Characterizing, Constructing and Managing Resource Usage Profiles of System S Applications: Challenges and Experience

Sujay Parekh
sujay@us.ibm.com

Kirsten W. Hildrum
hildrum@us.ibm.com

Deepak Rajan
drajan@us.ibm.com

Joel L. Wolf
jlwolf@us.ibm.com

Kun-Lung Wu
klwu@us.ibm.com

IBM T.J. Watson Research Center, Hawthorne, NY 10532

ABSTRACT

We describe the challenges of characterizing, constructing and managing the usage profiles of System S applications. A running System S application is a directed graph with software processing elements (PEs) as vertices and data streams as edges connecting the PEs. The resource usage of each PE is a critical input to the runtime scheduler for proper resource allocation. We represent the resource usage of PEs in terms of resource functions (RFs) that are used by the System S scheduler, with one RF per resource per PE. The first challenge is that it is difficult to build good RFs that can accurately predict the resource usage of a PE because the PEs perform arbitrary computations. A second set of challenges arises in managing the RFs and performance data so that we can apply them for PEs that are re-run or reused by the same or different applications or users. We report our experience in overcoming these challenges. Specifically, we present an empirical characterization of PE RFs from several real streaming applications running in a System S testbed. This indicates that our simple models of resource usage that build on the data-flow nature of the underlying application can be effective, even for complex PEs. To illustrate our methodology, we evaluate and analyze the performance of these applications as a function of the quality of our resource profile models. The system automatically learns the models from the raw metrics data collected from running PEs. We describe our approach to managing the metrics and RF models, which allows us to construct generalizable RFs and eliminates the learning time for new PEs by intelligently storing and reusing the metrics data.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Measurements, Modeling and prediction*

General Terms

Measurement, performance

Keywords

Streaming, resource model, performance characterization

1. INTRODUCTION

System S [2, 7, 17, 18] and other distributed stream processing systems [1, 10, 20] are geared toward processing long-running queries on continuous streams of data. Streaming applications in such systems are typically organized as data-flow graphs with the software processing elements (the PEs) as verticies and the data streams as directed edges connecting the PEs.

A key resource allocation problem faced by the runtime scheduler in such systems is to map the PEs in the applications to compute resources in a way that utilizes the available CPU and network resources efficiently without overloading any individual node or network link. A critical input to the runtime scheduler for solving such a problem is the "size" of a PE, that is, the resource usage of a PE. The resource demand r_p of a PE p for resource r is given by the functional form $r_p = f_{p,r}(d_1, d_2, ...)$ where $d_1, d_2, ...$ are the factors on which the resource usage depends. The function $f_{p,r}$ is called a resource function (RF) and is a model of the PE's resource usage.

There are several challenges that arise in the context of RFs. First and foremost, accurately predicting the resource usage of PEs can be difficult. A PE can be the result of combining multiple simpler streaming operations, or it can directly implement a user-defined, arbitrarily complex data analytic algorithm. Thus, the PE sizes are not fixed or even known a-priori. Furthermore, the resource usage of a PE can depend on dynamic characteristics of the input streams (such as data volume and data content). As these characteristics change, the resource usage of a PE may increase or decrease.

A second set of challenges arises in the context of managing the RFs, 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 (eg, 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 (i.e., 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. Hence, one question is to identify a PE so we can associate an RF with it. In a similar vein, how should RFs 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? For PEs that may not have been run before – what sensible initial RF can be provided?

In this paper, we describe our experience in addressing these challenges. Specifically, we present our practical approach to characterizing, constructing and managing the resource usage profiles of stream PEs for the purpose of providing a critical input to the SODA scheduler [16, 17] in System S. (SODA, the Scheduling Optimizer for Distributed Applications, is the runtime scheduler for System S .) Our approach is not the only one; other alternatives certainly exist, and we continue to explore some of them. Moreover, we focus only on learning the resource usage of CPU and network.

With inaccurate models, the scheduler may place PEs in ways that reduce application throughput. To highlight the impact of RF inaccuracy on system performance, we compare in Figure 1 the total ingest rate achieved for two applications: DAC [18] and SKA [5] when SODA has increasingly incongruous RFs. The higher the total ingest rate, the better the system performance. The applications are described in Section 2.2 and the experiments are detailed in Section 3.5.1. An incongruity level of 1 represents the application performance when it is scheduled with SODA using the actual trained RFs, i.e., the most congruous/accurate RFs. Higher incongruity levels represent RFs which are increasingly divergent from these trained RFs, and therefore more inaccurate. We see that the RFs need not be completely accurate to achieve good performance in practice (see the incongruity levels of 3 and 5 for DAC and the incongruity level of 1.5 for SKA). However, they cannot be too far off, either. For example, with severely incongruous RFs, the application performance can reduce significantly, e.g., by over 30% for DAC and over 50% for SKA. In extreme cases, the application may even fail to start.

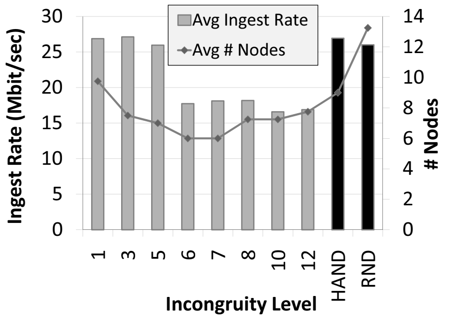
Specifically, based on our experience, we make the following contributions in this paper:

- We demonstrate a simple data flow-based approach to modeling the resource usage of PEs. We show that simple piecewise linear models are generally effective in practice, even for complex PEs. We validate this approach empirically against PEs from several System S applications, including both simple and complex PEs.

