L	Set of (floating) licenses
λ_l	Number of available tokens for license l
π_l	Set of PEs that require license l
i_p	Integrity level of PE p
Δ_n	Relative integrity range for PN n
i_{min}	Minimum integrity level over all PEs
i_{max}	Maximum integrity level over all PEs
$r_{p,n}$	Risk of using PN n as a candidate node for PE p
R	Global risk limit
c_p	Color of PE p
X_n	Incompatible color pairs for PN n
ϕ_p	Minimum aggregate multiple of g_p allocated to PE p
$\hat{v}_{p,n}$	Candidate node assignment for PE p and PN n in previous macro epoch
$\bar{\Delta}$	Maximum cumulative candidate node change from previous macro epoch
$\bar{\Delta}_p$	Maximum cumulative candidate node change for PE p from previous macro epoch
V_n	Maximum number of PEs allowed on PN n
$\tilde{m}_p$	Minimum number of candidate nodes for PE p
$\tilde{M}_p$	Maximum number of candidate nodes for PE p
$\bar{C}$	Set of pairs of colocated PEs
$\bar{E}$	Set of pairs of exlocated PEs
W_1	Average PN utilization weight
W_2	Maximum PN utilization weight
W_3	Average link utilization weight
W_4	Maximum link utilization weight

Table II. Key *macroW* Notation

To handle the objective function, we will define two auxilliary real-valued variables. α_n will represent the utilization of PN n, and β_e will represent the utilization of link e. As previously noted, these two variables may exceed 1. (We note that these two decision variables are for notational convenience. They are not crucial

Variable	Type	Purpose
$v_{p,n}$	Binary	1 if PN n is a candidate node for PE p, 0 otherwise
$w_{p,\kappa}$	Binary	1 if cluster κ contains the candidate nodes for PE p, 0 otherwise
$y_{p_1,p_2,e}$	Binary	1 if PEs p_1 or p_2 uses edge e, 0 otherwise
$y_{p_1,p_2,e,1}$	Binary	1 if PEs p_1 or p_2 lies in cluster set $k_{e,1}$, 0 otherwise
$y_{p_1,p_2,e,2}$	Binary	1 if PEs p_1 or p_2 lies in cluster set $k_{e,2}$, 0 otherwise
α_n	Real	Utilization of PN n
β_e	Real	Utilization of link e
u_n	Integer	Maximum integrity level of all PEs p which have PN n as candidate node
$x_{c,n}$	Binary	1 if there exists a PE p with $v_{p,n} = 1$ and color $c = c_p$, 0 otherwise
$\Delta_{p,n}$	Binary	1 if assignment for PE p and PN n differs from previous macro epoch, 0 otherwise

Table III. *macroW* Decision Variables

to the problem itself. Without them the problem can be regarded as having only integer decision variables.)

Let N denote the number of PNs and E denote the number of network links.

Now we will discuss notation for the various practical constraints. These are either user-defined input or automatically calculated.

(1) *Resource matching*: For each PE, *macroW* is given a list of potential candidate nodes. These are the PNs where this PE will be allowed to run. The set may contain all PNs. But there are many possible reasons to restrict the set of candidate nodes for a particular PE. Hardware requirements are one example, and we shall give others shortly. Define $A_{p,n}$ to be 1 if PN n is resource matched to PE p, and 0 otherwise.

(2) *Licensing*: Some PEs use software for which licenses are required. The most common type is the *floating*, or standard license. Let L denote the set of floating licenses, indexed by l. For each $l \in L$ let λ_l denote the number of tokens available, and let π_l denote the set of PEs which require this license. *macroW* can also handle *node-locked* licenses, directly via resource matching constraints.

(3) *Security*: *macroW* supports four types of security constraints. Each PE p is assigned an integral *integrity level* i_p. The first type of security constraint limits the *relative* integrity levels on a given PN n. Let Δ_n denote the maximum allowable difference in integrity level between any two PEs which use PN n as a candidate node. Define $i_{min} = \min_p i_p$ and $i_{max} = \max_p i_p$. We will add an integral decision variable u_n between i_{min} and i_{max}, which will serve as the maximum integrity value of any PE p assigned to PN n. The second type of security constraint limits the absolute integrity on a given PN n between a predetermined minimum and maximum. However, this can again be handled via resource matching constraints. The third type of security constraint limits *global risk*. Each PE p is assigned a risk level $r_{p,n}$ if PN n is a candidate node. The constraint limits the global risk across all PNs and PEs to some fixed maximum R. The last type is the so-called *Chinese Wall* security constraint. We can best describe this in terms of *colors*. Assume that each PE p has a

color c_p. For each PN n there is a set X_n of incompatible color pairs. In other words, if $(c_{p_1}, c_{p_2}) \in X_n$, then PEs p_1 and p_2 cannot both be assigned to PN n. To model this we add an additional decision variable $x_{c,n}$ which will be 1 if there exists a PE with color c and PN n as a candidate node, and 0 otherwise.

(4) *Fairness*: To ensure that each PE p get a fair allocation of candidate PNs we let ϕ_p denote the minimum aggregate multiple of the goal g_p allocated to PE p among all assigned PNs.

(5) *Incremental changes*: Several constraints ensure that the candidate node assignments do not change too significantly from one macro epoch to another. To model these, let $\hat{v}_{p,n}$ denote the candidate node assignment for PE p and PN n in the previous macro epoch. We will form an auxilliary decision variable $\Delta_{p,n}$, which will be 1 if assignment for PE p and PN n has changed from previous macro epoch, and 0 otherwise. And we will constrain the incremental changes on a global and a per PE level. Let $\bar{\Delta}$ denote the maximum allowable cumulative candidate node change from the previous macro epoch, and $\bar{\Delta}_p$ denote the maximum allowable candidate node change for PE p from the previous macro epoch. Of course, these maxima may be infinite, resulting in no incremental change limits. Or they may be 0, fixing the corresponding candidate node assignments.

(6) *Cardinality constraints*: There are three separate cardinality constraints. Let V_n denote the maximum number of PEs allowed on PN n. Let $\tilde{m}_p$ denote the minimum number of candidate nodes for PE p, and $\tilde{M}_p$ denote the maximum.

(7) *Colocation*: *macroW* can force two PEs p_1 and p_2 to have the same candidate nodes. Let $\bar{C}$ denote the set of pairs of colocated PEs.

(8) *Exlocation*: Similarly, *macroW* can force two PEs to never have the same candidate nodes. Let $\bar{E}$ denote the set of pairs of exlocated PEs. One can also force a PE *isolation* constraint using exlocation. In other words, PE p can be isolated on its own candidate nodes by adding (p, p') to $\bar{E}$ for all $p' \neq p$. (One could also enforce isolation using a weighted version of one of the cardinality constraints, but we omit details.)

5.2.2 *Mathematical Formulation.* Although we will solve variants of the *macroW* problem in the three stages below we describe here the "actual" problem. (In fact, only *miniW* attempts to solve the exact problem formulated here.)

We seek to minimize the weighted sum of four terms involving the average and maximum utilizations of the PNs and links. Let W_1 through W_4 denote these weights. The objective is therefore to find

$$\min \left[\frac{W_1}{N} \left(\sum_n \alpha_n \right) + W_2 (\max_n \alpha_n) + \frac{W_3}{E} \left(\sum_e \beta_e \right) + W_4 (\max_e \beta_e) \right]$$

subject to the following constraints:

$$\alpha_n = \sum_p H_{p,n} v_{p,n} / H_n \qquad \forall n \tag{4}$$

$$\beta_e = \sum_{p_1} \sum_{p_2} t_{p_1,p_2} y_{p_1,p_2,e} / K_e \qquad \forall e \tag{5}$$

$$v_{p,n} \le w_{p,\kappa_n} \qquad \forall p, n \tag{6}$$

$$w_{p_1,\kappa} \le y_{p_1,p_2,e,1} \qquad \forall (p_1,p_2) \in \mathcal{S}, e \in \mathcal{E}, \kappa \in k_{e,1} \tag{7}$$

$$w_{p_2,\kappa} \le y_{p_1,p_2,e,1} \qquad \forall (p_1,p_2) \in \mathcal{S}, e \in \mathcal{E}, \kappa \in k_{e,1} \tag{8}$$

$$w_{p_1,\kappa} \le y_{p_1,p_2,e,2} \qquad \forall (p_1,p_2) \in \mathcal{S}, e \in \mathcal{E}, \kappa \in k_{e,2} \tag{9}$$

$$w_{p_2,\kappa} \le y_{p_1,p_2,e,2} \qquad \forall (p_1,p_2) \in \mathcal{S}, e \in \mathcal{E}, \kappa \in k_{e,2} \tag{10}$$

$$y_{p_1,p_2,e} \ge y_{p_1,p_2,e,1} + y_{p_1,p_2,e,2} - 1 \qquad \forall (p_1,p_2) \in \mathcal{S}, e \in \mathcal{E} \tag{11}$$

$$v_{p,n} \le A_{p,n} \qquad \forall p, n \tag{12}$$

$$\sum_{p \in \pi_l} \sum_n v_{p,n} \le \lambda_l \qquad \forall l \in L \tag{13}$$

$$i_p v_{p,n} \ge u_n - \Delta_n - (i_{max} - \Delta_n)(1 - v_{p,n}) \qquad \forall p, n \tag{14}$$

$$(1 - v_{p,n})(i_{min} - i_p) \le u_n - i_p \qquad \forall p, n \tag{15}$$

$$\sum_p \sum_n r_{p,n} v_{p,n} \le R \tag{16}$$

$$v_{p,n} \le x_{c_p,n} \qquad \forall p, n \tag{17}$$

$$x_{c_1,n} + x_{c_2,n} \le 1 \qquad \forall (c_1,c_2) \in X_n \tag{18}$$

$$\sum_n H_{p,n} v_{p,n} \ge \phi_p g_p \qquad \forall p \tag{19}$$

$$\Delta_{p,n} \le v_{p,n} - \hat{v}_{p,n} \qquad \forall p, n \tag{20}$$

$$\Delta_{p,n} \le \hat{v}_{p,n} - v_{p,n} \qquad \forall p, n \tag{21}$$

$$\sum_p \sum_n \Delta_{p,n} \le \bar{\Delta} \tag{22}$$

$$\sum_n \Delta_{p,n} \le \bar{\Delta}_p \qquad \forall p \tag{23}$$

$$\sum_p v_{p,n} \le V_n \qquad \forall n \tag{24}$$

$$\tilde{m}_p \le \sum_n v_{p,n} \qquad \forall p \tag{25}$$

$$\sum_n v_{p,n} \le \tilde{M}_p \qquad \forall p \tag{26}$$

$$v_{p_1,n} = v_{p_2,n} \qquad \forall (p_1,p_2) \in \bar{C}, n \tag{27}$$

$$v_{p_1,n} \ne v_{p_2,n} \qquad \forall (p_1,p_2) \in \bar{E}, n \tag{28}$$

The first set of constraints are structural in nature. They ensure the right behavior of the various decision variables. Constraint 4 is the definition of α_n, the utilization of PN n. Similarly, Constraint 5 is the definition of β_e, the utilization of link e. Constraint 6 forces consistency: The cluster decision variable w_{p,κ_n} to be 1 whenever the PN decision variable $v_{p,n}$ is. Similarly, Constraints 7 through 10 force consistency for the first and second cluster set decision variables. Taken together, Constraint 11 then implies consistency for the link decision variable $y_{p_1,p_2,e}$.

Next there are user constraints. Constraint 12 enforces resource matching. Likewise, Constraint 13 handles licensing. Constraints 14 and 15 enforce relative security. The trick employed is somewhat subtle, but the basic idea is that i_p is forced to lie between $u_n - \Delta_n$ and u_n whenever PN n is a candidate node for PE p. The decision variable u_n is itself forced to lie between i_{min} and i_{max}, and will take the maximum integrity level as its value if forced to do so for feasibility. (This will happen when the constraint is actually tight.) Constraint 16 bounds the global security risk. Chinese Wall security is handled by Constraints 17 and 18. The former forces the Chinese Wall decision variable $x_{c_p,n}$ to be 1 whenever PN n is a candidate node for PE p. The latter then ensures that PEs with incompatible color pairs are never assigned to the same candidate PN. Constraint 19 ensures fairness among the various PEs. (In fact, given the objective function, this is the single constraint forcing *macroW* to allocate sufficient resources to the various PEs.) Constraints 20 and 21, taken together, insist that $\Delta_{p,n}$ is 1 precisely when the candidate node assignment for PE p and PN n have changed from the previous macro epoch. Next Constraint 22 limits the global incremental changes, while Constraint 23 limits the incremental changes for each PE. Constraint 24 limits the cardinality of PEs assigned to each PN. Constraints 25 and 26 ensure that the cardinality of the candidate nodes for a particular PE falls between the minimum and maximum. Finally, Constraints 27 and 28 enforce PE colocation and exlocation, respectively.

