RC24945 (W0901-038) January 12, 2009
Computer Science

IBM Research Report

Performance Modeling of Operators in a Streaming System

Xiaolan J. Zhang
UIUC

Sujay S. Parekh, Bugra Gedik, Henrique Andrade, Kun-Lung Wu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Performance Modeling of Operators in a Streaming System

ABSTRACT

Modeling the resource consumption of each runtime processing element (PE) is essential to the optimal resource allocation of System S–a distributed streaming processing platform. SPADE is the programming language of System S for developing streaming applications using an operator-based approach. Because a SPADE operator tends to be small in CPU consumption, multiple operators are usually fused at compile time into PEs for efficient runtime deployment. As a result, modeling the resource function (RF) at the SPADE operator level becomes increasingly important for the system to optimally (1) fuse operators into PEs at compile time and (2) allocate PEs to physical nodes at runtime. There are two main challenges in modeling operator-level resource functions. First, how do we recover the baseline operator-level resource functions (OP RFs) from the raw data collected with limited precision and under a changing runtime environment? Second, how do we estimate the resource function for a PE with any given fusion and node mapping from the baseline OP RFs?

In this paper, we propose a new operator-level RF learning infrastructure for System S. (i) The infrastructure specifies the necessary procedures to recover OP RF(s) from PEs running in fused/unfused mode and (ii) use the resulting OP RF(s) to predict the PE RF(s) with different fusion scenarios. We studied the resource profiling for major SPADE built-in operators and presented several techniques to overcome measurement errors from SPADE OP data collection. The impact of hardware speed and multi-threading contention are also studied. We show that our method can be applied to several SPADE applications and the prediction of the PE RFs is on the average within 15% of the actual CPU usage fractions from runtime PE measurement.

1. INTRODUCTION

As the world becomes ever more information-centric, we are entering an era in which it is necessary to process large volumes of heterogeneous data in near-realtime, in order to make effective decisions and maintain a competitive advantage. Traditional offline-based models of information processing and decision support are not effective here, and there has been an increasing interest in systems that process data "on-the-fly", also known as stream processing systems. In these systems, data is seen as arriving in continuous flows (streams), such as stock and options trading data in financial systems, environmental sensor readings, satellite data in astronomy, and network traffic state or statistics. Key considerations in such systems are performance, scalability, and efficient use of available resources.

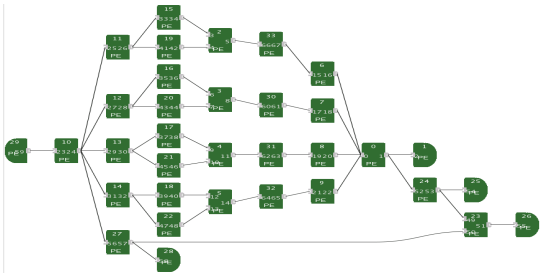


Figure 1: An example PE processing graph.

This paper is about building quantitative resource models of streaming computations. Our work is in the context of the System S distributed streaming platform [3, 12, 20, 11, 19] and the SPADE [9, 4] application development environment. In System S, a streaming application is composed of one or more jobs, each of which is organized as a data flow graph with processing elements (PE) as the nodes and data streams between the PEs as directed edges in the graph. Each stream carries data in the form of typed *tuples*. Data may be exchanged between PEs from any jobs, even across applications. An example is shown in Figure 1. PEs execute inside a PE *container* (PEC) which is an operating system process. A PE consists of one or more operating system threads that carry out the processing logic of that PE. These PEs are deployed onto the physical nodes of a distributed compute cluster, which is shared among multiple applications, and they are managed by the System S runtime.

The first important motivation for building quantitative resource models is to provide a key input for dynamic intelligent resource management. These resource management decisions in System S are performed by the optimizing scheduler SODA [19] which dynamically determines which jobs are admitted into the system, the placement of admitted PEs onto the nodes, and the share of node resources received by a PE over the time according to the demand of streams. These allocations must respect a number of user-provided constraints such as restricting PEs to a subset of nodes, license availability, and memory footprint, while simultaneously making effective use of available resources without overloading individual nodes or network links. At its core, this is a highly complex bin packing and flow balance problem, and multiple heuristic techniques are applied to meet the deadline for realtime load balancing. Knowing accurate "size" of the PEs is critical to make the right resource management decisions.

A second motivation arises from the development environment SPADE, which takes a code generation approach to facilitate the development of efficient, scalable streaming applications. In this approach, the developer composes an application in the SPADE language by using building blocks known as *operators* (OP), which are simpler, finer-grained

computations than PEs. SPADE comes with a set of built-in operators (mostly providing relational algebra operations in a streaming context) and also allows the flexible use of user-defined operators. Similar to the PE dataflow graph, the operators are organized in a *logical* dataflow graph. The SPADE compiler assembles the physical PE-level graph from this logical operator-level graph, through a process called *fusion*, where multiple operators are combined to form a PE. A key decision is to decide how many, and which operators must be fused together. For example, on a small cluster of powerful nodes it is desirable to fuse more operators into less PEs of larger size, whereas on a larger cluster of weaker nodes, it should preferably to have smaller PEs. The SPADE compiler allows a developer to automatically generate the appropriate optimized code in either scenario without rewriting or refactoring their application. To do this well, it needs to know the "size" of both operators and PEs.