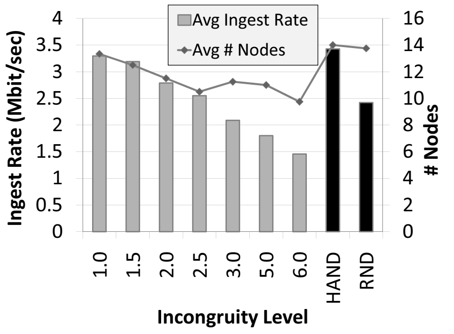
- We show a practical scheme for managing and building these models in a way that maximizes the usage of raw input (training) data and enables the learned resource models to be generalized to new PEs.

- We validate that the RFs need not be very accurate. Schedules generated by SODA degrade, but gracefully, as the incongruity in RFs grows. The sensitivity to RF incongruity appears to depend on the spare capacity in the system as a whole.

The rest of the paper is organized as follows. In Section 2 we introduce System S and describe the testbed and applications used in the experiments in this paper. Next, we present in Section 3 the resource model for CPU and network that we use, along with an evaluation of the resource model sensitivity. A description of our approach to model management is given in Section 4. Related work is reviewed in Section 5, and we conclude in Section 6.



(a) DAC



(b) SKA

Figure 1: Scheduling effects of bad resource profiles

2. BACKGROUND

2.1 System S

System S is a large-scale distributed stream processing middleware being developed at IBM Research. 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 [7]. 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 *processing element* (PE), which is the unit of deployment in System S. 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,

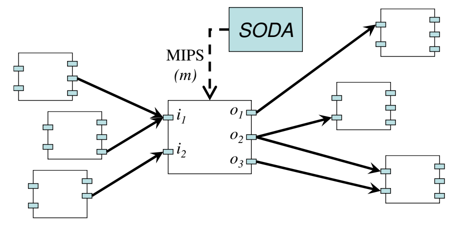


Figure 2: Illustration of a PE showing input ports i_k, output ports o_k and streams.

and may contain of one or more threads of execution. The PEs of an application are distributed across the compute nodes. Each compute node can run multiple PEs and divides its CPU resource between them according to fractions dictated by the SODA scheduler (Section 2.1.1). 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.

A general PE is depicted in Figure 2. 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.

2.1.1 SODA

Resource allocation in System S is performed by a centralized, epoch-based scheduler called SODA. In streaming systems, the jobs are usually long-running, and often continue to run until they are terminated by the user. Therefore, metrics such as completion time and response time, traditional in batch processing systems, are not relevant. Instead, SODA maximizes a utility-theoretic measure known as *importance* subject to a variety of real-world constraints. Because the best way of running a job may depend on the available resources and competing jobs, SODA allows a user to submit multiple job templates for a job, only one of which will be chosen to run. To schedule, SODA analyzes a vast number of PE resource allocation alternatives for different job admission and template choices, using importance as a black box objective function.

The importance metric is defined quite generally [17], but for our purposes in this paper it is sufficient to think in terms a highly common special case. Specifically, importance can be regarded a weighted sum of rates at the terminal streams in the data-flow graphs. The notion is that these streams represent the final "products" of the various jobs, and the weighted value functions translate these stream rates into measures of goodness of the work done in System S. In order to perform its job, SODA must therefore have a way to accurately predict the resource usage and output rates of the PEs in the system. In fact, any scheduler could benefit from accurate resource prediction. The RFs are the mechanism for the scheduler to generate these predictions under various resource assignment scenarios.

A streaming scheduler such as SODA must also consider *flow-balance* when computing resource allocations. By flow balance we mean that PEs are given the *right* level of resource allocation relative to their predecessors. To understand this, consider one particular PE. Giving relatively too many resources to the PE's predecessors will "flood" the PE, while giving relatively too few resources will starve the PE. The allocation levels need to be balanced throughout the entire data-flow graph.

For the scheduler to do its job, the RFs must provide estimates of the CPU requirements of the PEs and the network traffic between PEs. A static estimate of resource usage would perform poorly, because these requirements could (and do) vary according to several factors. For example, a classification PE will likely need resources that depend upon its input rate, and will have an output rate that also depends on its input rate. Resource usage could also depend on properties of the input to the PE – if the classifier above is classifying speech based on audio clips, the length of the audio clips could affect the required resources.

Without going into further details of the scheduler algorithm (see [16]), there are some requirements imposed on the RFs:

- Scope: A PE should have a common RF across the nodes in the cluster, even for heterogeneous clusters. Otherwise the scheduling problem becomes intractable.
- Accuracy: Since the required accuracy depends on the resource granularity chosen by scheduler, we may, in practice, get away with less than perfect RFs as long as they are not dramatically inaccurate.
- Form: The RFs should be monotonic non-decreasing in their input parameters. Before some input parameter threshold, increases in either the compute resources or the input rate should also increase the output rate. After this threshold the function should become flat. This is an implicit and natural assumption of the scheduler algorithm. The consequences were discovered experimentally, as discussed in Section 3.6.2.

2.2 Streaming Applications

For our tests, we study PEs from four applications running on System S; these represent different but typical uses of streaming systems.