There are the normal constraints on the decison variables themselves. For example, integral and continuous variables must be at least 0. We do not write these here, since they are implicit from Table III.

The second and fourth terms in the *macroW* objective function may not look linear at first glance, but the introduction of two auxilliary variables will transform the problem into standard linear form. (This is a standard trick.)

5.2.3 *Solution Approach.* First, some generic comments about reducing the problem size for each stage are in order. Although the formulation implies otherwise, we point out that constraints such as resource matching need not be modeled explicitly. We can simply define the variable $v_{p,n}$ *only* if $A_{p,n} = 1$. Note also that the PE pairs $(p_1, p_2) \in \mathcal{E}$ are likely to be relatively sparse, much smaller in cardinality than the square of the number of PEs. No PE pairs not in $\mathcal{E}$ need be considered. The colocation constraint can be modeled via creation of coalesced "super-PEs" with appropriately recomputed constraints, and thus not considered separately. And, of course, any constraints that are not actually enforced do not need to be added to the problem given to CPLEX. Recall Figure 11.

5.2.3.1 *TopLevelMILP.* : At the top level in *macroW*, we decide the (currently single) cluster to which each PE will be assigned. But we do this "approximately" in several ways. First, we solve an MILP in which we allow each variable $v_{p,n}$ to be continuous rather than binary. We do insist that the variable is fractional: This means we force $0 \leq v_{p,n} \leq 1$. This relaxation makes the problem substantially quicker to solve. On the negative side, the single cluster assumption adds an additional constraint, specifically

$$\sum_{\kappa} w_{p,\kappa} = 1 \qquad \forall p \tag{29}$$

On the positive side, the assumption allows us to reduce the cardinality of the constraint set, replacing Constraints 7 through 10 with

$$\sum_{\kappa \in k_{e,1}} w_{p_1,\kappa} \leq y_{p_1,p_2,e,1} \qquad \forall (p_1,p_2) \in \mathcal{S}, e \in \mathcal{E} \tag{30}$$

$$\sum_{\kappa \in k_{e,2}} w_{p_2,\kappa} \leq y_{p_1,p_2,e,1} \qquad \forall (p_1,p_2) \in \mathcal{S}, e \in \mathcal{E} \tag{31}$$

$$\sum_{\kappa \in k_{e,1}} w_{p_1,\kappa} \leq y_{p_1,p_2,e,2} \qquad \forall (p_1,p_2) \in \mathcal{S}, e \in \mathcal{E} \tag{32}$$

$$\sum_{\kappa \in k_{e,2}} w_{p_2,\kappa} \leq y_{p_1,p_2,e,2} \qquad \forall (p_1,p_2) \in \mathcal{S}, e \in \mathcal{E} \tag{33}$$

Finally, we omit constraints that do not make sense in the presence of fractional variables $v_{p,n}$. These include relative security, Chinese Wall security, incremental change, the maximum PEs per PN, and exlocation constraints. So we remove Constraints 14, 15, 17, 18, 20 through 24 and 28. We enforce them instead in the subsequent stages of *macroW*. The problem is solved by CPLEX.

5.2.3.2 *BottomLevelILPs.* : After the *TopLevelMILP* stage we know the cluster to which each PE is assigned. Moreover, this phase also provides information about how many resources (such as licenses) are required by each cluster. This allows to decompose the problem and solve it for each cluster independently. At the *BottomLevelILP* stage we solve the problem exactly using integer variables for the decision variables $v_{p,n}$, since each problem is now much smaller than the top-level one.

There are other clear differences between each bottom-level problem and the top-level problem. The cluster variables $w_{p,\kappa}$ are irrelevant, because we are only considering PEs that are assigned to the cluster under consideration. Only the first two summands now appear in the objective function, since inter-cluster traffic (and the variables β_e) are no longer relevant. We also naturally and dynamically apportion certain global terms in the top-level problem to bottom-level quotas as we work through the various cluster problems. Among these are the risk, license and reassignment quotas. For example, consider the value $\sum_p \sum_n r_{p,n} v_{p,n}$ in Constraint 16 of the *TopLevelMILP* solution. We partition this value into quotas $\sum_p \sum_n r_{p,n} v_{p,n} w_{p,\kappa}$ for each cluster κ. Even though the values $v_{p,n}$ are fractional, this term still makes sense. Next we apportion the *slack* $R - \sum_p \sum_n r_{p,n} v_{p,n}$ evenly among the clusters. Finally, as we loop through the various clusters, we continue to evenly apportion the per cluster slack among the subsequent clusters. As noted, the *BottomLevelILP* does consider relative security, Chinese Wall security, incremental change, the maximum PEs per PN, and exlocation constraints. The problem is again solved by CPLEX.

5.2.3.3 *miniW.* : As we have noted, the nature of the *macroW* problem lends itself to a local search heuristic [Papadimitriou and Steiglitz 1982]. Such heuristics are natural and easy to implement. On the other hand, they do not generally solve to optimality.

The *miniW* local search heuristic is useful for several reasons. First, CPLEX may fail in one of the first two stages of *macroW*. Even if CPLEX does not fail per se, one of the problems may not converge to a solution by its internal *macroW* deadline. Second, the hierarchically decoupled nature of the first two stages may prevent the finding of an existing feasible solution. Third, we have cut some corners in the first two stages, so the quality of the solution found may be compromised. The continuous relaxation in *TopLevelMILP* is one example. (The *macroW* problem is inherently quadratic in nature, and this may result in weak LP relaxations.) The simplifying assumption that all candidate nodes lie in a single cluster is another. A good heuristic solution to an accurate version of the problem may actually perform as well or better than an exact solution to an approximate version of the problem. Fourth, and finally, even if all goes well in the first two stages of *macroW*, there may still be time left in the *macroW* epoch, and using this time to improve the quality of the solution is an obviously good idea. In fact, this is by far the most common *macroW* scenario.

We will briefly outline the *miniW* local search heuristic here. Note that we will discuss only the candidate node choices, that is, the values of the decision variables $v_{p,n}$. (The values of all other decision variables will then automatically be determined.)

The *initialization* phase provides a first feasible solution. If the *TopLevelMILP* and *BottomLevelILP* stages have completed successfully, there is nothing to be done in this phase. If not, there are two cases: Either this is the first *macro* epoch, or we are in some subsequent epoch.

—In the former case, we order the streams based on their traffic rates, and proceed greedily, finding candidate nodes for the source and destination PEs of this stream. At any given stage we may find that one or both of these PEs has been assigned candidate nodes. For those PEs that have not, *miniW* makes the smallest number of candidate node assignments which meet the fairness and minimum cardinality constraints (Constraints 19 and 25, respectively), satisfies all other feasibility requirements, and minimizes the objective function. Ties are adjudicated in favor of choices which retain the most slackness in the constraints.

—In the latter case, the solution from the previous epoch is adopted, at least for the PEs that are not new this epoch. This obviously eliminates incremental movement, at least for the moment. Because of changes to the input data it is possible that this solution may not be fully feasible, but a greedy scheme is employed modified as necessary eliminate these infeasibilities. The new PEs are then assigned using an approach like that for *macro* epoch 1.

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.

Similarly, it may add or remove a candidate node for a PE, if that is feasible. In the neighborhood search literature these are traditionally called *1-opt* moves. If no useful 1-opt moves exists the algorithm will consider the moving of both a candidate node for a source PE and a candidate node for a destination PE associated with some stream, again assuming feasibility and a decrease in the objective function. These are called 2-opt moves. Note that candidate node swaps are included in this category, as are the movement of both candidate nodes to a third PN. Such a move can have the effect of trading a reduction in link utilization for an increase in PN utilization. Each of these techniques can be helped by judicious orderings of the PEs, streams and PNs. 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. PNs are ordered by decreasing load. Streams are ordered by decreasing traffic rates. In general this process can continue in a local search heuristic through k-opt moves, for an arbitrary value k. However, the neighborhood sizes typically increase exponentially with k. So *miniW* stops at $k = 2$.

Finally, there may be a *perturbation* phase. *miniW* is designed to run until it reaches its deadline. But the local improvement phase may reach a local minimum, and thus be unable to improve the solution. So if the local improvement phase reaches a locally optimal solution before the deadline, *miniW* will perturb that solution, insisting on feasibility but ignoring the fact that the solution does not improve. This allows the heuristic to possibly *escape* a local minimum, and the local improvement phase proceeds iteratively in this manner until the deadline is reached. The best solution found at the *macroW* deadline becomes the *macroW* output.

5.3 *microQ*

The role of *microQ*, the *micro quantity* model, is to adjust the *mips* goals of the various PEs for use during the next micro epoch. Although *macroQ* also computes *mips* goals, recall that it does so in advance of knowing the candidate nodes from *macroQ* to which each PE can be assigned. So this alone may force a refinement to the *macroQ* goals. Furthermore, *microQ* must adjust to dynamic changes occurring at the micro epoch level.

microQ takes input from the macro level problem, including the importance functions and goals from *macroQ*, and the candidate nodes from *macroW*. It also knows the current set of active PNs and jobs. Recall that algorithmic speed is critical for *microQ*, because it must be solved, along with *microW*, within a *micro* epoch. In particular, a computational complexity comparable to that of the *macroQ* scheme is not likely to be acceptable for *microQ*. We handle this by employing two approximations derived from *macroQ* output. The first is a piecewise linear approximation for each weak component of importance as a function of allotted resources. The second is a linear approximation, within each weak component and each piecewise linear segment, of the fraction of the resources allotted to each PE in the component. Together, these approximations allow us to solve the *microQ* problem iteratively to convergence, in each iteration solving computationally easy linear programs (LPs).

Table IV provides a summary of notation used, in order of appearance. And Figure 12 provides an outline of the *microQ* pseudo-code.

Variable	Definition
H_n	*mips* of PN n
$v_{p,n}$	1 if PN n is a candidate node for PE p, 0 otherwise
$\mathcal{P}_c$	Set of PEs in weak component c
$\mathcal{I}_c$	Piecewise linear importance function for weak component c
η_c	Number of piecewise linear segments in $\mathcal{I}_c$
$\varepsilon_{r,c}$	Right endpoint for rth segment of $\mathcal{I}_c$
$\sigma_{r,c}$	Slope of rth segment of $\mathcal{I}_c$
$\mathcal{R}_{c,p,r}$	Linear pacing constraint function for PE $p \in \mathcal{P}_c$
m_p	Minimum *mips* for PE p
M_p	Maximum *mips* for PE p
g_p	*mips* allotted to PE p
G_c	*mips* allotted to component c
$f_{p,n}$	Fraction of PN n used by PE p
$y_{r,c}$	1 if G_c is in rth segment of $\mathcal{I}_c$, 0 otherwise
$\tilde{v}_{c,n}$	1 if there exists PE $p \in \mathcal{P}_c$ for which $v_{p,n} = 1$, 0 otherwise
$\tilde{f}_{c,n}$	Fraction of PN n used by PEs from component c
r_c	Current piecewise linear segment for component c

Table IV. Key *microQ* Notation

```
1: Compute piecewise linear approximation to importance function
2: Compute pacing constraints
3: Solve preprocessor network flow problem, computing $\{r_c\}$
4: Set $OPT = 0$
5: Set $CONVERGED$ = false
6: while $CONVERGED$=false do
7:   $CONVERGED$=true
8:   Solve iterative LP, computing $\mathcal{I}$ and $\{r_c{}'\}$
9:   if $\mathcal{I} > \mathcal{OPT}$ then
10:     $OPT = \mathcal{I}$
11:   end if
12:   for $c = 0$ to $c = C$ do
13:     if $r_c{}' < r_c$ then
14:       $CONVERGED$=false
15:       $r_c - -$
16:     else if $r_c{}' > r_c$ then
17:       $CONVERGED$=false
18:       $r_c$++
19:     end if
20:   end for
21: end while
22: Output $OPT$
```