There are two levels of monitoring infrastructure available in System S for collecting usage metrics, on which we can base the resource models. First, the System S runtime provides PE and PEC level CPU monitoring information, which is identical to the OS-level thread information available from Linux `top` command or `/proc` filesystem. It also collects information on the data tuples flowing in and out of PEs. Second, SPADE provides a profiling mechanism that inserts instrumentation points and collects metrics at the operator level. These metrics include statistics on the CPU resources consumed and the data tuples processed (size and rate) at the operator level.

We describe the resource usage models as *resource functions* (*RF*), which are described in more detail in Section 2.1. In this paper, we address two inter-related questions. First, using the metrics collected from possibly fused operators, how can we obtain consistent, reusable operator *RF*s? By *consistent*, we mean that the resource model should be the same regardless of how the operator is fused with other operators (which may introduce interference in the measurements). By *reusable*, we mean that the resource model is useful for predicting the outcome of a fusion. We refer to such an *RF* as the *baseline* OP *RF*, and the process as OP *RF* recovery. Second, how to compose the PE *RF* for a PE given its constituent operators and their baseline *RF*s? This is called PE *RF* prediction. We find that in addition to "adding up" the constituent *RF*s, it is necessary to factor in issues such as multi-threading and contention.

Our contributions include a unified framework for SPADE and SODA to obtain and use both operator- and PE-level resource models, integrated in the System S infrastructure. We show that the existing SPADE instrumentation may yield inaccurate OP *RF*s due to some approximations performed by the profiling system to overcome the limitations of the underlying OS APIs. We analyze the problem and find a way to "patch" these inaccurate *RF*s as a post processing step that does not require changes to the infrastructure or introduce additional overhead. Our method includes an approach to profiling the communication overhead of PEs, which is used for both OP *RF* recovery and PE *RF* prediction. We evaluate our methodology on several common built-in SPADE operators and show the effectiveness of the solutions. While our results are encouraging, we face limitations in terms of handling complex PEs and operators, and we discuss the challenges posed by those structures. However, our initial approach can already handle many operators and produce useful results for our resource allocation engine to optimize many practical streaming applications.

Table 1: Definitions

u	CPU usage fraction (cpuFrac) of a PE or OP.
u^r	CPU usage fraction overhead of a PE/OP input port.
u^s	CPU usage fraction overhead of a PE/OP output port.
r^r	tuple data rate (bps) of a PE/OP input port.
$\mathbf{r}^r$	a vector of rates for a set of ports.
t^r	tuple count rate (nps) of a PE/OP input port.
$\mathbf{t}^r$	a vector of rates for a set of ports.
r^s	tuple data rate (bps) of a PE/OP output port.
$\mathbf{r}^s$	a vector of rates for a set of ports.
t^s	tuple count rate (nps) of a PE/OP output port.
$\mathbf{t}^s$	a vector of rates for a set of ports.
c^r	CPU process time of a tuple in an OP
c^s	CPU submit time of a tuple in an OP
b	CPU process basecost of a thread
M	set of additional threads of a OP not driven by an input
I	set of input ports of a PE/OP
O	set of output ports of a PE/OP
K	set of OPs fused in a PE
f	CPU usage function in an *RF*
$\mathbf{g}$	rate function that specifies the relation of the input and output data rates of a PE/OP. $\mathbf{g}^i$ is ith vector function that corresponds to the ith output ports in a PE/RF.

The organization of the paper is as follows. Section 2 reviews the basic concepts of resource function, SPADE operator fusion and OP-level metric collection. Section 3 presents our OP *RF* modeling framework on System S and in it, we discuss the techniques that we use for each functional block. Two examples of OP *RF* recovery and PE *RF* prediction are show in Section 3.6. Section 4 discusses previous related works. Finally, Section 5 concludes the paper.

2. BACKGROUND

In this section, we review the concepts of resource function, SPADE operator fusion, and operator level metric collection. Table 1 summarizes the notations used in this paper.

2.1 PE Resource Functions

The resource demands of a PE are described to SODA in the form of *resource functions* (*RF*). Let u_e be the CPU usage fraction for PE e, $\mathbf{r}_e^r$ be the vector of input stream data rates (bytes per second) of the PE inputs and $\mathbf{r}_e^s$ be the vector of output stream data rates for the PE outputs. Then, the *RF* of PE e is defined in Equations 1-2 as:

$$u_e = f(\mathbf{r}_e^r) \quad (1) \qquad \mathbf{r}_e^s = \mathbf{g}(\mathbf{r}_e^r) \quad (2)$$

The functions f and $\mathbf{g}$ capture the effect of the input rates on the CPU usage u_e and the output data rates (respectively). While accurate PE *RF*s are crucial to the performance of the SODA scheduler, obtaining *RF*s is a challenge, not least because the PE logic can be arbitrary. For long-running PEs, it is conceivable to learn the *RF*s over time, and make better resource reallocation decisions as the learning improves. However, for new PEs, there is a bootstrapping question of how to obtain reasonable initial *RF*s to allow SODA to perform a good initial resource allocation. For this purpose,

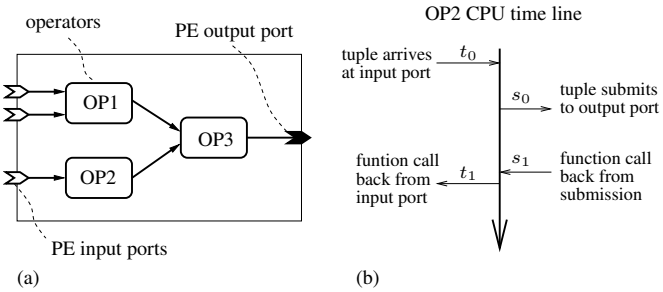


Figure 2: (a) An example fused PE. (b) Operator profiling statistics collection.

linear $f, \mathbf{g}$ are often sufficient. For linear PEs, the f is a scalar, $\mathbf{g}$ is a matrix.