- **DAC** [18] represents an insurance claims fraud detection and alerting system involving some heavy streaming analytics, i.e., CPU-intensive complex stream mining algorithms. Consisting of six jobs and 51 PEs. DAC provides some scheduling challenges because its PEs have a wide range of processing requirements.
- **SKA** [5] is a radio astronomy application which reconstructs images from data received by radio telescope antennas using interferometry [6]. SKA includes PEs performing processor and memory intensive computations on large amounts of streaming data.
- **Fab** [11] is an application that processes streaming data from automated tests in a chip manufacturing plant, with a goal of monitoring and altering the process to improve yields.
- **VWAP** [3] represents a financial markets scenario in which real-time quotes are processed to detect bargains and trading opportunities.

We use DAC and SKA for most of our application-level experiments, while Fab and VWAP provide some interesting operators which are useful to study on their own.

2.3 Testbed

The experiments and data discussed in this paper are collected using a System S deployment on a cluster consisting of IBM BladeCenters running Linux 2.6.9. We run our applications on 14 blades with dual-CPU, dual-core 3GHz Intel Xeon processors with 8GB RAM. The blades are in the same rack; they communicate over 1GBit/s links and are inter-connected using a high-speed 20GBit/s backplane.

The input data for DAC is provided by a set of workload generators, while the other applications use source operators that read data from the filesystem. In both cases, the data enters the system as fast as the system can ingest it. The system is operating in reliable transport mode (no tuples dropped between PEs in the system). The applications are configured to run for 30 minutes, each run (for a particular setting) is repeated 4 times, and averages are collected. We collect infrastructure metrics such as CPU and network traffic rates in terms of averages for 1 minute intervals. Metrics are discussed further in Section 3.5.

For the specific set of applications and system software used in our experiments, the network, backplane or network interface card (NIC) is almost never a bottleneck. Thus, the only disadvantage of placing two PEs that communicate with each other on separate blades is the additional processing overhead involved in sending data to a different node. Although SODA allocates PEs to the nodes while trying to minimize the traffic across blades, this feature is not critical for the specific combination of infrastructure and applications described here. Workloads that will stress this aspect of the system are currently being developed. Nevertheless, accurate RFs are a key factor in SODAs ability to balance the load on the processing nodes.

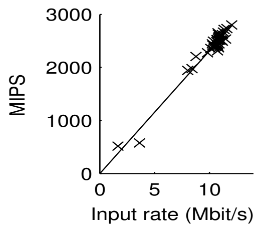
3. PE RESOURCE USAGE MODELING

A PE's RF represents a resource usage model for that PE. In this section we present a model for streaming PEs, and validate it against actual PEs from the test applications described above. Our RF models focus on the usage of CPU and network by the PE.

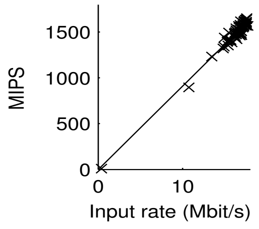
3.1 Model Parameters

In order to predict the resource usage of a PE, we must consider two broad categories of factors: *dynamic* and *static*, depending on whether they vary at runtime. Dynamic factors include the input data rates to the PE, the distribution of input data types, and data content. Static factors include the PE code, the PE arguments, stream flowspecs and system configuration such as the communication model. One factor which could be regarded as dynamic but is treated as static for modeling purposes is the nature of physical resources given to the PE, for example, whether it is running on generic x86 hardware or special-purpose processors. Also, the data content is difficult to characterize/represent in the general case, so in the interest of simplicity we do not consider it. PEs whose behavior is adaptive to available resources may further depend on the time history of system load and traffic. The use of the static PE attributes are discussed later, in Section 4.

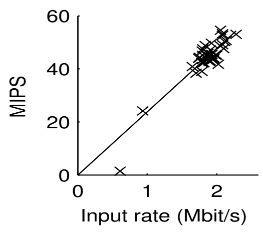
We also remark that RFs are further classified as *source*,



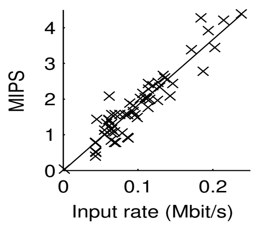
(a) DAC LoadDiffusion



(b) Fab Splitter PE



(c) DAC VOIP.LoadStar



(d) Fab Barrier PE

Figure 3: Linear MIPS profile

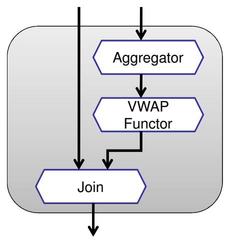
sink, or *transform* RFs, depending on whether the PE is a source PE (feeding from primal streams only), a sink PE (not writing any streams, except, perhaps, to disk), or a transform PE (everything else), respectively. Our discussion in this section assumes that the PE is a transform PE (with both input and output streams), but the methodology and techniques naturally carry over to the other types. The notion of PE types will re-appear when we address RF model management in Section 4.

3.2 Modeling CPU Usage

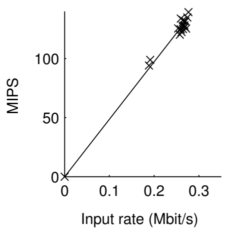
A challenge in modeling the CPU needs of a program is that this demand will vary depending on the specific CPU being used. To enable us to construct a general model that can be used across all nodes in a heterogeneous cluster, we use MIPS as a measure of CPU demand. The MIPS used here is the processor BogoMips [15] reported by the Linux kernel. The MIPS consumed by a PE is calculated by multiplying (a) the CPU time fraction used by a PE on a specific node (reported by the OS) and (b) the total BogoMips of the cores on a processor. In spite of the known limitations [15] of the BogoMips measure, this approach is still useful as it allows us to generalize across CPU speed variability within the same processor family.