Fig. 12. *microQ* Pseudocode

5.3.1 *Notation.* We use the following input data. Recall that H_n denotes the total processing power, in *mips*, of PN n. Let $v_{p,n}$ be 1 if PN n is a candidate node for PE p, and 0 otherwise. This is output from *macroW*. Index the set of weak components derived in *macroQ* by c, and let $\mathcal{P}_c$ denote the set of PEs in component c. As noted, *microQ* first fits, from the component importance function of allotted resources computed in *macroQ*, a piecewise linear concave approximation. By abuse of notation we will also call this new function $\mathcal{I}_c$. (Standard approximation techniques are employed, and we will omit details. Recall that we expect most importance functions to have an essentially concave structure, because of a diminishing returns argument. So forcing concavity should not, in general, be a major assumption.) Let η_c be the number of piecewise linear segments in $\mathcal{I}_c$. Let $\varepsilon_{r,c}$ denote the right endpoint, in *mips*, for the r^{th} piecewise linear segment of $\mathcal{I}_c$. Let $\sigma_{r,c}$ denote the slope of this segment. Second, *microQ* computes so-called *pacing constraints*, for each piecewise linear segment r, each component c, and each PE $p \in \mathcal{P}_c$. These pacing constraints specify the linear proportion of the incremental *mips* to be allocated for PE p within segment r. (Again, standard approximation techniques are used. If the number of piecewise linear segments η_c is reasonably large, these linear approximations should be sufficiently accurate.) Said differently, we assume each entry for PE p and piecewise linear segment r is a linear function $\mathcal{R}_{c,p,r}(g)$, and that the entries in segment r satisfy the properties $\sum_{p\in\mathcal{P}_c} \mathcal{R}_{c,p,r}(g) = g$, $\mathcal{R}_{c,p,r}(\varepsilon_{r,c}) = \mathcal{R}^c_{p,r+1}(\varepsilon_{r,c})$ and $\mathcal{R}_{c,p,r}(g) \geq 0$ for any value of g in the rth segment. As before, let m_p and M_p denote the minimum and maximum *mips* which can be given to PE p.

We employ the following four decision variables: Let g_p denote the processing power goal, in *mips*, for PE p. This is the prime output to be used by *microW*. Similarly, let G_c denote the processing power goal, in mips, for component c. This will be the sum of all the processing power goals of the PEs p in component c. A third decision variable, $f_{p,n}$, will denote the fraction of PN n used by PE p. Finally, there will be an indicator decision variable $y_{r,c} \in \{0,1\}, r = 1,\ldots,\eta_c$. This auxilliary variable will be 1 if G_c is part of the r^{th} piecewise linear segment of $\mathcal{I}_c$, and 0 otherwise.

5.3.2 *Mathematical Formulation.* The objective is to find

$$\max \sum_c \mathcal{I}_c(g_c)$$

subject to the following constraints:

$$\sum_r y_{r,c} = 1 \qquad \forall\, c \tag{34}$$

$$\varepsilon_{r-1,c} y_{r,c} \leq g_c \qquad \forall\, c,\ 1 \leq r \leq \eta_c \tag{35}$$

$$g_c - \varepsilon_{\eta_c,c} \leq [\varepsilon_{r,c} - \varepsilon_{\eta_c,c}] \sum_{i=r+1}^{\eta_c} y_{i,c} \qquad \forall\, c \in C, 1 \leq r \leq\ \eta_c \tag{36}$$

$$g_p = y_{r,c}\mathcal{R}_{c,p,r}(g_c) \qquad \forall\, c,\ p \in \mathcal{P}_c,\ 1 \leq r \leq \eta_c \tag{37}$$

$$\sum_{v_{p,n}=1} H_n f_{p,n} = g_p \qquad \forall\, p \tag{38}$$

$$\sum_{v_{p,n}=1} f_{p,n} \leq 1 \qquad \forall\, n \tag{39}$$

$$m_p \leq g_p \leq M_p \qquad \forall p \tag{40}$$

$$y_{r,c} \in \{0,1\} \qquad \forall\, c, 1 \leq r \leq \eta_c \tag{41}$$

$$f_{p,n} \geq 0 \qquad \forall\, p, n \tag{42}$$

$$g_c \geq 0 \qquad \forall\, c \tag{43}$$

Constraint 34 ensures that precisely one piecewise linear segment will be chosen for each weak component. Constraints 35 and 36, taken together, force G_c to be between the left and right endpoints of that segment. (Constraint 36 is somewhat technical.) Constraint 37 determines g_p based on the appropriate pacing constraint. Constraint 38 ensures that g_p can be achieved by the fraction assignments $f_{p,n}$ to the various PNs, while Constraint 39 ensures that these fractions are not beyond range. Constraint 40 requires the goals g_p to be within their minimum and maximum values. Finally, Constraint 41 insists that the variables $y_{r,c}$ be binary, and Constraints 42 and 43 ensure that the other decision variables are at least 0.

5.3.3 *Solution Approach.* The above optimization problem is difficult on its face. Some variables are binary. The objective function, while separable and concave, is non-linear, and the pacing constraints, taken together, are non-linear and not concave. We therefore do not solve the problem directly. Instead, we take the following approach. We first solve a simpler problem as a preprocessing step, to obtain an initial estimate of the segment in which each G_c lies. (The simpler problem is solvable via network flow or LP techniques, using CPLEX [ILOG].) The solution determines a set of pacing constraints to enforce. We then solve an LP that is a network flow problem with these additional linear pacing constraints, also using CPLEX. If the values of G_c returned by this program lie in the same segment as the initial estimates, then we are done. Otherwise, we modify our initial estimate of the segments, impose the newly appropriate pacing constraints, and re-solve. The final solution is obtained when this iterative process converges, or when time runs out. In the latter case, we take the best solution (according to objective value) seen so far.

In more detail, the preprocessor solves a variant of the *microQ* problem, maximizing the importance, but ignoring pacing constraints entirely. We essentially "blur" the PE information in the problem. Instead of using $v_{p,n}$, we define $\tilde{v}_{c,n}$ to be 1 if there exists a PE $p \in \mathcal{P}_c$ for which $v_{p,n} = 1$, and 0 otherwise. Instead of using the decision variable $f_{p,n}$ we employ a component-based decision variable $\tilde{f}_{c,n}$. (The other decision variables are G_c and $y_{r,c}$, as before.) The problem has the same objective function as before, but with Constraints 34 through 36, 43, and

$$\sum_{\tilde{v}_{c,n}=1} H_n \tilde{f}_{c,n} = g_c \qquad \forall\, c \tag{44}$$

$$\sum_{\tilde{v}_{c,n}=1} \tilde{f}_{c,n} \leq 1 \qquad \forall\, n \tag{45}$$

$$\tilde{f}_{c,n} \geq 0 \qquad \forall\, c, n \tag{46}$$

These three are the natural analogues of Constraints 38, 39 and 42, respectively. Because of the piecewise linear, concave objective function this problem can be

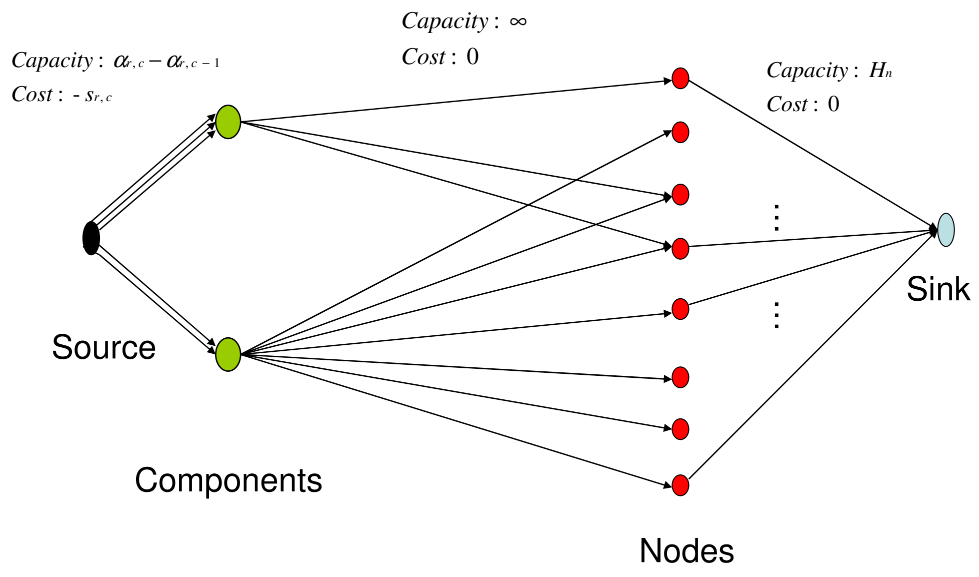


Fig. 13. *microQ*: Preprocessor Network Flow Problem

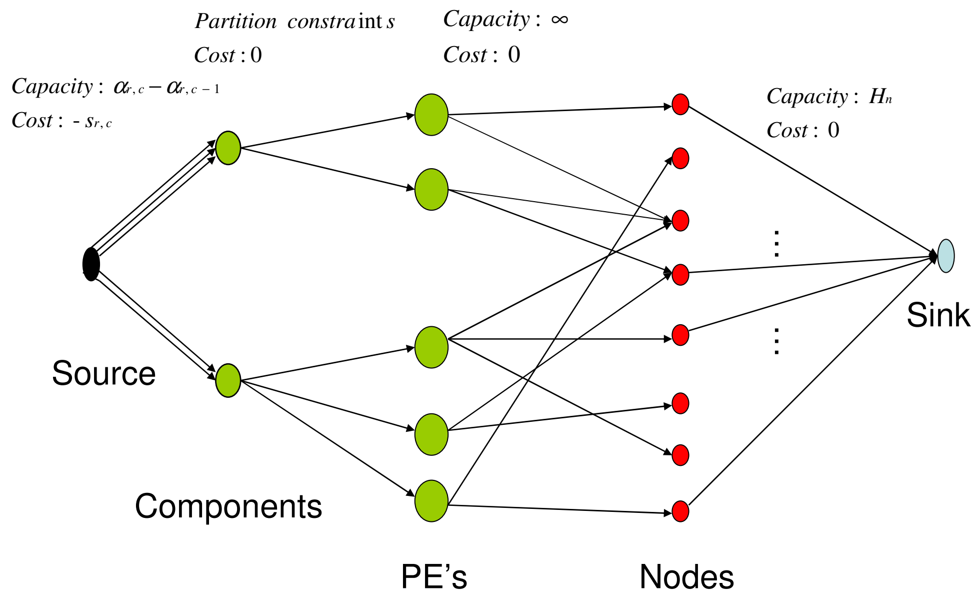


Fig. 14. *microQ*: Iterative "Network Flow" Problem

converted via a standard trick into an LP, and more specifically into a minimum cost network flow problem [Ahuja et al. 1993].

To understand this network flow problem, see Figure 13. There is a source, a sink, a node for each component c, and a node for each PN n. The source node s is connected to a component node c with η_c arcs. Each arc represents one of the piecewise linear segments. The capacity of segment r is $\varepsilon_{r,c} - \varepsilon_{r-1,c}$. The cost of segment r is the negative of the slope $\sigma_{r,c}$ of the component importance function. There is an arc from a component node c to a PN node n if and only if $\tilde{v}_{c,n} = 1$. The capacity of these arcs is infinite and the cost is 0. All PNs are connected to the sink node t. The arc from a PN n to t has capacity H_n and cost 0.

We compute a minimum cost flow in this network, which maximizes importance. The "flow" through the network is in *mips*. Then G_c is the sum of flow on the parallel arcs from s to c. (The trick mentioned above is that the flow will automatically fill from left to right segment because of the concavity of the importance function.)

This preprocessing stage yields initial estimates of G_c for each component c, and the corresponding piecewise linear segment r_c (which satisfies $y_{r_c,c} = 1$) determines the first set of pacing constraints to enforce for each component in the iterative stage of the problem. We now describe this problem, which has the basic structure of a network flow problem, but with the additional pacing constraints.

Consider Figure 14. The network is formed from the network of Figure 13 by adding in nodes for each PE p. These are between the component nodes and the PN nodes. The previous arcs from components to PNs are replaced by two sets of arcs. There is an arc from a component c to a PE node p if and only if $p \in \mathcal{P}_c$. The capacity of this arc is M_p and the minimum flow is m_p. The cost is 0. There is an arc from a PE node p to a PN node n if and only if $v_{p,n} = 1$. The capacity of these arcs is infinite and the cost is 0. As noted, for each component c there is are additional linear pacing constraints which determine, based on the flow into c, the flow from c to each PE $p \in \mathcal{P}_c$. We solve this LP, finding a flow that minimizes the cost subject to the additional constraints. This maximizes importance, as before. The new value of G_c is then the sum of the flows on the parallel arcs from s to c. The value g_p is the flow on the arc from c to p. The value $f_{p,n}$ is the flow on the arc from p to n, normalized by H_n.