2.2 Operator Fusion

SPADE [9, 4] is an extensible stream-oriented operator-based language, compiler and toolkit. The SPADE language provides a set of type-generic built-in operators and also allows users to define their own operators. In addition, the language allows the flexible composition of these operators into the logical data flow graphs representing the desired computation. The current built-in operator set is focused on providing relational algebra operations in a streaming context. Operators have zero or more input and output ports, where an output port produces a data stream in unit of tuples to the connected input ports of another operator. *Source* operators have no input ports (they are intended to obtain data from the external world, such as reading from network connections, disk, sensors, etc.), while *sink* operators do not have any output ports. They provide an output interface to the external world. An input port may also receive streams from multiple output ports. The SPADE compiler creates PEs and PE-level physical data flow graphs from the OP-level logical dataflow graphs, which are then deployed on the cluster.

The operation of combining some operators into a PE is called *fusion*. Figure 2(a) shows an example PE fused from three SPADE operators. The PE code is executed by one (or more) threads of the underlying operating system. The main PE thread waits for input on one of the input ports. (This input is provided when a tuple streams in from another PE). Upon receiving a tuple, the thread executes the intra-PE operator graph in a depth-first fashion. In the example, after receiving a tuple for OP1, the thread executes the OP1 code, then makes a function call to OP3. At this point, the OP3 code is executed, possibly resulting in output sent via the output port. At this point, the PE thread is now free to process the next input. Not all operators are single-threaded, multi-threaded implementations exist as well. Typically these threads are triggered by data arrival or synchronization events, but in all cases the sending of data to downstream operators occurs by a function call to that operator, with the appropriate parameters.

It is important to note that sending tuples across PE containers involves additional CPU cost for communication. Within one PE, tuples are passed by references and operators are fused by function calls. Therefore, fusing small operators together can save the overhead of unfused PEs.

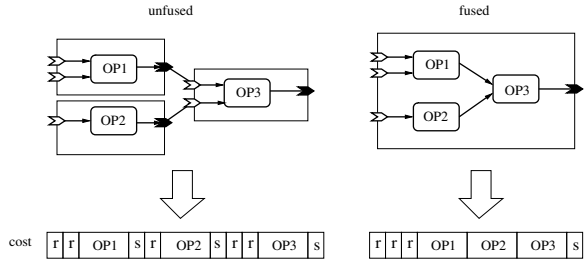


Figure 3: Fusion saves PE communication overhead.

Figure 3 shows that two unit of sending and receiving overheads are saved in fused PE for the OP graph in Figure 2(a).

2.3 Operator and PE Profiling

SPADE provides a profiling system [8] to collect various metrics on each individual operator that is contained within a PE. These metrics can be used by the SPADE compiler to make better PE fusion decisions. In addition, it may also enable us to obtain initial resource functions of fused PEs, which can help SODA to optimize PE runtime placement. The collection of an operator's resource profile is illustrated in Figure 2(b). The arrival of a tuple triggers a series of function calls, each corresponding to the entry into an operator's executable code. For each such operator function call, SPADE records the start time t_0 and completion time t_1. Each downstream operator is also a function call, which means we can obtain the total downstream operators' *submit time*, $s_1 - s_0$. Note that these times are in terms of the elapsed CPU time for the corresponding thread (vs. wall-clock time). SPADE also counts tuple receiving rate t^r in terms of the number of tuples received per second (nps) for each input port, and the tuple submission rate t^s for each output port. Finally, each operator thread that is not driven by input tuples contributes a *basecost* b. The total CPU usage fraction (cpuFrac) u of an operator k is computed as:

$$u_k = \sum_{i \in M_k} b_i + \sum_{i \in I_k} c_i^r t_i^r - \sum_{i \in O_k} c_i^s t_i^s \qquad (3)$$

where M_k is the set of additional threads, c^r is the average process time, c^s is the average submission time, I_k is the set of input ports, O_k is the set of output ports. Essentially, Equation 3 counts the net CPU process time by subtracting the portion used by downstream operators. Using the SPADE metrics, we can estimate the CPU resource usage of an operator for a given data set in terms of CPU fraction and tuple rate. However, as shown in Section 3.3.2, the SPADE instrumentation does not provide an accurate measure of OP *RF*s. While the SPADE profiling instrumentation is a compile time option to collect OP-level metrics, the System S runtime always collects and provides PE-level metrics. This uses the Linux built-in APIs to collect per-thread CPU usage information which allows us to determine per-PE CPU usage. In addition, the runtime also reports information on the dataflow, such as tuple rates and sizes.