For processing resources, a natural model is to consider a processing cost per incoming data object. In a queueing theory sense, this is the "service time" per request. Even though there is not necessarily a well-defined "service time" for each incoming data object in streaming systems, in the aggregate, one may expect the CPU requirements of a PE to scale proportionally with input rate. For simplicity we consider a linear scaling based on the *total* data rate into the PE; we do not distinguish the rates on individual ports.

To validate this model, we study the behavior of PEs in our test applications. Some examples are shown in Figure 3. For a majority of PEs, the MIPS-per-datum model seems to hold true. This seems to be the case for both heavier, computationally more demanding PEs (Figure 3(a), Figure 3(b))



(a) VWAP PE Internals



(b) VWAP PE RF

Figure 4: Internal structure and RF of a fused PE from VWAP

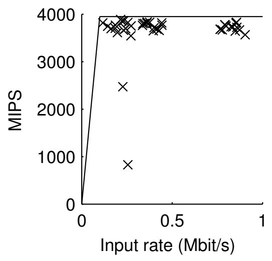


Figure 5: Maximum MIPS profile (DAC LoadDiffusion.JoinOperator)

as well as lighter PEs that use few MIPS per datum (Figure 3(c), Figure 3(d)). In the case of the SPADE-based applications, this model applies even for those PEs which are composed (fused) from simpler SPADE operators. As an example, the PE shown in Figure 4(a) is composed of an *aggregator*, a *functor* and a *join*. Yet, the MIPS profile follows the same linear MIPS-per-tuple pattern as seen in Figure 4(b). Surprisingly, even though a join operation may produce output proportional to the product of its inputs, many PEs containing joins show this linear behavior.

Some PEs, on the other hand, notably from the DAC application, consume a lot of CPU even when the input rates are not very large, effectively saturating the processing node. One example is shown in Figure 5. The PEs shown are single-threaded, and the BogoMips capacity of one core of the node is 4000 MIPS. However, even such PEs can be modeled using the linear model described in (1). Some PEs in our example applications do not show a linear relationship between CPU and input rate, but ultimately, we get good predictions in spite of this fact.

Based on our experience, we present our initial candidate model for CPU.

$$m_p = \min(M_p, a_p \sum_{k \in \text{IPorts}(p)} r^{i_k}) \qquad (1)$$

As a function of the input rates r^{i_k} on all input ports i_k of PE p, the model (1) predicts the PE's MIPS usage (m_p). Here, IPorts is the set of input ports to PE p, and M_p represents the maximum MIPS that can be allocated to the PE p. This limit often occurs due to system considerations: the maximum MIPS on any processing node in the system, for instance. Even when this is not the case, the limit allows us to place bounds on the search space explored by SODA. In this model, the term a_p represents the best-fit slope of the MIPS needed by the PE as a function of its input rate.

To illustrate, consider the DAC Load Diffusion PE illustrated in Figure 3(a). For this PE, the coefficients M_p and a_p in (1) can be derived from fitting the best linear model to the data, yielding $M_p = 3000$, and $a_p = 200$. Also consider the Join PE from DAC, as illustrated in Figure 5. By recognizing that the resource usage of such PEs is largely determined by the maximum MIPS available, we can set a_p sufficiently large (in this case larger than 50000), and set $M_p = 4000$ (from the observed data). Now the first term in (1) dominates the second, resulting in a MIPS prediction m_p of 4000 for any sufficiently large data input rate.

An alternate approach to modeling PE-level RFs is to build RFs for each operator and then combine them to get whole-PE RFs. We have found that for any non-trivial operator topology within a PE, it is not obvious how the RFs should be combined. The PE-level RF seems to depend on the details of the operators' implementations (such as their multi-threaded behavior) as well as their connection topology. In any case, we need a whole-PE approach to model PEs which may not be created from operators. Hence, we use an approach that does not depend on the intra-PE operator graph.

3.3 Modeling Output Rates

Output rates for most streaming PEs are relatively easy to model, since in many cases they have a 1:1 relationship to the inputs, meaning that for every incoming tuple, an outgoing tuple is produced. Operations that do not fit this pattern include filtering, aggregation and timer-based data output. We use the following model for output rate:

$$r^{o_j} = \min(R_p, b_p \sum_{k \in \text{IPorts}(p)} r^{i_k}) \qquad (2)$$

where, r^{o_j} is the predicted output rate for output port o_j of PE p. Thus, for PEs that have a 1:1 relationship to the inputs, b_p is equal to 1, and R_p is set to the largest data rates seen at the output port. Consider the PLLTest PE from Fab, illustrated in Figure 6. From the data, we observe that $R_p = 5$. Filtering PEs will have $b_p < 1$. Tuple count-based operations (that produce an output for every k input tuples) will have the slope $b_p = 1/k$. PEs that produce output based on timers are the exception, requiring a different model since they are independent of the input rate. For such output ports of PEs (output every T time units), the model takes the form $r = 1/T$.