Next, we check for each component c if the new value of G_c corresponds to the same piecewise linear segment as the previous value. Let ${r_c}'$ be this new segment. If G_c lies in a segment below its original segment r we switch to the pacing constraints for the $(r-1)$st segment. (In other words, if ${r_c}' < r_c$ we decrement r_c by 1.) Similarly, if G_c lies in a segment above its original segment r we switch to the pacing constraints for the $(r+1)$st segment. (If ${r_c}' > r_c$ we increment r_c by 1.) Then we iterate the process, solving the LP again. If at some iteration, the new and old values of G_c lie in the same segment for all components c, we have converged. So we output the PE goals. The linear program solves quickly, so we can afford many iterations. If it does not converge, however, due to cycling, or if we run out of time, we return the best feasible solution found.

Variable	Definition
$f_{p,n}$	Fraction of PN n used by PE p
χ_n	1 if PN n is touched, 0 otherwise
g_p	*mips* goal for PE p
$\hat{f}_{p,n}$	Fraction of PN n used by PE p in previous micro epoch
H_n	*mips* of PN n
δ	Maximum cumulative change in fractional assignment from previous micro epoch
δ_p	Maximum cumulative change in fractional assignment from previous micro epoch for PE p
ζ	Maximum number of PNs that can be modified from previous micro epoch
$\mathcal{A}_{p,n}$	Legal fractions of PN n for PE p
$v_{p,n}$	1 if PN n is a candidate node for PE p, 0 otherwise
$\mathcal{F}_p$	PNs whose fractional allocations for PE p are fixed from previous micro epoch
$\mathcal{G}$	Network flow directed graph of PNs, over- and under-allocated PEs
U	Number of under-allocated PEs
O	Number of over-allocated PEs
P	Number of PEs

Table V. Key *microW* Notation

5.4 *microW*

The goal of *microW* is to make new fractional assignments of PEs to PNs for the current *micro* epoch. The idea is to match as closely as possible the overall processing power goals computed for each PE by the *microQ* model, while respecting the candidate nodes computed by the *macroW* model and meeting additional practical constraints:

—Three of these are incremental in nature. One limits the cumulative amount change in fractional assignment values from the previous *micro* epoch. One limits this amount on a per PE basis. A third constraint limits the number of PNs that can be "touched" during the current *micro* epoch. (Touched here means any change in the fractional assignments to the PN.)

—Another constraint ensures that only acceptable fractional assignments are made. These restrictions may come from a variety of sources. For example, the only acceptable fractional assignment for a PE on a PN not chosen as a candidate node by the *macroW* model is 0. We will give other examples shortly.

—Still another constraint fixes certain fractional assignments to their values from the previous *micro* epoch.

—And a final constraint ensures that no PN becomes overloaded.

Table V provides a summary of notation used, in order of appearance. And Figure 15 provides an outline of the *microW* pseudo-code.

The *microW* problem is solved via suitably modified techniques borrowed from the network flow literature [Nemhauser and Wolsey 1988].

5.4.1 *Notation.* We will employ two decision variables. First, $f_{p,n} \in [0, 1]$ will be the fraction of PN n used by PE p. This is the primary output of the *microW*

```
1: Set $DONE$ = false
2: while $DONE$ == false do
3:   Set $DONE$ = true
4:   Compute $OPT = \sum_p |\sum_n f_{p,n} H_n - g_p|$
5:   if $OPT > 0$ then
6:     Index under-allocated PEs from most under-allocated (index 0) to least under-allocated
       (index $U - 1$)
7:     Index over-allocated PEs from least (index $P - O$) to most (index $P - 1$)
8:     Compute directed graph $\mathcal{G}$
9:     for $p_1 = 0$ to $U - 1$ do
10:      for $p_2 = P - 1$ to $P - O$ by $-1$ do
11:        Find shortest path in $\mathcal{G}$ from $p_1$ to $p_2$
12:        if path exists then
13:          Find maximal flow $f$ from $p_1$ to $p_2$ subject to constraints
14:          if $f > 0$ then
15:            Push flow $f$ from $p_1$ to $p_2$
16:            Go to line 4
17:          end if
18:        end if
19:      end for
20:    end for
21:  end if
22: end while
23: Output $OPT$
```

Fig. 15. *microW* Pseudocode

model. Second, $\chi_n \in \{0, 1\}$ will effectively be an indicator variable, being 1 if PN n is touched during the current *micro* epoch, and 0 otherwise.

We use the following input data. Let g_p be the processing power goal, in *mips*, for PE p and is the output of the *microQ* model. $\hat{f}_{p,n}$ is the fraction of PN n used by PE p during the previous *micro* epoch. H_n denotes the total processing power, in *mips*, of PN n. δ is the maximum allowable cumulative change in fractional assignment values from previous *micro* epoch. The number may be infinite. δ_p, on the other hand, denotes the maximum allowable cumulative change in fractional assignment values from previous *micro* epoch for PE p. The number may also be infinite. ζ will be the maximum number of PNs that can be modified from the previous *micro* epoch, again possibly infinite. $\mathcal{A}_{p,n} \subseteq [0, 1]$ consists of all legal fractional allocations for PE p on PN n. As noted, if $v_{p,n} = 0$, that is if *macroW* does not assign n as a candidate node for p, this subset will be $\{0\}$. Recalling the candidate node decision variable $v_{p,n}$, the prime output from the *macroW* model, this means that $v_{p,n} = 0$ implies that $\mathcal{A}_{p,n} = \{0\}$. Similarly, if p must use all or nothing of a PN n this subset will be $\{0, 1\}$. As a final example, a single-threaded PE cannot use more than one core at any time in a multiprocessor candidate node. So, on a 4-core PN such a PE would be restricted to fractional assignments between in the range $[0, .25]$. It is assumed in general that $\mathcal{A}_{p,n}$ is a finite set of (closed) ranges in the unit interval. In particular, it may be the entire unit interval, meaning that there are no restrictions on legal fractional allocations. $\mathcal{F}_p \subseteq \{0, ..., N\}$ is the subset of PNs whose fractional allocation for PE p is fixed from the last *micro* epoch.

5.4.2 *Mathematical Formulation.* We would like to reassign allocations of PEs to PNs in order to minimize the cumulative difference between the PE processing goals and achieved allocations. Hence the objective is to find

$$\min \sum_p \left| \sum_n f_{p,n} H_n - g_p \right|$$

subject to the following constraints:

$$\begin{aligned}
\sum_n \sum_p |f_{p,n} - \hat{f}_{p,n}| H_n &\leq \delta && (47)\\
\sum_n |f_{p,n} - \hat{f}_{p,n}| H_n &\leq \delta_p \qquad \forall p && (48)\\
(1 - \chi_n)(f_{p,n} - \hat{f}_{p,n}) &= 0 \qquad \forall p, n && (49)\\
\sum_n \chi_n &\leq \zeta && (50)\\
f_{p,n} &\in \mathcal{A}_{p,n} \qquad \forall p, n && (51)\\
f_{p,n} &= \hat{f}_{p,n} \qquad \forall n \in \mathcal{F}_p && (52)\\
\sum_p f_{p,n} &\leq 1 \qquad \forall n && (53)
\end{aligned}$$

Constraint 47 limits the cumulative change in fractional assignment values from previous *micro* epoch. Constraint 48 performs the same function on a per PE basis. Constraint 49 effectively defines χ_n as an indicator variable. It forces χ_n to be 1 if there exists $1 \leq p \leq P$ such that $f_{p,n} \neq \hat{f}_{p,n}$, and either 0 or 1 otherwise. But then constraint 50, which limits the number of PNs that may be affected during the current *micro* epoch, ensures that χ_n will take on the value 0 if the PN is untouched and the constraint is not slack. Constraint 51 ensures that the individual fractional assignments are acceptable. Constraint 52 allows certain fractional assignments to be fixed to their values during the previous *micro* epoch. Finally, constraint 53 ensures that PNs do not get overloaded. If an input data term (such as δ, δ_p or ζ) is infinite, there is no limit associated with the relevant constraint (47, 48 and 50, respectively), which means that the constraint is not enforced.

5.4.3 *Solution Approach.* This optimization problem is solved heuristically, but using techniques borrowed from the theory of network flows [Nemhauser and Wolsey 1988]. At any stage in our heuristic we can compute for any PE p the total amount $\sum_n f_{p,n} H_n$ of allocated *mips*. Comparing this term to g_p we partition the set of PEs into those which are under-allocated, those which are over-allocated and, finally, those which are properly allocated. Obviously the goal is to get all PEs into the properly allocated state. We attempt to do this essentially via iteration of a doubly nested loop, to be described shortly.

A key concept in the scheme is the construction, use and maintenance of a directed graph $\mathcal{G}$ which is illustrated in Figure 16. Note that the PEs are represented by colors in the figure.

This digraph has three types of nodes, as follows.

—On the left side of the figure the nodes are the under-allocated PEs, ordered from most under-allocated to least under-allocated.

—In the middle of the figure the nodes are the PNs themselves.

—On the right side of the figure the nodes are the over-allocated PEs, ordered from least over-allocated to most over-allocated.

These nodes are annotated by their "sizes". For an under-allocated PE node p the size is the degree $g_p - \sum_n f_{p,n} H_n$ to which the PE is under-allocated. For a PN n the size is its processing power H_n. For an over-allocated PE node p the size is the degree $(\sum_n f_{p,n} H_n) - g_p$ to which the PE is over-allocated.

Similarly, there are three types of directed arcs.

—There is a directed arc from an under-allocated PE p to a PN n provided

(1) $v_{p,n} = 1$. That is, n is a candidate node for PE p.

(2) $f_{p,n} < 1$. That is, there is some fractional allocation increase that is possible for PE p on PN n.

(3) $n \notin \mathcal{F}_p$. That is, PE p is not fixed on PN n.

—There is a directed arc from PN n_1 to PN n_2 for a particular PE p provided

(1) $v_{p,n_1} = v_{p,n_2} = 1$. That is, both PNs n_1 and n_2 are candidate nodes for PE p.

(2) $f_{p,n_1} > 0$. That is, there is some fractional allocation for PE p that can be moved from PN n_1.

(3) $f_{p,n_2} < 1$. That is, there is some fractional allocation for PE p that can be moved to PN n_2.

—There is a directed arc from a PN n to an over-allocated PE p provided

(1) $v_{p,n} = 1$. That is, n is a candidate node for PE p.

(2) $f_{p,n} > 0$. That is, there is some fractional allocation decrease that is possible for PE p on PN n.

(3) $n \notin \mathcal{F}_p$. That is, PE p is not fixed on PN n.

These directed arcs are annotated by their "widths". Specifically, a directed arc from an under-allocated PE p to a PN n has width $\min(g_p - \sum_n f_{p,n} H_n, (1 - f_{p,n}) H_n)$. This is the maximum amount of fractional allocation for PE p that can be moved to PN n. A directed arc for PE p from PN n_1 to PN n_2 has width $\min(f_{p,n_1} H_1, (1 - f_{p,n_2}) H_2)$. This is the maximum amount of fractional allocation for PE p that can be moved from PN n_1 to PN n_2. A directed arc from PN n to over-allocated PE p has width $\min((\sum_n f_{p,n} H_n) - g_p, f_{p,n} H_n)$. This is the maximum amount of fractional allocation for PE p that can be moved from PN n.