3. METHODOLOGY

We now describe a unified methodology to tackle both problems of OP *RF* recovery as well as PE *RF* prediction, based on the System S and SPADE metrics. After an overview of all the components, we describe each step in the method-

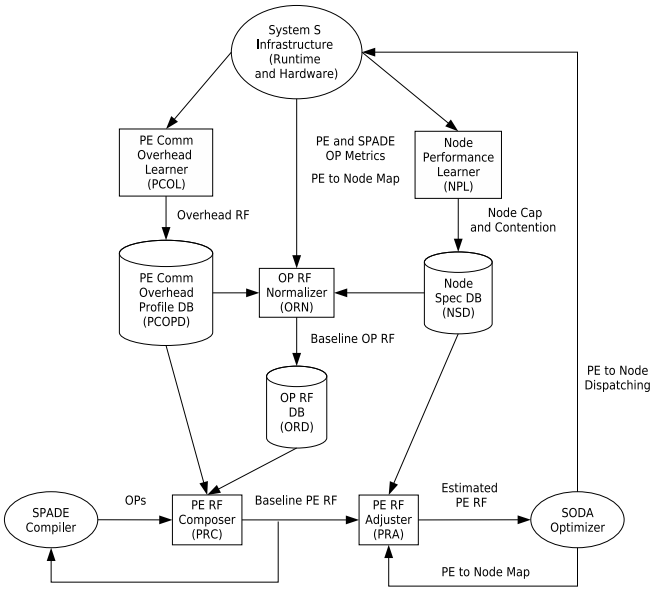


Figure 4: *RF* modeling infrastructure for System S.

ology in detail. Then we work through some illustrative examples of first recovering OP *RF*s and then using them to predict PE *RF*s which are fused from those learned operators.

Our methodology is shown in Figure 4 along with the information flows in the context of the existing System S components (depicted by oval). Here, the rectangular boxes represent the building blocks of our methodology. The cylindrical objects represent data repositories which are either populated or used by the various steps. The lines represent flow of information between the blocks, repositories and System S components.

The starting point of the *RF* modeling is the raw metrics reported by the System S infrastructure. The first main step of the OP *RF* recovery is performed by the *OP RF Normalizer* (ORN), and the resulting OP *RF*s are maintained in the *OP RF Database* (ORD) for reuse. The second principal issue of PE *RF* prediction is performed in the *PE RF Composer* (PRC), which is a component that can be used by either the SPADE compiler for PE fusion or by SODA for runtime scheduling. For both the ORN and PRC, it is necessary to separate communication and computation cost. The information on communication overhead is learned by the *PE Communication Overhead Learner* (PCOL).

Our cluster may contain heterogeneous nodes, which poses two subproblems. First, we must normalize any node-specific aspects so that the models in the ORD are generalizable across all the nodes. This normalization occurs based on information about nodes which is calibrated by the *Node Performance Learner* (NPL) and stored in the *Node Spec Database* (NSD). Second, when combining PEs to execute on a specific node, the node-specific information may be used to obtain a better prediction of the combined PE behavior. This latter adjustment is performed in the *PE RF Adjuster* (PRA) using information from the NSD.

To summarize, for fusion decisions, the SPADE compiler can obtain PE size estimates from the PRC by submitting the list of operators to fuse to the PRC. In this sense, PRC serves as a PE size "Oracle" for the fusion optimizer. For runtime decisions, SODA must examine various PE placement and fractional allocation options. The node-specific *RF*s for the PEs are obtained from the PRA, which in turn combines the baseline PE *RF* from the PRC and the node-specific information from NSD.

In this paper, we do not cover memory usage profiling because most PE/OPs in our applications are CPU-bound programs and the consideration of CPU resource is a more important factor. In System S information on PE memory usage can be manually supplied by the developer, and this information is used as a constraint by SODA to ensure no node is overallocated. Dynamic memory usage profiling and optimization are still under research and out of the scope of this paper.

3.1 Experiment Setup

First, we introduce our experimental environment and notational conventions that are used in the examples shown in the paper. All results were collected on three type of machines. Machine type 1 is an Intel Xeon 3G hyper-threading, 1M cache and 6G memory. Machine type 2 is an AMD Opteron 2.6G dual core, 1M cache and 8G memory. Machine type 3 is an Intel Xeon 3.4G hyper-threading. Most of the single PE results were collected from machine type 1. The SPADE profiling system were run at sampling ratio 0.01. The 98% confidence interval is the $\pm 6\%$ data range of CPU fraction and $\pm 1\%$ of I/O rates, respectively.

In this paper, we show four different types of SPADE applications: Regex, Aggregate(aggs/aggt), Join and VWAP. Their SPADE source codes are attached in the Appendix. Figure 6 illustrates the OP data flow graphs of each application. The name of the application for each OP graph is shown in the caption. The entire Regex application contains three functor OPs (Regex1, Regex2, and Regex 3), each of which performs some regular expression operations on each input tuple. Depending on the number of functors contained in an application, we have func1, func2, and func3. The two aggregate examples (aggs/aggt) share the same OP graph and almost identical logic except the types of aggregate windows used. The join example contains an operator with two input ports. VWAP is a larger example with two functors (TradeFilter, VWAPSum) and an aggregate (VWAPAggreg). All OPs we focus on in this paper are single-threaded. Except the source OP ($|M_k| = 1$) that has a driver thread of its own, all other OPs do not have basecost ($|M_k| = 0$). The CPU fraction of PE/OPs in our examples will be always a real number ranging from 0 to 1.

Figure 5 illustrates different PE fusion and node placement configurations for a certain OP graph. The connection of source and sink OPs (solid half circles) are also presented in the figures. The other boxes with OP marks are operators. Dashed boxes that groups one or more operators represent PEs. Solid boxes that contain PEs represent nodes that only run the PEs and Linux OS. An application could only use a certain fusion and placement configuration if its OP graph matches the exact OP graph shown in the configuration.