3.4 Building RFs

A parametric RF can be built for each output port of a PE. Such an RF first describes the type of model considered, followed by all the relevant parameters.

This study also reveals a methodological challenge: the stable flow balance during this workload generator driven data collection can result in very narrow input rates observed at the PEs. This can result in skewed data for the resource function learning. One way to address this is to explicitly "tweak" the input rates to each PE to ensure coverage over some reasonable range, so that the function fitting will be valid and generalizable outside the range observed during the calibration step.

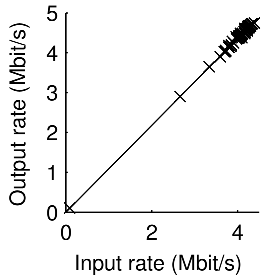


Figure 6: Linear output rate profile (Fab PLLTest PE)

3.5 Model Evaluation

To evaluate the RFs, we run the applications as described in Section 2.3, and look at the following application and infrastructure level 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 with importance. In stream processing systems such as System S, flow-balanced resource allocations for the PEs and a load-balanced allocation of PEs to processing nodes minimizes bottlenecks, thus maximizing the amount of data processed at the source PEs (ingest rate).

- *Stream affinity*: One way to measure the quality of the placement is in terms of the traffic load on the system. We compute the amount of traffic that is sent between PEs on the same node divided by the total traffic. The higher this quantity, the better, since PEs which share a stream should be put on the same node (or nearby) to minimize network utilization.

- *Maximum node utilization*: This is a measure of how well SODA distributes the processing load across various machines. This metric is especially interesting when evaluated in conjunction with stream affinity; SODA attempts to maximize stream affinity while simultaneously minimizing maximum node utilization.

The first metric (ingest rate) is the most tangible measure of system performance for streaming systems. The latter two metrics, in conjunction, illustrate the quality of the placement of PEs to processing nodes. Given the same stream affinity, smaller the maximum node utilization the better. Given the same maximum node utilization, the higher the stream affinity the better. These metrics are computed from the raw system metrics such as CPU usage per PE and traffic consumed and produced by each PE.

3.5.1 *Making RFs Less Congruous*

This section describes the experiments shown in Figure 1 and presents additional results. We deem as "good" the RFs which are derived from data collected from a run (denoted HAND) in which the PEs are placed by an expert. We then start systematically degrading these RFs. The expert placement is an allocation of PEs to processing nodes determined via a trial-and-error process. To illustrate the effectiveness of these learned RFs, we compare them to the performance of the HAND placement. Note that the RFs are not tuned by hand; their parameters are learned automatically by the system using the models described in Sections 3.2 and 3.3.

To analyze the deterioration of RFs carefully, we parameterize the de-tuning of the RFs using a parameter κ. (When $\kappa = 1$, the RFs are not modified.) Since our modifications involve generating random numbers, to ensure that these modifications are consistent across runs and values of κ, we pre-generate a sequence of random numbers, and use the same sequence for all the runs. This is equivalent to seeding our random number generator with the same value for each run.

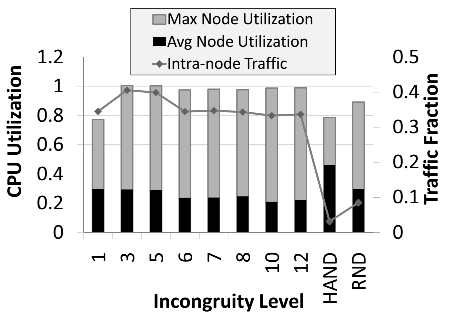
We now describe precisely how each term in an RF is changed, given κ. Fix a particular value of κ. We modify each term t in an RF by a factor α, as follows. We draw two random numbers, say r_1 and r_2. The first number r_1 determines how much the term t gets modified, and the second number determines whether t is increased or decreased. Let $round(a, b)$ denote the value of a rounded to the nearest multiple of b. We set $\alpha = 1 + round(r_1 * (\kappa - 1), 0.1)$; Thus, α is some multiple of 0.1 between the values 1 and κ, determined by the value of the random number r_1. If $r_2 < 0.5$, we multiply t by α. Otherwise we divide t by α. Observe that when $\kappa = 1$, $\alpha = 1$, and t is not modified. Furthermore, as κ is increased, the amount of de-tuning increases probabilistically. This allows us to modify each term in the RFs in a controllable fashion, analyzing the performance of SODA as the RFs degrade. Since we use the same sequence of random numbers, each term is modified using the same r_1, r_2 in all the runs. Different κ values will result in different values of α that are strongly correlated with κ, as desired. As we increase κ to ∞, α also increases to ∞ (for positive r_1), and the RF term t either increases to ∞ or decreases to 0, depending on whether r_2 is smaller or larger than 0.5.

We illustrate, using RF modeling, the CPU usage of the DAC Load Diffusion PE (see Figure 3(a)). The RF for this PE has two terms, $M_p = 3000$ and $a_p = 200$. Consider the first term. Suppose the two random numbers r_1 and r_2 are 0.84 and 0.39. When $\kappa = 3$, $\alpha = 2.7$, and M_p is modified to 8100. On the other hand, when $\kappa = 5$, $\alpha = 4.4$, and M_p is modified to 13200. If $\kappa = 1$, the value of M_p is unchanged. Observe that larger values of r_1 result in a larger perturbation for a given κ, and also that if $r_2 > 0.5$, the value of M_p would have decreased instead.