Consider an arbitrary path $(p_1, n_1, ..., n_k, p_2)$ in the directed graph $\mathcal{G}$ from an under-allocated PE p_1 to an over-allocated PE p_2. All interior terms in this path will be PNs. If we move a given amount f of flow along this path the effect will be to reduce the under-allocation of PE p_1 and to simultaneously reduce the over-allocation of PE p_2 by f. The load on the interior PNs remains constant, though the composition of the load changes: For some PE p_i the fractional allocation will increase by f, but for some other PE p_j the fractional allocation will decrease by

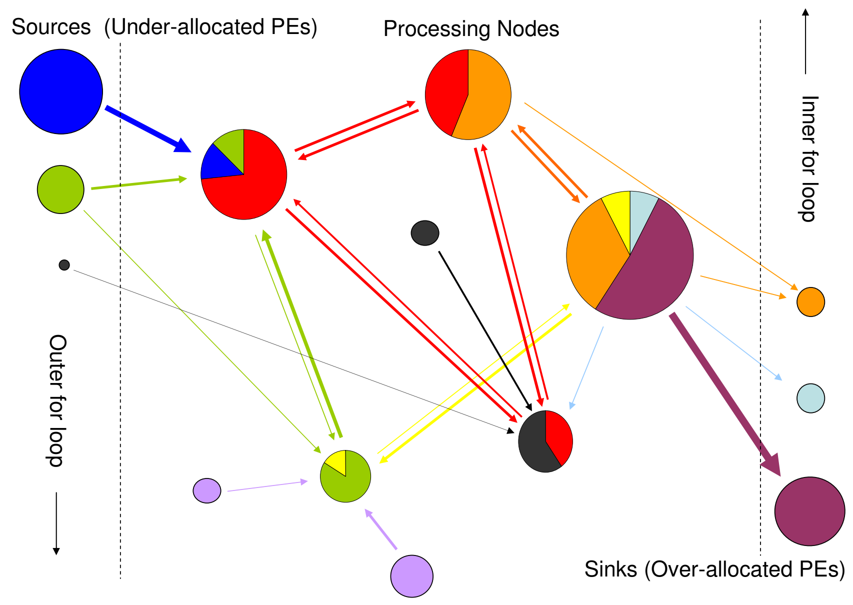


Fig. 16. *microW*: Network Flow Digraph

the same amount. The objective function will thus be reduced by $2f$ if we make this flow push.

How should we choose f? The obvious answer is as large as possible, which would suggest picking f to be the minimum width of the directed arcs along the path. But we also have to respect the constraints. The idea is to check each of the relevant constraints in turn, possibly shrinking f as we proceed. If we reduce f to 0 at any point in the process we will regard the push of flow along this path as a failure. If on the other hand the revised f remains positive at the completion of our constraint checks we will regard the push as a success. Constraint 47 can be looked at as a simple problem of finding the maximum γf for which the hypothetical constraint remains intact. There will be k summands of the form $|f_{p,n} + \gamma f - \hat{f}_{p,n}|$ and k summands of the form $|f_{p,n} - \gamma f - \hat{f}_{p,n}|$. ($\gamma$ may or may not be forced to 0 if the constraint was previously tight.) Then f is replaced by γf, and a similar test is performed for Constraint 48. Constraints 49 and 50 can be handled by recomputing and checking the number of hypothetically touched PNs. If the number exceeds χ we have a failure. Constraint 51 can be handled in a manner similar to Constraints 47 and 48. Constraints 52 and 53 are not relevant by construction.

It remains to give an overview of the entire flow pushing process: This is performed as a doubly nested loop. The outer loop is performed on the under-allocated PEs, from most under-allocated to least under-allocated. (Line 6 of the pseudo-

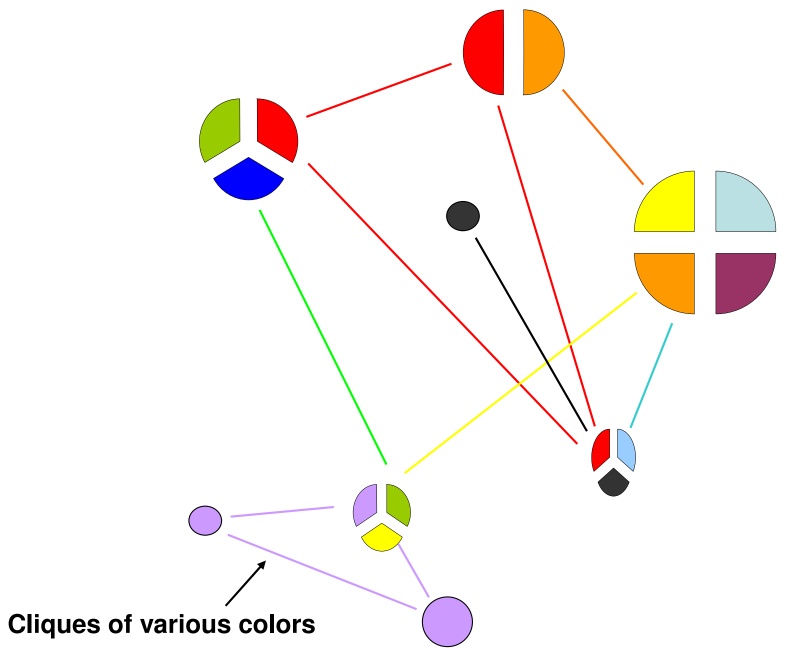


Fig. 17. *microQ*: Surrogate Graph in macroW

code assumes there are U under-allocated PEs.) The inner loop is performed on the over-allocated PEs, from most over-allocated to least over-allocated. (Line 7 of the pseudo-code assumes there are P total PEs and O over-allocated PEs.) The path chosen between under-allocated PE p_1 to over-allocated PE p_2 is determined via a shortest path algorithm [Ahuja et al. 1993]. (The idea here is to affect as few PEs and PNs as possible.) 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 $\mathcal{G}$. 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. (The latter condition is not noted in the "pure" pseudo-code.)

Although we will not go into details, the scheme above can easily be modified to handle cases where the PNs are not fully utilized. One simply adds a dummy PE corresponding to the unutilized portions of each PN.

The *microW* scheme can also be randomized by picking arbitrarily amongst shortest path ties. In practice this has limited utility. We also comment that *macroW* could, in principal, consider the (undirected) graph $\tilde{\mathcal{G}}$ defined as follows. The nodes are the PNs themselves. There is an arc between PN n_1 and PN n_2 for a particular PE p provided $v_{p,n_1} = v_{p,n_2} = 1$. That is, both PNs n_1 and n_2 are candidate nodes for PE p. Note that this condition mimics the first interior condition for the directed graph $\mathcal{G}$. The idea is that $\tilde{G}$ serves as something of a *macroW* surrogate for $\mathcal{G}$. See Figure 17 and compare it with Figure 16. It is "likely" that $\tilde{\mathcal{G}}$ will be highly connected if $\mathcal{G}$ is. So it would be a reasonable *SODA* strategy to add a heuristic to the end of *macroW* which improves the connectivity of $\tilde{\mathcal{G}}$. We have not yet implemented this, however. (Improving connectivity turns out to be a difficult optimization problem.)

6. OBTAINING KEY INPUT DATA

Clearly *SODA* is a sophisticated scheduler with many capabilities. But equally clearly it depends critically on good quality input data. In this section we will describe three infrastructure components. The components are known as the *Resource Function Learner (RFL)*, the *Rank Manager (RM)* and the *Weight Manager*. They supply *SODA* with resource functions, ranks for the jobs, and weights for the relevant streams, respectively. They are intentionally designed to be *separate* from the rest of the *SODA* scheduler, and modular: One can remove one *RM*, for example, and replace it by another. Such a change may significantly impact the scheduling in *System S* without any changes to the scheduler itself.

6.1 Resource Function Learner (RFL)

The *RFL* component is primarily responsible for managing *RF*s and providing the right *RF* to *SODA* (*macroQ*) as needed. Managing the *RF*s is driven by how the PEs are used. A PE from a job may be resubmitted at a later time, either as part of the same job or a different job. Users may share PEs (for example, a classifier) in their own applications. The same PE may run in the system under a different set of circumstances, such as different parameterizations or context (that is, with different upstream or downstream PEs). Since even PEs that have never been run before do need to be scheduled, it is very helpful for good resource allocation to have *some* initial estimate of the PE's resource usage.

In order to construct *RF*s for a PE, we take an empirical approach: a specific model structure is chosen and actual system observations are used to learn or calibrate the model parameters. There are potentially many choices for the right functional form for an *RF*. Our experiences with both parametric and non-parametric forms are described in an earlier work [Hildrum et al. 2009]. For the rate-based RFs in this paper, we used a simple mathematical model for the *RF*s and trained that model with data collected from prior runs. For *transform* PEs (with both input and output streams), we employ the model $r_o = \min(Ar_i m, Br_i)$, where r_o is the output rate, r_i is the input rate, m is *mips*, and A and B are constants picked to best fit the data. The intuition behind this choice is that there are two regimes. In the first regime, where the allocated resources are insufficient, the output rate is constrained by the *mips* themselves, and more *mips* leads to higher output rates. This is the first term. The second regime occurs when the *mips* are plentiful. In this case, the output rate is constrained by the input rate. This is the second term. This surprisingly simple model has performed well in our experiments, actually better than more complex models based on regression trees. At its core, we have a classic online machine learning problem, and we are actively exploring the use of more advanced techniques that may yield more accurate models. We note that *macroQ* "expects" that the *RF*s will initially be strictly increasing. (Once they stop increasing, *SODA* will stop further exploration.)

Over time, the *RFL* seeks to use system measurements to obtain and evolve a set of *RF*s that can be as accurate as possible, while also making maximal use of any metrics collected for PEs running in the system. This leads to the following questions:

—How do we identify a PE so we can associate an *RF* with it?

—What should we do about PEs when they are seen by the system for the first time, and there is no empirical data available for them yet? What sensible initial *RF* can be provided?

—How can we identify whether previously collected data (or a model built from it) is applicable for a PE that runs in a slightly different environment than where the data is collected? The environment may include factors such as PE configuration (via, for example, command-line arguments), or systems issues, such as processing node architecture. This issue also arises when the same PE is reused in multiple applications.

—In a similar vein, how should *RF*s be shared between instances of a PE, and how can observing one instance of a PE yield clues about the resource usage of another, slightly different instance?

—How should metrics be stored and used to build and update the PE *RF*s?

From a functional perspective, the *RFL* performs the following two tasks: (a) given a specific instance of a PE in a specific job that is submitted to the system, choose an *RF* to be used by the scheduler, and (b) given an observation of the resource usage of a specific PE instance, identify which *RF*s should be updated, and update them.

To facilitate the model management, we identify each PE based on a multi-part *signature*. The *RFL* learns and maintains an *RF* for each signature. For a specific PE, the scheduler uses the signature to decide which *RF* should be used. In our system, we use the following four parts of the signature, in order of increasing specificity:

—*PE type*: The first part of the signature may be a source, a transform or a sink. (A *source* has no input streams, while a *sink* has no output streams.) In the future, it is possible to envision a much finer granularity of PE classification into types.

—*Executable*: The second part of the signature is the most general, consisting of an MD5 hash of the PE's executable. If the PE has been run before in any context, a learned model will be available. This will likely be better than a default model based only on PE type.

—*Arguments*: The third part is a MD5 hash of the arguments. A PE's command-line arguments may alter its behavior, so this attempts to capture the dependency. Using an MD5 hash means that there is no need to understand the structure of the arguments (i.e., knowing that a particular argument is window size).

—*Flowspec*: The fourth part is a representation of how the PE is connected to its upstream PEs, which is known as the *flow specification* or *flowspec*. In *System S*, it represents the most specific attribute of a particular instance of a PE that is connected in a specific way to other PEs.

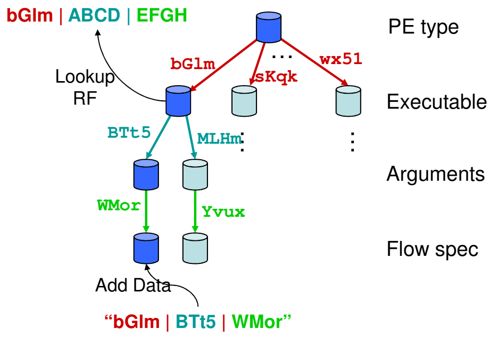


Fig. 18. Hierarchical PE signature management

The signature captures key attributes of the PE which are knowable at *job submission time*. This allows the scheduler to choose an *RF* before the PE even begins execution. After execution, the PE *RF* may be refined further based on ongoing observations, but the initial lookup addresses the bootstrap problem.

The collection of signatures constitutes a hierarchical organization, as depicted in Figure 18. There is an *RF* for each node in this hierarchy. The node labeled '`bGlm`' represents all PEs whose executable hashes to that string. Its two child nodes represent that PE executable being run with two different command-line arguments. Thus each node in the tree represents a generalization of all its children. To look up the *RF* for a specific PE, we find the most specific node which matches the PE signature. In the figure, the lookup for '`bGlm | ABCD | EFGH`' stops at the PE level node '`bGlm`' because the rest of the signature represents entries that are not (yet) in the database. Thus, if specific information about the PE in that context was available, that is the information that would be used. If no specific information was available, the system uses information based only on the executable. In the case of a brand new PE that has never been run before in any configuration, the system uses either (a) a generalized *RF* based on the root node (PE type), or (b) a default type-based *RF* which is hand-populated based on calibrations from earlier *System S* applications.