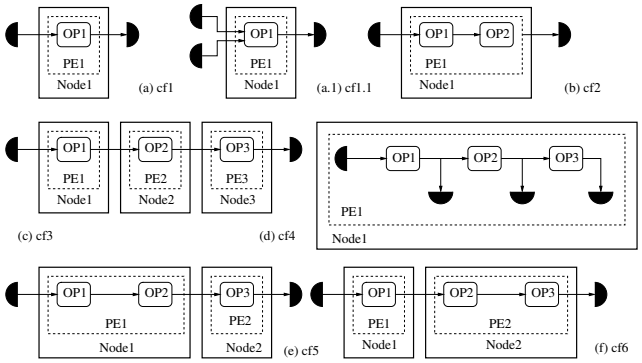


Figure 5: Example fusion and PE-to-node mapping scheme for operator graphs. (a) cf1, (a.1) cf1.1, (b) cf2, (c) cf3, (d) cf4, (e) cf5, (f) cf6

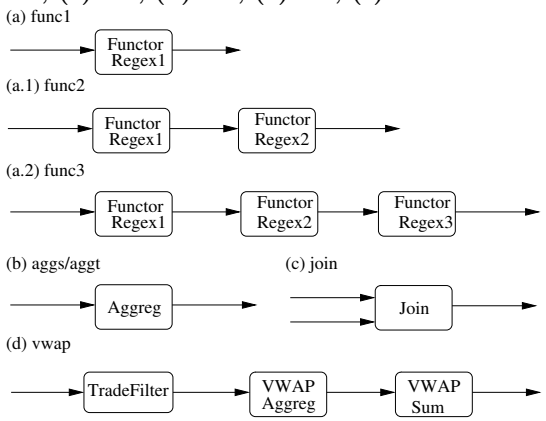


Figure 6: Example SPADE operator graphs. (a) func1, (a.1) func2, (a.2) func3, (b) aggs/aggt, (c) join, (d) vwap

For example, notation "func3-cf3" means the three functors in Regex (OP graph Figure 6(a.2)) are unfused and placed in the same way as the three operators in Figure 5(c). The same application, func3, can also be configured as in cf4, which fuses all OPs together with additional sinks. This application configuration is notated by "func3-cf4".

3.2 PE Communication Overhead Learner (PCOL)

As discussed in Section 2.2, the cost of receiving and sending tuples between PEs is an integral part of the overall PE resource usage, so it is important to obtain a good measure of his overhead. An earlier study [8] has established that this overhead grows with increasing tuple sizes and rates. The PCOL component learns this functional dependency of the overhead on tuple sizes and rates. It operates on metrics collected by running a special calibration application consisting of one source operator connected to one sink operator, each in its own PE on separate nodes, as shown in Figure 8. The source operator is a cheap tuple generator that does no other processing on the tuple, and the sink simply discards the received tuple. Thus, the CPU time for the PE is almost all spent on the communication. We run the application using variable tuple sizes m and obtain the resulting maximum tuple data rate $r_{max}(m)$, source PE CPU fraction $u_{src}(m)$ and sink PE CPU fraction $u_{sink}(m)$. Figure 7 shows these metrics for machine types 1 and 2. For these machines, we see that when the tuple size is small, the source PE is the bottleneck and uses 100% CPU. As tuple size increases, the TCP network interface becomes the bottleneck. For a given PE input port with measured input data rate r^r and tuple rate t^r, we can estimate the input port CPU overhead $u^r(r^r, t^r)$ using Equation 4 and our overhead profiling data. In the same way we can compute the output port overhead by Equation 5.

$$u^r(r^r, t^r) = u_{sink}(\frac{r^r}{t^r})\frac{r^r}{r_{max}(\frac{r^r}{t^r})} \quad (4)$$

$$u^s(r^s, t^s) = u_{src}(\frac{r^s}{t^s})\frac{r^s}{r_{max}(\frac{r^s}{t^s})} \quad (5)$$

3.3 OP RF Normalizer (ORN)

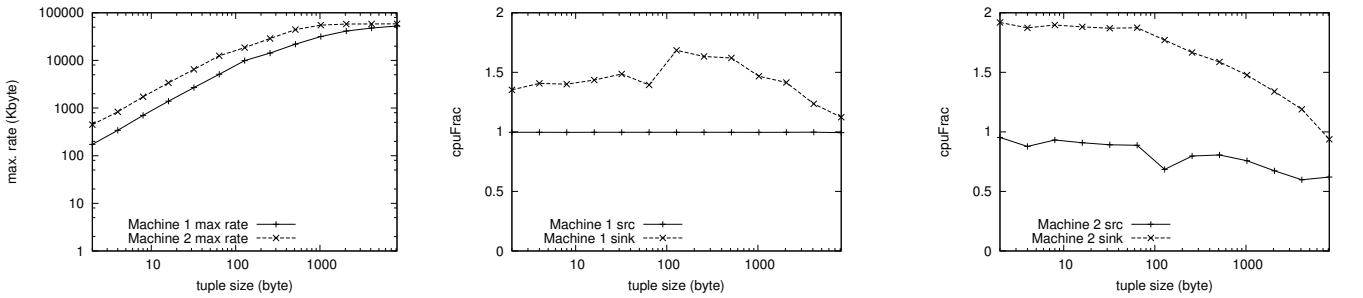
This step tackles the construction of the baseline OP *RF* using data from the System S runtime and the SPADE profiling infrastructure. The *RF*s are stored in the ORD for easy access by other components. We first discuss our formulation of the OP *RF*s. Next, we show that the SPADE profiling introduces some undesirable approximation errors, but that these errors can be corrected by post-processing in certain cases. We illustrate this approach by applying it to some example applications.