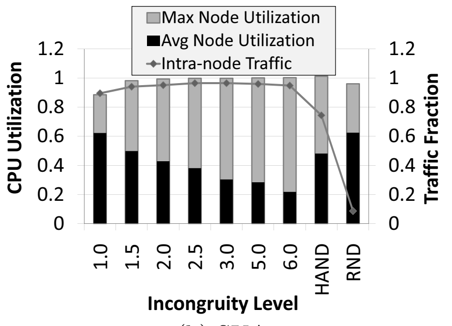
3.5.2 *Impact on Application Performance*

The effect of making the RFs less congruous is to make SODA's estimates of the resource requirements of the various PEs more inaccurate. SODA first determines the CPU requirements of the PEs (and traffic between PEs) in the submitted job in the process of maximizing the net importance of the system. These numbers are then used as input to a second phase to allocate the PEs to nodes in an intelligent manner (load-balancing the nodes and minimizing inter-node traffic simultaneously). Less congruous RFs results in worse estimates and thus less intelligent placement.

These effects are shown in Figure 1 and Figure 7 for two applications: DAC and SKA. We also show the performance of the expert placement (HAND) and the RND scheduler that assigns PEs to nodes in the cluster randomly. Given our cluster size, the Fab and VWAP applications did not present enough work, so we do not use them in the evalua-



(a) DAC



(b) SKA

Figure 7: Effect of scheduling with bad resource profiles on node utilization and intra-node traffic

tion here. We see that for SKA there is a systematic drop in performance (ingest rate) as the RFs are made more incongruous, whereas for DAC, the performance is relatively insensitive to RF perturbation. In both cases, we see that the ingest rate of congruous RFs is comparable to that of HAND. This illustrates that our simple RFs encapsulate all the necessary information for the runtime scheduler to place the application as well as a domain expert. In other words, our RFs are effective in practice, with not much room for improvement in performance, at least for the applications in our testbed. In SKA, we see that RND performance is considerably worse than SODA with congruous RFs. On the other hand, the performance of the RND placement on DAC is close to that of SODA with congruous RFs (see Figure 1), even though RND uses more nodes than SODA with congruous RFs, thus sending a larger traffic load on the network (smaller stream affinity).

This behavior of the RF sensitivity as well as effectiveness of RND placement is explained by considering the overall node utilizations of the various scenarios (see Figure 7). We see that in general, DAC has low average utilization, implying it is very over-provisioned. In this case, the effect of a poor placement (in terms of network traffic load and node utilization) on ingest rate is not going to be significant until the placement gets dramatically tweaked. For SKA, on the other hand, some PEs are quite computationally intensive. Here, the nodes are well utilized in a placement computed with congruous RFs, and the effect of making the RFs less congruous is to create bottlenecks. This increases the maximum node utilization while decreasing average utilization. For SKA, the RND case also seems to utilize the nodes well, but because it does not account for intra-node traffic, its performance is much lower than SODA with congruous RFs.

Thus, we see that in the under-utilized case, the performance can be less sensitive to RF incongruity than under higher node utilizations. On the other hand, with larger workloads (higher node utilizations as in SKA), and with mixed application workloads, the performance of the scheduler SODA deteriorates significantly when using incongruous RFs as input data.

3.6 Experience with Advanced Models

In this section, we describe our experience and lessons learned with more advanced models than the simple models presented in Section 3.2 and Section 3.3. We also consider the impact of alternate data transport mechanisms, from the perspective of both the RFs and the scheduler itself.

3.6.1 Unreliable Data Transport

In the preceding discussion, the RF was a function from the input rates to either MIPS or to the output rate. This was because System S, by default, operates in reliable data transport mode. In other words, no tuples are dropped, and queues get backed up if the PEs do not get sufficient resources. As a result, in these RFs, input rates are the only independent variables. We also refer to this mode of operation, and the corresponding RFs, as the "no-drop" model.

On the other hand, System S can also operate in an unreliable mode, in the sense that reducing the MIPS allocation to a PE effectively forces tuple drops. We call this the "drop" model. It affects the RF and the scheduler primarily in the following way.

- Implications to SODA: the scheduler now has one more knob to manipulate in attempting to maximize system importance. It can intentionally decide to give less resources to some PEs than they would otherwise need. In some sense, SODA can now decide to "partially" allocate resources to PEs in some jobs, or even to portions of a job. To be able to do so meaningfully, SODA needs to know how the data output rates change as a function of the MIPS allocated to the PEs. In other words, MIPS is no longer an independent variable in the RFs.

- Implications to RFs: Now, the RF is a function mapping from the input rates and MIPS to the output rate. These sorts of functions, where MIPS are input rather than output, are appropriate for the cases in which resource usage limitations result in tuples drops. These new kinds of RFs need to be learned from the data.

3.6.2 Non-parametric Approaches

The model we described previously is a parametric model. It assumes a particular formula and then determines parameters that best fit the data. If the formula is wrong, then even the best fit model will be a bad fit. An alternative is to use a non-parametric model. Non-parametric models, as the name suggests, do not assume a particular form of the resource function.

In particular, we elaborate on our experiences using one particular kind of non-parametric model for the RFs: a decision tree [9], with the "no-drop" mode mentioned above. Although the resulting RFs for a decision tree based model cannot be described as compactly as the parametric RFs, we hoped that using these more accurate RFs would result in better scheduling results.