Analogously, when a new data point is collected for a PE, we update the model at each of these three levels, starting at the most specific node and propagating up the tree. Thus, an observation $< r_j^I, m_p, r_k^O >$ for PE p with input port j and output port k whose signature is '`bGlm | BTI5 | WMor`' can update not only the model at the most specific node, but at every generalization above it in the path to the root, namely '`bGlm | BTI5`' and '`bGlm`'. Thus, a leaf model reflects observations about the PE in the most specific context, while an internal tree node generalizes data across the sub-tree rooted at that node. When a known PE is encountered in a new context, the *RFL* can obtain some information about that PE's behavior by using this generalized information.

To highlight the possibility of reuse in actual applications, Table VI shows the number of unique nodes at each level in the tree for three separate *System S* applications. (DAC [Wu et al. 2007], already mentioned regarding Figure 2, is an insurance claims fraud detection and alerting system. SKA [Biem 2008] is a ra-

Application	**Level** Executable	Arguments	Flowspec
DAC	40	64	81
SKA	27	102	102
VWAP	25	365	365

Table VI. Signature counts by level

dio astronomy application which reconstructs images from data received by radio telescope antennas. VWAP [Andrade et al. 2009] represents a financial markets scenario in which real-time quotes are processed to detect bargains and trading opportunities.) SKA and VWAP (and most *SPADE*-based applications) do not use flowspecs on their streams, so each argument-level node has only one child node (the 'null' flowspec). However, these applications contain several replicas of the same PE, executing with different arguments. This indicates that maintaining the executable-level information is likely to be useful if we encounter a new PE with a different set of arguments than seen before.

6.2 Rank Manager

We have noted that the *SODA* macroQ component decides, among other things, whether to admit jobs or not subject to the rank-legality constraints. So rank is a highly important input metric. It is the role of the *Rank Manager (RM)* to supply the rank information for jobs to *SODA*. We have designed two different *RM*s, depending on the requirements of the system: One is called the *Budget Driven Rank Manager (BDRM)* and the other is called the *Merit Driven Rank Manager (MDRM)*. We will describe both of these briefly. The idea behind multiple *RM*s is that while rank is critical data, different *System S* customers will have different perspectives about how to compute the various job ranks. The appropriate solution is therefore to provide an easy interface by which different customers can plug in their favorite *RM* choices.

6.2.1 *Budget Driven Rank Manager.* The *BDRM* uses a notion of *budget* to determine ranks, employing a temporal fairness policy. The *BDRM* actually gives "users" submitting jobs some control over the ranks, but full control would not be workable: Savvy users would quickly adopt a greedy approach and assign all their jobs a rank of 1. So the *BDRM* does not allow users to submit actual ranks, just the relative ordering of all the jobs they are submitting.

To begin with, the *BDRM* employs a separate process to create a budget allocation for each of the users over a relatively long time interval. This interval might be months or quarters. This is done in a top-down tree-like fashion, presumably following the organizational structure. The users correspond to the leaves in this tree. The budget allocation process may be political in nature, just as monetary budget allocation schemes typically are. (There is no magic here. The *BDRM* cannot compare apples and oranges. People will do a better job of that.) In the end, each user receives a *rank budget* for the time interval. The *BDRM*'s job is to deplete that long term budget in a "fair" manner, and it employs a simple greedy scheme to do so. Fairness is based on the apportionment of processor resources used by *SODA* during overloaded periods. This requires a bookkeeping mechanism to keep track of resource usage. Ideally, every user's budget should deplete at the same rate, and the *BDRM* attempts to enforce this.

Consider the amount of remaining user budgets. The *BDRM* first marks all required jobs. These jobs will be done, so the rank Π for these jobs is set to 1. They are removed from consideration and the remaining user budgets are adjusted accordingly. Then the *BDRM* sets $\Pi = 2$. Among the remaining jobs it compares the most preferred jobs per user and picks the one(s) with the largest remaining user budgets. The *BDRM* sets these to have rank Π, removes them from the list, *hypothetically* recomputes the remaining user budgets and increment Π by 1. This process is repeated until the list is exhausted.

After running *SODA* for an actual epoch the *actual* remaining user budgets are recomputed and the process begins again. (We don't deplete budgets during underloaded epochs: Every jobs gets in, and it is a good idea to encourage users to submit jobs during underloaded epochs.)

We optionally allow the ranks computed by the *BDRM* to be visible and overridable by the *System S* site administrator.

6.2.2 *Merit Driven Rank Manager.* The *MDRM* is an inherently simpler design than that of the *BDRM*. The rank of a job in the *MDRM* will be calculated based on its *merit*, which is the weighted sum of one or more terms. The exact number and definitions of the summands is outside the scope of *System S*. We list two examples of *types* of terms.

—There are so-called *external priority* terms, terms which a *System S* customer may already be comfortable using. They are typically relatively static.

—There are so-called *precomputed analytic* terms, which are typically dynamic and can be computed by independent analytic modules in more or less real time. One example might be a *failure factor*, that is, an estimate of how likely the job is to fail. Lower would mean less likely to fail. Another example might be a *deadline factor*. Lower would mean closer to a (meetable) deadline. The design of these analytic modules are essentially outside the scope of *System S*.

Note that this approach can also automatically incorporate either a full or a partial Dewey Decimal variant, by judicious choice of weights. A scenario in which one term is deemed absolutely more important than a second can be accommodated by picking the first weight much greater than the second.

Next the jobs are partitioned into buckets of fixed size, say B. Thus the jobs with merit less than B will go into the first bucket and assigned a rank of 1; the jobs with merit greater than or equal to B but less than $2B$ will go into the second bucket, assigned a rank of 2, and so on. This bucketing process is designed to deal with the obvious fact that the merit value is a somewhat imprecise measure. Bigger buckets will result in more ties in rank. If B is tiny this will basically cause rank to be based on the ordering of the merit values. But even without bucketizing it is possible to have ties in rank.

We allow the weights to depend on time of day, day of week and so on. We optionally allow the ranks computed by the *MDRM* to be visible and overridable by the *System S* site administrator.

6.3 Weight Manager

Weight is also a key metric of each stream that *SODA* uses to decide which jobs to schedule, and the quantity of resources to allocate to them. The *Weight Manager (WM)* supplies this data to *SODA*. As with the *RM*, we have created a modular design into which various *WM*s can be plugged in. We have designed two different *WM*s, depending as before on the requirements of the system: One is called the *Budget Driven Weight Manager (BDWM)* and the other is called the *Simple Weight Manager (SWM)*.

The *BDWM* is very much like the *BDRM* in design, so we will omit details here. The *SWM* is indeed extremely simple. Weights of 1 are assigned to all terminal PEs streams in job templates. All other streams are given a weight of 0.

The *Weight Override Module* optionally allows the site administrator to revise weights arbitrarily, only enforcing the rule that they must be between 0 and 1. A weight of 0 will definitely force the kind of behavior indicated in the description of Figure 5. Effects of non-zero weights will be more subtle. A weight of 1 will encourage the system to give as many resources as is possible to the PEs upstream of the relevant stream. Longer term plans include a much more elaborate machine learning *WM* based on "discoveries" made during the operation of *System S*.

7. EXPERIMENTS

We present two sets of experiments, focusing on two critical functions of *SODA*: job admission and placement. The first set of experiments illustrate the performance of *macroQ*. The second set of experiments are end-to-end, demonstrating the performance of *SODA* in placing jobs admitted to the *System S* cluster.

7.1 Job admission experiments

In this section we experimentally evaluate *SODA* performance, focusing on the functions of job admission and resource allocation. The tradeoffs between the two can be quite subtle, as we will show.

First we will describe the experimental setup. The largest system installation we consider has 100 PNs with a rating of 11,000 *mips* each. In the experiments we examine the effect of removing 5 PNs (and thus 5% of the processing power) from the system at a time. The jobs presented to *macroQ* always remain the same: There are 19 jobs (labeled A through S), consisting of 7 required jobs of rank 1, 6 optional jobs of rank 2, and 6 optional jobs of rank 3. The jobs are not interconnected, so rank and revised rank are identical for each job. The experiments are designed so that at 100 PNs the jobs will nearly (but not quite) use all the resources in the system when each is allocated their maximum useful resources. This occurs, as per the previous section, when each of the component importance functions becomes flat as a function of allocated resources. In fact, when *macroQ* is run on the full 100 PNs all 19 jobs are admitted and the average utilization of the processors is 97%.

Figure 19 shows the number of jobs admitted by rank as the number of PNs decreases in 5% increments from 100 PNs to 25 PNs. At 95% all jobs are still admitted, though the processor utilization now is 100%. From there on the utilization remains at 100%, as one would expect based on *macroQ*'s design: The system is

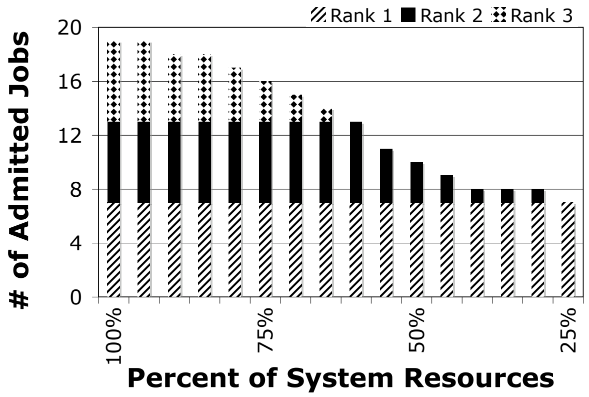


Fig. 19. Admitted Jobs by Rank

%	Rank 1							Rank 2						Rank 3					
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
100	786	882	462	540	558	318	336	786	870	462	702	264	366	786	870	516	558	294	336
95	780	840	462	540	558	318	336	780	780	462	696	264	366	780	780	516	558	294	336
90	786	792	462	540	558	318	336	780	780	462	612	264	366	696	780	516	558	294	
85	696	792	372	540	468	318	318	696	780	372	612	264	366	696	780	516	468	294	
80	690	786	366	534	462	312	312	690	780	366	606	258	366	696	774	510		288	
75	684	780	366	534	354	270	312	626	768	366	606	258	360	684	774	510			
70	690	786	366	534	444	312	312	690	774	366	606	258	360	690		510			
65	696	792	372	540	468	318	336	696	780	390	612	264	366			516			
60	696	792	372	540	468	318	324	696	780	372	612	264	366						
55	696	792	462	540	552	318	336	696	780		612	264							
50	690	786	366	534	426	312	312	692	774		606								
45	690	792	366	534	462	318	312	696	780										
40	696	792	462	540	558	318	336	696											
35	626	710	354	494	426	300	314	626											
30	510	558	344	426	354	272	292	544											
25	510	558	344	426	354	272	286												

Table VII. *mips* x 100

overloaded. At 90% 1 job of rank 3 is rejected, and all of the rank 3 jobs are gone by the 60 PN level. But rank 1 and 2 jobs remain during the 65% to 100% range. In other words, the rank waterline is 3. In the 30% to 60% range the rank waterline is 2. All rank 1 jobs are admitted, but more and more rank 2 jobs are rejected as the processing power decreases. At 25 PNs only the required rank 1 jobs are admitted. The system is fully stressed at this point, and a *macroQ* run at 20% of the PNs would not find a feasible solution: There would not be sufficient processing power to admit all of the required jobs even at their minimum acceptable resource allocations.