3.3.1 Operator RFs

In general, the OP *RF*s take the same form as the PE *RF*s presented in Section 2.1, capturing the input rate-dependent aspect of the operator's resource usage. There are other factors such as the operator parameterization (e.g., window size for the Join operator) that can affect the resource usage. Rather than model the effect of such parameters in the *RF*, we treat each different parameterization of an operator as a distinct operator. This may result in a larger set of operator models, but on the other hand, it is a simpler approach that does not require much modeling of the use and semantics of that parameter in that operator's algorithm.

For many OPs, CPU usage metrics can show non-linear effects when the load on a processor approaches its limit, even when the actual OP *RF* is linear. In this paper, we ignore such effects. Most operators in our study have CPU *RF*s which are linear in their tuple input rate with a few exceptions, such as a join OP with time-based windows on each input port. The output rate *RF* $\mathbf{g}$ is related to the probability of a tuple being filtered at each input port and the change of tuple size between an input and output tuple of the OP. For single input and output linear operators, $\mathbf{g}$ is simply a scalar g.

The training data for building the OP *RF*s is obtained by running the application at a range of source rates. It is feasible that the OP *RF* data is dependent on the data content as well. At this time, we do not model the dependency of the OP *RF* on the input data distribution. It is not always possible to obtain representative input data at development or compile time. The design point of System S is that learning OP and initial PE *RF*s is a starting point, which is better than making fusion or initial SODA placement decisions without any information at all. The SODA scheduler is explicitly designed to respond to by dynamic and responds to



(a) Maximum data rate

(b) CPU fraction at maximum rate on machine type 1

(c) CPU fraction at maximum rate on machine type 2

Figure 7: PE communication overhead profiling.

changes in incoming data or PE behavior by updating its *RF*s for already running PEs based on new observed data.

3.3.2 SPADE OP-Level Profiling and Inaccuracy

The training data for OP-level *RF* construction is collected by the SPADE profiling infrastructure mentioned earlier. It provides OP-level resource utilization metrics, including the CPU time t_c used by each operator. The only source of CPU usage information is the underlying OS, and in our case the native Linux OS maintains CPU usage information at a 10-millisecond precision. This is a critical limitation, since the CPU time spent on processing a single tuple arriving on an input port of the operator could potentially be at a nanosecond scale. Hence, most of the measurements of process time and submit time in Figure 2(b) will be zero. To work around this limitation, SPADE uses the following approximation. Instead of the CPU time, it measures the elapsed time t_e (based on the CPU cycle counter), which is available at a nanosecond resolution (for modern CPUs which operate at GHz frequency). Thus, the raw process time and submit time are actually in terms of the elapsed time. To convert t_e it into the CPU time t_c, these times are scaled by the average OS thread-level CPU utilization u (including both user and system time charged to the process) during the previous 500 milliseconds, to obtain an estimate $\hat{t}_c = t_e \times u$. However, because u includes activity from all the component operators of the PE, as well as the tuple reading and writing, the resultant $\hat{t}_c$ is not necessarily representative of the actual t_c. This causes an inaccuracy in the operator's CPU usage measurement.

Specifically, CPU-bound operators may be under-estimated because their CPU utilization are very likely to be higher than the average utilization of the whole thread. Analogously, I/O-bound operators may be over-estimated because their CPU utilization are likely lower than the thread-wide average. For example, Figure 10 shows the CPU usage fraction of the Regex1 operator in various applications (func1, func2, func3) and configurations (cf1, cf2, cf3, cf4) using Equation 3 and based on the scaled CPU time $\hat{t}_c$. Refer to Section 3.1 for a detailed explanation of the application configurations. In this simple application, all the input rates equal the output rates in terms of number of tuples along the chain. The maximum rate where each curve ends is the saturated rate for the corresponding application running in a specific resource configuration. func3-cf4 cannot sustain as high a saturated rate as func3-cf3 because cf4 fuses all operators into one PE with additional sinks and uses only one node. For comparison purposes, we also show the CPU utilization based on raw elapsed times t_e, which is denoted by the legends marked with "-rt". Regex1 is a functor with a regular expression match logic (see the Appendix for source code) that does not contain any blocking function calls. Since the PE that contains Regex1 also runs on an empty node[1], the CPU should be able to be dedicated to the PE thread while the tuple is being processed in Regex1, so that there is no context switch during the tuple processing. Therefore, the wall-clock result should be an accurate measure of the CPU usage time, and we see that the SPADE profiling result is an under-estimate of the true value for all input rates below the saturated rate. The reason is that the PE thread will block to wait for the arrival of a new tuple when the application is running at a lower rate than the maximum sustainable rate given the processing capability of the node. The blocking time at PE input ports reduces the average CPU utilization of the whole thread, which is smaller than 100%. In this case, $\hat{t}_c < t_c$.

Figure 11 shows that Regex2 exhibits the same problem. Unlike Regex1, the $\hat{t}_c$ for Regex2 in func3-cf3 is not accurate even at the saturated input rate, because Regex2 is not the bottleneck operator in the application (Regex1 is the bottleneck) so even when the application throughput is saturated, the PE that contains Regex2 still needs to wait. However, when Regex1 and Regex2 are fused together in the func2 configuration, the measurement is more accurate because Regex2 is invoked only when there is data to process. The wall-clock time measurement in Figures 10 and 11 also verify that these functors should have linear *RF*s.