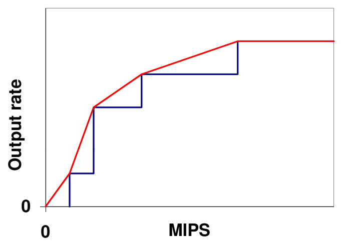


Figure 8: Decision tree RF and its envelope

The decision tree is trained using training data, with the CPU usage and input rate as the classification attributes. The output of the decision tree is an output rate, which is used by the scheduler. Given a set of training data, the output would be one of the values from the training set.

However, our initial decision-tree-based RFs resulted in bad scheduling decisions because all the PEs were allocated low amounts of resources. Upon investigation, we learned that the data were noisy. The noisy data resulted in decision tree outputs that did not satisfy the implicit form assumptions by the scheduler (listed in Section 2.1.1).

As a general trend, when MIPS increases, the output rate also increases. However, because the data were noisy, there were cases where $\mathsf{MIPS}_1 < \mathsf{MIPS}_2$ but $orate_1 > orate_2$. When the SODA scheduler makes RF queries and encounters a point where increasing the MIPS decreases the output rate, the scheduler naturally "assumes" that further increasing the MIPS will not produce a higher output. As a consequence, the source PEs (and correspondingly, the downstream PEs) end up getting too few MIPS. The question becomes how to deal with this mismatch in the data and the scheduling algorithm.

Our first response was cleaning up the data by discarding points that caused the problem. Once those outlier points had been removed, the resulting RF had a stair-step shape as in Figure 8. These RFs still had similar problems as the previous ones, for a very similar reason. When the SODA scheduler encounters a flat portion of the function, where $\mathsf{MIPS}_1 < \mathsf{MIPS}_2$ and $orate_1 = orate_2$, it also does not probe further. (Essentially, it treats the RF as if the first flat spot lasted forever.) Thus, the MIPS assignments were again too small.

As a next step, we tried interpolating between the outer points (see the envelope in Figure 8), so long as there was only one input. This model is effectively piecewise-linear and not truly non-parametric. The results achieved were not significantly better, so we discarded that approach as well. If there is more than one input, it is difficult to see how to ensure that the surface is increasing in both directions, and we did not test models of that type.

We also considered a parametric model that did not include the maximum function. The problem encountered there was quite interesting: Unbounded, the SODA scheduler probed too far, finding improvements that could never be realized. Though the optimization was not affected, the running time became much longer.

From these different scenarios, we learned two lessons about the RFs. First, the RFs must initially be strictly increasing. Once they stop increasing, SODA will stop further exploration. Second, they must eventually reach threshold where they stop increasing. While these lessons are specific to the SODA scheduler, they should be applicable to other schedulers which use RFs as their inputs. Furthermore, because of the fundamentally different way in which SODA works in the no-drop case, it may be possible for other model types to be effective in the no-drop case; this is a topic of future research.

4. MODEL AND DATA MANAGEMENT

The study of real PEs in the previous section indicates that an empirical approach based on collecting a few observations from each PE and constructing a model could be a simple and practical approach to obtaining useful RFs. While the previous section addresses the mathematics issues, it does not touch the management issues. It leaves several questions unanswered:

- 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?
- 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? In addition to the input rate dependency, a PE resource usage can depend on other factors such as: the nature of the input data (which may be a function of the upstream PEs), PE configuration (via, for example, command-line arguments), or systems issues, such as processing node architecture. This issue arises also when the same PE (e.g., a classifier) is reused in multiple applications.
- How should metrics be stored and used to build and update the PE RFs?

Our observation is that even though there are a myriad of factors (in addition to input rate) that affect a PE's behavior, a PE will not be run in every possible configuration. Thus, rather than building a model that explicitly tries to model all these factors, we build and manage separate models for different combinations of those factors. This multiplicity of models raises the need for managing the models as well as the raw metrics data that is collected from the system. Specifically, two aspects must be addressed: (a) given a specific instance of a PE in a specific job that is submitted to the system, which model should be used by the scheduler? (b) given an observation of the resource usage of a specific PE instance, which models should be updated, and how?

To facilitate the model management, each PE is associated with a multi-part *signature*. We learn and maintain 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**: source, transform or sink, as described in Section 3.1. 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.

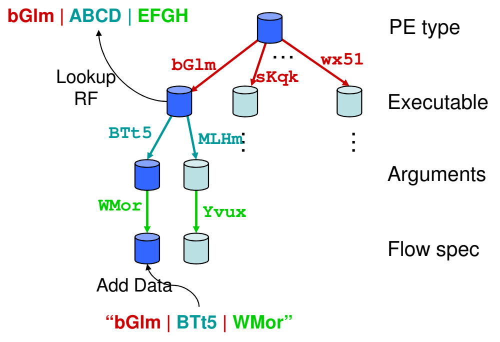


Figure 9: Hierarchical PE signature management

- **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.

The signature captures some 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 9. 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 lookup 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 in the database (yet). 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 on only 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 by us based on our calibrations from some 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, we can obtain some information about that PE's behavior by using this generalized information.

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

Table 1: Signature counts by level

To highlight the possibility of reuse in actual applications, Table 1 shows the number of unique nodes at each level in the tree for our applications. 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.