Figure 20 shows the contribution to overall importance by rank as the number of PNs decreases from 100 to 25. The importance is decreasing as a function of system resources, as should be the case. But between 100 and 85 PNs the importance curve is actually quite flat: The component importance curves turn out to be concave or close to concave, and there are sufficient resources available so that the solution lies near the flat part of each curve. Job S is rejected at 85% and 90%. But its importance is low and its resource requirements is high. One

%	Rank 1							Rank 2						Rank 3					
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
100	402	389	136	262	222	279	13	402	389	136	261	250	291	402	390	304	222	279	13
95	402	388	136	262	222	279	13	402	388	136	261	250	291	402	388	304	222	279	13
90	402	388	136	262	222	279	13	402	388	136	259	250	291	400	388	304	222	279	
85	400	388	129	262	216	279	10	400	388	129	259	250	291	400	388	304	216	279	
80	395	383	126	260	212	275	9	395	388	126	257	245	291	400	383	301		275	
75	389	377	126	260	215	274	9	389	377	126	257	245	286	389	383	301			
70	395	383	126	260	205	275	9	395	383	126	257	245	286	395		301			
65	400	388	129	262	216	279	13	400	388	129	259	250	291			304			
60	400	388	129	262	216	279	11	400	388	129	259	250	291						
55	400	388	136	262	221	279	13	400	388		259	250							
50	395	383	126	260	178	275	9	395	383		257								
45	395	388	126	260	212	279	9	400	388										
40	400	388	136	262	222	279	13	400											
35	359	355	125	259	178	273	10	389											
30	301	333	116	253	170	262	6	380											
25	301	333	116	253	170	262	5												

Table VIII. Importance

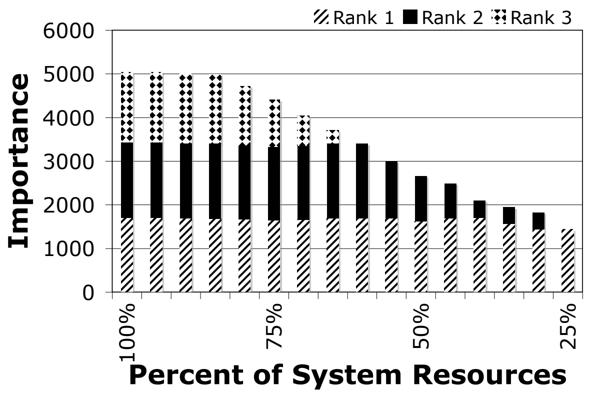


Fig. 20. Importance by Rank

can see this in Tables VII and VIII. The first table shows the allocated resources (in hundreds of *mips*) by individual jobs as the number of PNs decreases. The second table shows the corresponding importance values. Job S contributes an importance of only 13 at 33600 *mips*, so it is clearly highly expendable, and being of rank 3 it is jettisoned as soon as the offered load exceeds available resources. The effect on overall importance is minimal. The only other job with a poor ratio of importance to resources is job G. Job G is a twin of Job S, but it is required and *macroQ* cannot reject it. Observe in Figure 20 that importance does start to decrease linearly from 80 down through 25 PNs. The available *mips* dictate that the solution to the component resource allocation problems occur on the steeper portion of each importance curve.

Examine Table VII in the 3 ranges of resource allocations for which the admitted jobs remains identical. (The 100% and 95% rows have this property. So do the 90% and 85% rows. And finally, so do the 40%, 35% and 30% rows.) If the component importance curves are concave for each admitted job in these ranges, the separable resource allocation problems in *macroQ* would solve where the first differences (effectively the derivatives) at each job would be as close to equal as possible. And that would, in turn, imply that the resource allocations for each job would be monotone non-increasing as the number of processors decrease. In fact, this is the case in each of the three ranges, as an examination of the relevant columns shows.

Overall, however, the allocated resources for each job will not be monotone as the number of processors decrease. Consider, for example, the column for Job C in Table VII. The *mips* allocated are high in the 85% to 100% range, because the system is not heavily overloaded. As the number of processors decrease the *mips* allocated to Job C exhibits somewhat oscillatory behavior, decreasing through 70%, then increasing back to its maximum useful allocation at 55%, and so on. This behavior is primarily due to the changes in admission of the *other* jobs. As jobs get rejected the allocations they would have received become available, and *macroQ* will distribute these to the jobs that remain. (A secondary reason for the lack of monotonicity is the slight deviations from concavity.) At the 25 and 30 processor levels the system is truly stressed and there the job is given minimum acceptable *mips* allocations.

We have focused on the *macroQ* problems of job admission and resource allocation in these experiments. We present extensive experimental analyses of the overall performance of *SODA* next.

7.2 *SODA* Experiments

7.2.1 *Methodology.* In this section we describe quantitative and qualitative experiments for analyzing the performance of *SODA* and the quality of the *RF*s generated. We present experiments on three applications: SrcSink, LSD [Amini et al. 2006], and DAC [Wu et al. 2007]). SrcSink is a toy application with two PEs; it has been included in the test to more easily illustrate some key points. 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. 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.

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

—**Random (RAND)**: PEs are assigned to PNs uniformly at random. In expectation, each PN hosts the same number of PEs, but in fact, the number of PEs hosted by a PN may vary quite a bit.

—**Round-robin (RR)**: PEs are processed sequentially and each PE is assigned to a PN with the minimum PEs assigned so far. This is a very naive load balancing of PEs across the PNs.

—**Expert (EXP)**: The application developers for LSD and DAC decide on the number of PNs and an allocation of PEs to PNs based on both their knowledge of the application as well as several trial-and-error runs where all PEs are resource matched to specific PNs. 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.

7.2.2 *Metrics.* We use a variety of metrics to measure the quality of a solution. Recall that each job consists of at least one source PE and one sink PE. A source PE has no input streams internal to the system. It obtains data from primal streams. A sink PE, on the other hand, produces no output streams internal to the system. However, its output maybe be written to disk, or sent to a query.

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. This measurement is made at the output of the source PEs, and compared with the predicted rate (for *SODA* runs).

—*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. This is compared against the predicted importance (for *SODA* runs).

—*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 PN divided by the total traffic. The higher this quantity, the better, since PEs which share a stream should be put on the same PN (or nearby) to minimize network utilization. We can measure this at two levels. The *stream level* is the fraction of parent-child pairs on same PN and the *traffic level* is the fraction of intra-PN traffic.

—*Execution time*: The total time taken for the execution of the scheduler. For *SODA*, this corresponds to the sum of the time taken by the four separate *SODA* modules.

—*Utilization*: The total amount of CPU utilization, counting all the PEs. We compare this against the predicted utilization (for *SODA* runs).

—*Direct prediction accuracy*: The average prediction error of the *RF*s, for each stream in the submitted jobs.

Most of these are placement metrics and are useful to evaluate *SODA* (and indirectly the *RFL*). The direct prediction accuracy is an abstract metric for comparing various *RFs* against each other, which is described by the final metric, specifically designed to evaluate the *RFL*.

7.2.3 *Input Parameters.* Now we describe all the knobs (input parameters) that were changed in the experiments, to quantitatively measure *SODA*'s performance. 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 PNs 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. For each application (SrcSink, LSD, DAC), we perform three runs for each combination of scheduler and node pool size, and analyze the average across these runs.

We also describe two *SODA* configuration parameters that can be configured from the command line (or during run time). For our experiments, these were set as described below:

—*Exact (CPLEX macroW) vs heuristic (miniW)*: We can measure the impact of employing the exact optimization scheme of *macroW* by turning CPLEX on and off. Based on the initial experiments, we decided to turn off CPLEX.

—*Network information*: We can run *SODA* with and without network information. Based on the initial experiments, we always ran with network information available.

—*Defaults RFs vs Trained RFs*: We can run *SODA* with parametric *RFs* of the same type as the defaults, but have been fitted/trained to data obtained from the best EXP run. (Default *RF*s are ones that are used if *SODA* does not recognize the PE signature.) For these experiments, *SODA* always ran with trained *RFs*.

7.2.4 *Data Gathered.* All of the runs are incorporated within a standard unit-test framework, so that the process is controlled, organized and repeatable. These metrics are computed from the raw system metrics such as CPU usage per PE and traffic consumed and produced by each PE. For analyzing performance, this data is gathered and converted into the information we need: the total *mips* used by the system; the *mips* used, the bytes prepared, and the bytes actually sent for a particular subset of PEs; the total PN-to-PN traffic, and the total traffic that is intra-PN; and the accuracy of predictions generated from the *RF*s.

7.2.5 *Results.* First, we present the time taken by *SODA* to solve these instances. For the small SrcSink test, the time is 26 seconds, most of which is taken by *macroW* since it continues to compute until its time limit of 25 seconds is reached. Otherwise, these instances can be solved in less than 3 seconds. On the other hand, for the much larger LSD job, *SODA* takes 42 seconds. For DAC, which is of intermediate size, *SODA* takes 33 seconds.

Next we describe the effect of learning *RF*s from actual traffic, as opposed to using default *RF*s. This is a direct measure of the performance of the *RF*s. This comparison is best illustrated using prediction accuracy. We present results for this metric for SrcSink. (We repeated the experiments for a variety of configurations, presented here.) As one can see in Figure 21, accurate *RF*s make a huge difference to *SODA*'s accuracy in predicting *mips* usage and traffic; the lower the error the better. The average error drops from 27 to 7 percent. In this chart, “new” corresponds to the learned *RF*s.

Finally, we analyze the performance of *SODA* in scheduling LSD and DAC. The carefully constructed EXP placements use 82 PNs for LSD and 30 PNs for DAC. *SODA* uses far fewer PNs yet achieves a higher quality placement than EXP. In particular, *SODA* performs favorably with as few as 30 PNs for LSD and 9 PNs for DAC, 36% and 30% of the number of PNs used in the expert placement, respectively. To compare with these, we also present the results for RAND and RR for two

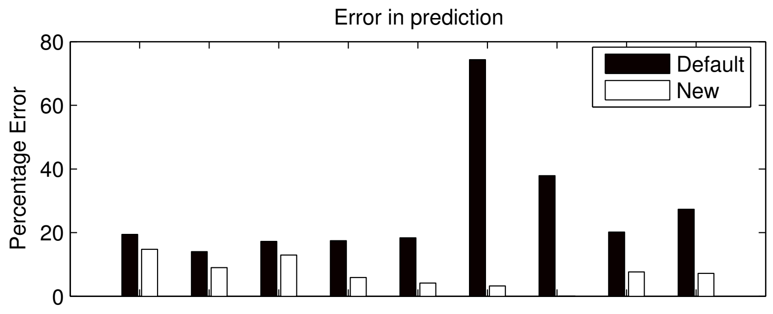


Fig. 21.

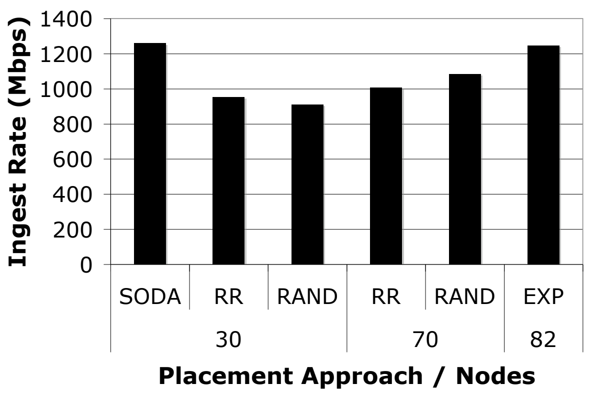
scenarios: 30 and 70 PNs for LSD, and 9 and 29 PNs for DAC. These allow us to compare their performance with *SODA*'s placement at one end of the spectrum (less PNs), and with EXP at the other end (more PNs).

Figure 22 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 PNs (30) for LSD. For DAC, *SODA* outperforms EXP by over 50% with just 9 PNs. 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 PNs for LSD, RR is able to ingest 25% less traffic than EXP, and with 29 PNs for DAC, RR is able to ingest 15% less traffic than EXP.

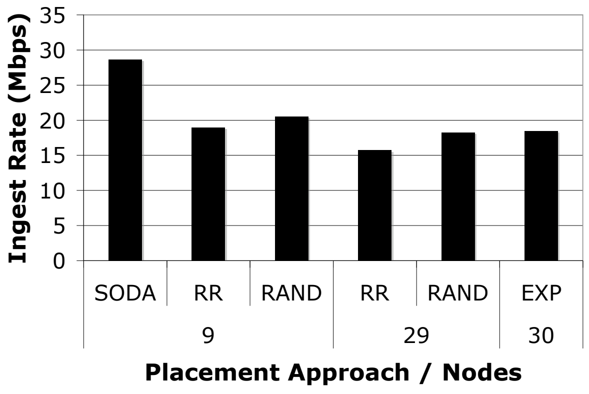
One of the metrics that *SODA* tries to maximize is the importance. Figure 23 presents the importance of DAC, as optimized by *SODA* in *macroQ*; recall that in our case this corresponds to the net traffic flowing into the sink PEs. Here, we see that *SODA* matches the performance of EXP in spite of using a third of the PNs. On the other hand, RR and RAND perform more than 10% worse than EXP, even when using 29 PNs. In particular, RR achieves only 84% of the traffic rates at the sinks attained by EXP and *SODA*.