There are some approaches that may improve CPU time measurement precision but are not suitable for System S. One way is to use kernel APIs to access hardware counters, such as PAPI [5] to acquire a better measurement. It requires installation of external kernel patches on Linux, which is problematic for the Enterprise versions of the OS, which preclude such actions. Moreover, the SPADE profiling is designed to minimize the impact on system performance. An alternate way is suggested by our Regex1 example – use the elapsed time measure. However, in general, a thread can be context switched during tuple processing, or block due to re-

[1]the Linux OS daemon threads use negligible CPU resources and are most likely scheduled on another context in our multi-process machines

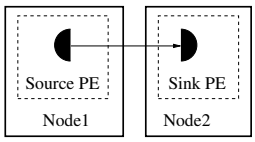


Figure 8: Experiment setup to profile a computing node.

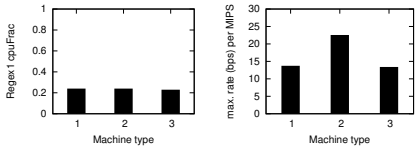


Figure 9: Comparison of node capacity on running func3-cf4.

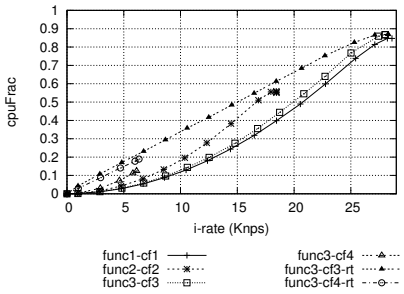


Figure 10: Comparison of the measured OP *RF* of Regex1 in different configurations on machine type 1.

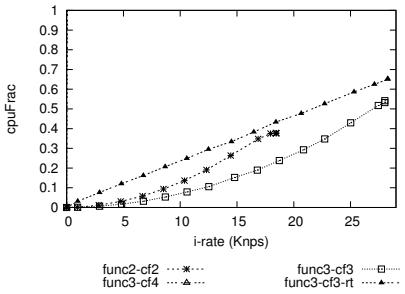


Figure 11: Comparison of the measured OP *RF* of Regex2 in different configurations on machine type 1.

source sharing and synchronization, which will erroneously inflate the measured in-operator elapsed time, causing it to deviate from the on-CPU time. Since it is hard to differentiate variance caused by data processing variance from that caused by blocking, it may be even harder to correct elapsed time measurements for the general case.

3.3.3 OP RF Recovery

In this paper, we are interested in two *RF*s for each operator: one for CPU, and another for the output rates. The operator metrics for input and output tuple counts and rates are not subject to the measurement error, so it is possible to obtain the output rate *RF* based on the SPADE profiling metrics. As mentioned above, we assume linear *RF*s, which are obtained from the raw metrics data using a linear regression that goes through the origin.

For the CPU *RF*s, given the inaccuracy in the OP-level CPU metrics, we formulate a two-pronged strategy. First, for an operator which is unfused with others (i.e., it is in a PE by itself) it is possible to use the PE-level metrics to recover the OP-level *RF*. A procedure to do this recovery is shown in Algorithm 1. We can estimate the PE's communication overhead via the PCOL information and subtract it from the PE's CPU usage fraction to obtain the OP's computational CPU usage. The functional *RF* forms are obtained from this data using a least-squares fit using the lowest order polynomial form that provides good fit. More advanced statistical techniques may be used as well, although we have not yet found it necessary in practice. For applications where it is possible to deploy each operator in its own PE, this approach can be used, and does not even use the SPADE profiling metrics (beyond the PCOL information).

For applications with hundreds or thousands of operators, it may not be possible to even deploy or start the application unless the operators are first fused into a more manageable number of PEs. For such operators, the PE level metrics are not very useful. Hence, we must rely on the OP-specific metrics collected by the SPADE profiling mechanism. The challenge here is whether the measurement errors introduced by the profiling mechanism can be corrected. This brings us to the second part of our strategy.

We begin from the observation that at saturation, the SPADE measure will accurately reflect the CPU usage. Hence, in the case of linear *RF*s, we can interpolate between the system performance at this saturation point and the origin to recover the *RF*. Here, saturation refers to the maximum rate at which the PE can run on this node without other constraints. It is *not* the maximum ingest rate of the system, which may be limited by other bottleneck PEs. For some PEs, the saturated point is "virtual" if they are not the bottleneck PEs. Regex2 in Figure 11 is such an example. Functor Regex2 only uses 70% CPU at the maximum throughput of the application. Regex1 is the processing bottleneck in this case.

Our approach combines both the PE-level metrics and the SPADE profiling metrics, and is shown in Algorithm 2. We first obtain the operator-specific input rate at which the containing PE is saturated. For this, we first obtain a functional relationship $u = f_{e,k}(r)$ between the operator's input rate r_k^r and the PE CPU usage data u_e (step 4). This function is interpolated or projected to find the input rate $\tilde{r}$ where the PE is saturated, i.e., $f_{e,k}(\tilde{r}) = 1$ (step 5). Then, we use that operator's SPADE profiling metrics (step 7) to find the lower-order polynomial $u = \tilde{f}_k(r_k^r)$ that best describes the OP-specific data. This operator's correct CPU utilization at the saturated point is given by $\tilde{f}_k(\tilde{r})$ (step 8). Finally, the operator's linear *RF* is the line between $(0, 0)$ to $(\tilde{r}, \tilde{f}_k(\tilde{r}))$ (step 9). This approach works well for linear *RF* operators that are single threaded, non-blocking, and have a single input and output port. Examples include functors and punctors in SPADE. Since functors are usually small operators and are heavily used in most streaming applications for basic data manipulation, such as data filtering, transformation and computation, it is worthwhile to study the fusion case specifically targeted at functor-like operators. Our correction method may also work for some single-thread blocking operators if the error is in an acceptable range. To illustrate the case, if an operator consumes 60% of the real time at 80% CPU utilization and the rest of time is non-blocking (so 100% utilization for that part), the average CPU utilization measured will be $0.6 \times 0.8 + 0.4 = 0.88$, which is used by SPADE to approximate the real OP CPU utilization that is 80%. Thus, the SPADE measure will have 10% error when it is used to compute the CPU fraction for that OP.