5. RELATED WORK

In the literature on resource allocation and scheduling in distributed systems, the resource requirements are typically assumed to be known or given. Much of the other known modeling work has occured in the context of single and multi-tier distributed systems. In [12], the authors develop and use linear models for CPU, disk and memory demands based on incoming workload rates, similar to our models. A more complex, analytical queueing model of multi-tier services is developed in [13]. However, this model is difficult to apply to streaming systems which are not neatly organized into tiers – in general they are directed graphs, may have cycles and little identifiable substructure.

In general cluster environments, [14] use kernel-based monitoring tools to learn application profiles in terms of stochastic token-bucket models of CPU and network usage. This approach builds an application-level workload-independent usage profile, and schedules to the tail of the distribution. In our case, for streaming applications, the workload data rates are expected to vary widely, and the system is expected to be quite dynamic. As we see, PEs are very sensitive to the incoming data rates, so a rate-sensitive model is needed to ensure responsiveness to changes in resource demands.

For streaming systems, a cost-per-tuple model is also proposed [19] in the context of the Borealis system. Their operators, however, are much simpler (like SPADE operators) compared to the PEs in our system. We improve on their work by showing that even complex PEs can be modeled using a similar approach, and further we propose a scheme for managing and generalizing these models.

[12] also raises the issues that models may become inaccurate due to interference caused by colocation of PEs on a node. They mention that developing models in heterogeneous clusters is a challenge, but others [8] have suggested a solution involving parametric cross-architecture models.

Model management is an even less discussed topic. For the profiling step, some authors [4, 12] suggest running the applications on idle nodes for accurate measurements.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented the challenges of predicting the CPU and network usage of PEs in System S applications. An empirical study of PEs from applications in a System S testbed reveals that simple piecewise linear models based on their input rates are sufficient for modeling these PEs, which is encouraging given that some of these PEs perform relatively complex analytics. We have also presented an approach based on hierarchical PE signatures for managing and updating these models that addresses the issues of getting usable models for new PEs and allowing one PE's model to be used effectively for another PE. Although not discussed in this paper, our RFs are dynamically updated based on new observations from the running PEs, this allows the SODA scheduler to respond to dynamic changes in resource demands of the applications.

Although we find that many PEs can be modeled using these linear models, there will be PEs that do not fit into this scheme. Although our initial attempts with non-parametric approaches were not very successful, we aim to refine them and pursue other advanced techniques that can capture a larger set of PEs. Our current models also are limited to handling CPU and network, and may even be generalizable to disk resources [12]. However, modeling memory consumption is still an open issue. In general, since PEs can arbitrarily allocate memory at runtime, this is a difficult issue. Currently SODA relies on PE developers to provide hints about the memory needs of their applications, but an automated RF-based approach would allow even the memory demands to be taken into account during scheduling. Finally, our use of the MIPS metric does not generalize well across architectures. For clusters of truly heterogeneous nodes, especially ones containing specialized resources (such as the Cell processor), a more generalizable metric and model (for example, [8]) would be very useful.

7. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proc. Conf. on Innovative Data Systems Research*, 2005.

[2] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: a distributed, scalable platform for data mining. In *Proc. Int'l Wkshp on Data Mining Standards, Services and Platforms*, 2006.

[3] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu. Scale-up strategies for processing high-rate data streams in System S. In *Proc. IEEE Int'l Conf. on Data Engg., to appear*, 2009.

[4] M. Aron. *Differentiated and Predictable Quality of Service in Web Server Systems.* PhD thesis, Computer Science, Rice University, Oct. 2000.

[5] A. Biem. Imaging for next-generation radio telescopes. Personal communication, July 2008.

[6] T. J. Cornwell, K. Golap, and S. Bhatnagar. W projection: A new algorithm for wide field imaging with radio synthesis arrays. In *Proc. Astronomical Data Analysis Software and Systems XIV*, volume 347 of *2005 ASP Conference Series*, 2005.

[7] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *Proc. ACM SIGMOD.* ACM, 2008.

[8] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proc. 2004 ACM SIGMETRICS*, 2004.

[9] T. M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[10] R. Motwani, J. Widom, A. Arasu, B. Babcokc, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. 1st Conf. on Innovative Data Systems Research*, 2003.

[11] D. Sankus, R. Redburn, D. S. Turaga, M. C. Johnson, A. Norfleet, O. Verscheure, and W. Fan. Drowning in data - a streaming solution, Submitted to Int'l Test Conf., 2009.

[12] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Proc. NSDI'05*, 2005.

[13] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proc. 2005 ACM SIGMETRICS*, 2005.

[14] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proc. OSDI'02*, 2002.

[15] W. van Dorst. BogoMips mini-Howto. `http://tldp.org/HOWTO/BogoMips/`.

[16] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, and K.-L. Wu. Job admission and resource allocation in distributed streaming systems. In *Proc. Int'l Wkshp on Job Scheduling Strategies for Parallel Processing*, 2009.

[17] J. L. Wolf, N. Bansal, K. W. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Proc. Int'l Middleware Conf.*, 2008.

[18] K.-L. Wu, P. S. Yu, B. Gedik, K. W. Hildrum, C. C. Aggarwal, E. Bouillet, W. Fan, D. A. George, X. Gu, G. Luo, and H. Wang. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In *Proc. VLDB '07*, 2007.

[19] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. B. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proc. VLDB '06*, pages 775–786. ACM, 2006.

[20] S. Zdonik, M. Stonebraker, M. Cherniack, U. Cetintemel, M. Balazinska, and H. Balakrishnan. The Aurora and Medusa projects. *IEEE Data Engineering Bulletin*, 26(1), 2003.