Another goal of *SODA* is to ensure the network is not overloaded. Figure 24 plots the stream affinity, showing the fraction of traffic that is sent on streams that have both source and destination PEs on the same PN; higher is better. From the figure, we see that *SODA* placements fare much better than the hand-placement on this metric. For instance, with 30 PNs, the *SODA* placement for LSD sends less than 30% of the traffic over the network (over 70% on the same PN); compared to 66% for EXP with 82 PNs. In addition to helping reduce network congestion, this also contributes to the stronger throughput results obtained by *SODA*, since the overhead of sending to a PE on the same PN is lower. Naturally, RAND and RR fare poorly on this metric since they do not use stream information in their placement algorithm. Thus, they do not utilize the network wisely and in fact are susceptible to exceeding the network capacity. For DAC, with 9 PNs, *SODA* places 20% of the traffic on the same PN, resulting in significantly larger ingest rate than any of the other schemes.

In all our experiments, we observe that *SODA* requires significantly fewer PNs, and utilizes much less network capacity to perform as well, if not better than, a care-



(a) LSD



(b) DAC

Fig. 22. Ingest rate: LSD and DAC

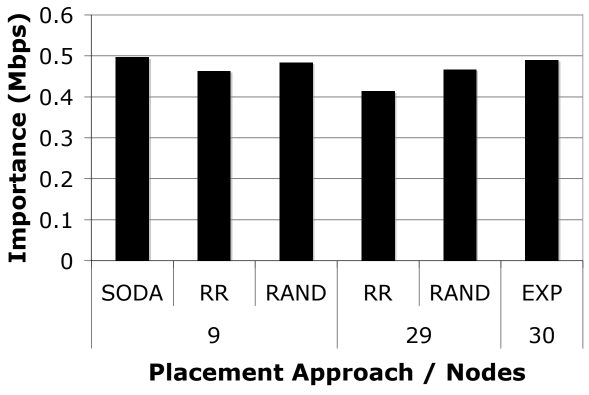
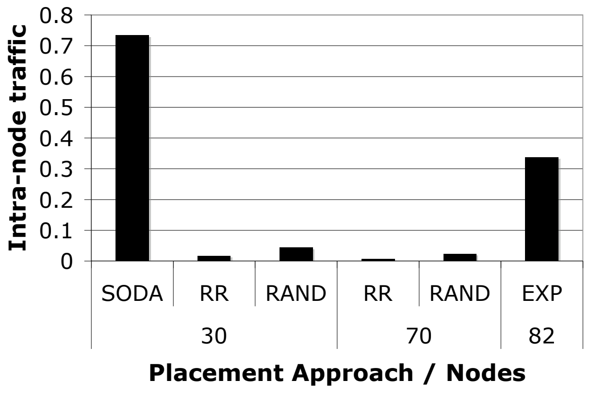
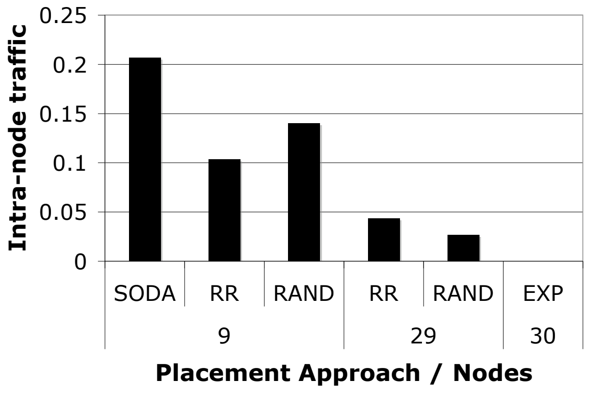


Fig. 23. Importance: DAC



(a) LSD



(b) DAC

Fig. 24. Stream Affinity: LSD and DAC

fully 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.

8. RELATED WORK

Stream processing systems have been an active area of research in recent years. Example systems include Borealis [Abadi et al. 2005], TelegraphCQ [Chandrasekaran et al. 2003], STREAM [Arasu et al. 2003], Aurora and Medusa[Zdonik et al. 2003]. 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 PN, 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 [Tatbul et al. 2007] 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 PN, and so could *increase* load at PNs 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. [Xing et al. 2006; Xing et al. 2005] 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. [Pietzuch et al. 2006] 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. [Lakshmanan and Strom 2008] also addresses scheduling to minimize latency.

The STREAM project [Motwani et al. 2003] 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. [Xia et al. 2007] address admission control problem in a hypothetical stream processing systems. Their model assumes a linear processing graph. In other words, input stream is processed, successively, by a series of operators. Thus, no operator takes input from more than one source stream.

9. CONCLUSIONS

In this paper we have described *SODA*, a new scheduler for very large-scale distributed stream processing applications. This scheduler is a major component in the *System S* project. We believe *SODA* is practical, novel, and effective. This paper is intended to be as complete a description of *SODA* as we can practically provide, focussing on the mathematical details. We have provided an introduction to *System S*, an introduction to the scheduling problem, an overview of the solution, descriptions of the input data and the modules that determine these data, and an experimental analysis.

There are many interesting *SODA* variants and related tools that space prevents us from describing completely. We list three of these here.

—While *System S* is stream-oriented there will always be more traditional workloads which coexist with the streaming applications. Traditional jobs have end times in addition to start times. We therefore refer to them as *non-continual (NC)* jobs. We have designed a companion scheduler for these workloads, known as *NC-SODA*. The streaming and non-continual problems contain surprising analogies.

(1) Instead of PEs we consider tasks with finite execution times.
(2) In the streaming problem jobs consist of PEs interconnected with streams into a dataflow graph. In the non-continual problem we consider jobs consisting of tasks which are interconnected via *precedence constraints*.
(3) In the streaming problem we have value functions. In the non-continual problem we have penalty functions (such as response times, completion times, lateness and tardiness) which depend on the completion of the makespan task.
(4) In the streaming problem we have value functions which increase with allocated resources. In the non-continual problem we have execution times which decrease with allocated resources.
(5) Instead of maximizing total importance we will attempt to minimize the total penalties.
(6) Instead of the *RFL* we learn the relationship between task execution times and allocated resources.

For further details see [Wolf et al. 2007].

—Note that communications between PEs in *System S* are long-lived, on the order of minutes or more. So there is a natural affinity between *System S* and circuit switching. *Optical Circuit Switches (OCS)* provide the benefits of circuit switching, using mirrors to allow the configuration of the network to be changed as needed. The overhead of making these changes is sufficiently low to allow such changes on an epoch by epoch basis. An OCS prototype of *System S* has been built, and an enhanced scheduler, *OCS-SODA*, makes network topology configuration decisions in addition to its usual optimizations. For more details see [Schares et al. 2009].

—We have noted that the *SPADE* compiler can partition operators into PEs for efficiency. We have built an optimizer named *COLA* to help solve this difficult optimization problem. In order to maximize throughput, *COLA* attempts to minimize the processing overhead associated with inter-PE stream traffic while simultaneously balancing load across the PNs. For further details see [Khandekar et al. 2009].

REFERENCES

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., and Zdonik, S. 2005. The design of the Borealis stream processing engine. In *Proceedings of Conference on Innovative Data Systems Research*.

Ahuja, R., Magnanti, T., and Orlin, J. 1993. *Network Flows.* Prentice Hall.

Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Selo, P., Park, Y., and Venkatramani, C. 2006. SPC: A distributed, scalable platform for data mining. In *Proceedings of the Workshop on Data Mining Standards, Services and Platforms.*

Andrade, H., Gedik, B., Wu, K.-L., and Yu, P. S. 2009. Scale-up strategies for processing high-rate data streams in System S. In *ICDE09.*

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., and Widom, J. 2003. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin 26.*

Biem, A. 2008. Imaging for next-generation radio telescopes. Personal communication.

BladeCenters, I. http://www.03.ibm.com/systems/bladecenter.

Blazewicz, J., Ecker, K., Schmidt, G., and Weglarz, J. 1993. *Scheduling in Computer and Manufacturing Systems.* Springer-Verlag.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Raman, V., Reiss, F., and Shah, M. A. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of Conference on Innovative Data Systems Research.*

Coffman, E. 1976. *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons.

Cormen, T., Leiserson, C., and Rivest, R. 1985. *Introduction to Algorithms.* McGraw Hill.

Douglis, F., Palmer, J., Richards, E., Tao, D., Tetzlaff, W., Tracey, J., and Yin, J. 2004. Position: Short object lifetimes require a delete-optimized storage system. In *ACM SIGOPS European Workshop.*

Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M. 2008. SPADE: The System S declarative stream processing engine. ACM.

Hildrum, K., Douglis, F., Wolf, J., Yu, P. S., Fleischer, L., and Katta, A. 2008. Storage optimization for large-scale stream processing systems. *ACM Transactions on Storage 3,* 4.

Hildrum, K., Parekh, S., Rajan, D., Wolf, J., and Wu, K.-L. 2009. Characterizing, constructing and managing resource usage profiles for System S applications: Challenges and experience. In *CIKM09.*

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

ILOG. CPLEX. http://www.ilog.com/products/cplex.

Jacques-Silva, G., Challenger, J., Degenaro, L., Giles, J., and Wagle, R. 2007. Towards autonomic fault recovery in System-S. In *Proceedings of Conference on Autonomic Computing.*

Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., and Venkatramani, C. 2006. 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.*

Khandekar, R., Hildrum, K., Parekh, S., Rajan, D., Wolf, J., Wu, K.-L., Andrade, H., and Gedik, B. 2009. COLA: Optimizing stream processing applications via graph partitioning. In *Proceedings of Middleware Conference.*

Lakshmanan, G. and Strom, R. 2008. Biologically-inspired distributed middleware management for stream processing systems. In *Proceedings of Middleware Conference.*

Motwani, R., Widom, J., Arasu, A., Babcokc, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., and Varma, R. 2003. Query processing, approximation, and resource management in a data stream management system.

Nemhauser, G. L. and Wolsey, L. A. 1988. *Integer and Combinatorial Optimization.* John Wiley and Sons, New York.

Papadimitriou, C. and Steiglitz, K. 1982. *Combinatorial Optimization.* Prentice Hall.

Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M. 2006. Network-aware operator placement for stream-processing systems. IEEE Computer Society, Washington, DC, USA.

Pinedo, M. 1995. *Scheduling: Theory, Algorithms and Systems.* Prentice Hall.

Schares, L., Zhang, X., Wagle, R., Rajan, D., Selo, P., Chang, S.-P., Giles, J., Hildrum, K., Kuchta, D., Wolf, J., and Schenfield, E. 2009. A reconfigurable interconnect fabric with optical circuit switch software optimizer for stream computing systems. In *Optical Fiber Communication Conference.*

Tatbul, N., Çetintemel, U., and Zdonik, S. 2007. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the International Conference on Very Large Data Bases Conference.* 159–170.

Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., and Wu, K.-L. 2009. Job admission and resource allocation in distributed streaming systems. In *Workshop on Job Scheduling Strategies for Parallel Processing, IPDPS.*

WOLF, J., BANSAL, N., HILDRUM, K., PAREKH, S., RAJAN, D., WAGLE, R., WU, K.-L., AND FLEISCHER, L. 2007. A scheduling optimizer for distributed applications: A reference paper. Tech. Rep. 24453, IBM Research Report.

WOLF, J., BANSAL, N., HILDRUM, K., PAREKH, S., RAJAN, D., WAGLE, R., WU, K.-L., AND FLEISCHER, L. 2008. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Proceedings of Middleware Conference*.

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., AND WANG, H. 2007. 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*.

XIA, C. H., TOWSLEY, D., AND ZHANG, C. 2007. Distributed resource management and admission control of stream processing systems with max utility.

XING, Y., HWANG, J.-H., ÇETINTEMEL, U., AND ZDONIK, S. 2006. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the International Conference on Very Large Data Bases Conference*. VLDB Endowment, 775–786.

XING, Y., ZDONIK, S., AND HWANG, J.-H. 2005. Dynamic load distribution in the Borealis stream processor. IEEE Computer Society, Washington, DC, USA, 791–802.

ZDONIK, S., STONEBRAKER, M., CHERNIACK, M., CETINTEMEL, U., BALAZINSKA, M., AND BALAKRISHNAN, H. 2003. The Aurora and Medusa projects. *IEEE Data Engineering Bulletin 26,* 1.