3.3.4 OP RF Recovery Examples

Let us consider some examples for the recovery of OP *RF*s using our algorithms. All the operators we have studied are

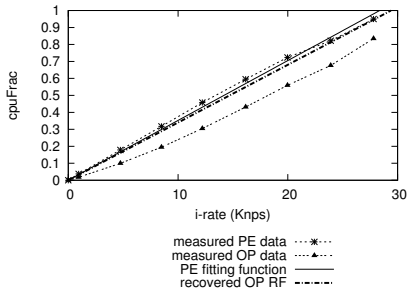


Figure 12: OP *RF* recovered from the unfused PE in aggs-cf1, on machine type 1

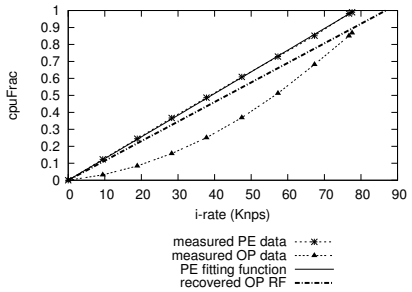


Figure 13: OP *RF* recovered from the unfused PE in aggt-cf1, on machine type 1

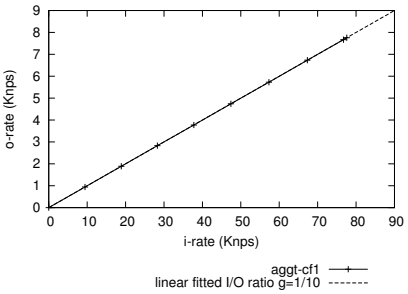


Figure 14: Input and output rates for aggt-cf1, on machine type 1.

Algorithm 1 Recover OP RF $(f, \mathbf{g})$ from unfused PE.

1: Run the PE application at different source rates.
2: For all rates, read PE CPU fraction u_e from Linux; Read all input ports data rate vector $\mathbf{r}^r$, tuple rate $\mathbf{t}^r$, and output ports data rate $\mathbf{r}^s$, tuple rate $\mathbf{t}^s$ from SPADE.
3: Find the least square polynomial fitting function f_e for data tuples $(\mathbf{t}^r, u_e)$.
4: Find rate function $\mathbf{g}$ for I/O tuple rate $(\mathbf{t}^r, \mathbf{t}^s)$ using least square polynomial fitting for each output port.
5: Compute the overhead function $\sum_{i\in I} u^r(r_i^r, t_i^r)$ + $\sum_{i\in O} u^s(\mathbf{g}^i(\mathbf{r}^r)^T, \mathbf{g}^i(\mathbf{t}^r)^T)$ using Equations 4-5 and get the OP RF by $f = f_e - (u^r + u^s)$.

Algorithm 2 Recover OP RF (f, g) from fused PE.

1: Run the PE application at different source rates.
2: For all rates, read PE CPU fraction u_e from Linux. Read each fused OP k's CPU fraction u_k, input port data rates r_k^r, tuple rates t_k^r, and output ports data rates r_k^t, tuple rates t_k^t from SPADE.
3: **for** each operator $k \in K_e$ **do**
4: Find the least square linear fitting function $f_{e,k}$ for data tuples (r_k^r, u_e).
5: Compute the saturated rate $\tilde{r}$ where $f_{e,k}(\tilde{r}) = 1$.
6: Compute rate RF as $g = \frac{t_k^s}{t_k^r}$.
7: Find the least square quadratic fitting function $\tilde{f}_k$ for data tuples (t^r, u_k).
8: Compute the saturated rate point $(\tilde{r}, \tilde{f}_k(\tilde{r}))$.
9: Recover the OP CPU RF as $f_k = \frac{\tilde{f}_k(\tilde{r})}{\tilde{r}} r_k^r$
10: **end for**

mostly CPU intensive, so their original SPADE OP measurements are under estimated. We show the recovery of aggregate and join operators RFs; the demonstration for fused functors will be presented in Section 3.6.1.

As its name suggests, an aggregate operator combines each newly arrived tuple with previously arrived tuples in its window according to some user defined logic and emits the result of the aggregation. Here we show two examples of the aggregate OP with different window types: *aggs* uses a sliding window of size 10 and step 1; *aggt* uses a tumbling window of size 10. The source code is provided in the Appendix. Both application have the same OP graph, shown in Figure 6(b). aggs outputs one tuple per arrival at steady state so the input and output rates are the same. Since the OP is unfused in its PE, we can use Algorithm 1 for recovering its RF. Figure 12 shows the quantities computed by the steps in the algorithm. aggt outputs one tuple every 10 arrivals. Figure 14 verifies that the I/O tuple rates measured by SPADE is precisely 10:1. Figure 13 illustrates the same recovery technique for aggt as for aggs.

The recovery of OP RF for multi-port operators can also be performed using Algorithm 1. Join operator is a built-in multi-port operator with 2 inputs and 1 output. Figures 15 and 17 present the recovery of the join RF with one input that is configured with a 30-slot sliding window and the other input that is configured with a 15-slot sliding window. The SPADE OP measurements for join operators also underestimate the actual values. Figure 16 shows the computation of rate function $\mathbf{g}$ for the multi-input case. We see that the I/O ratio for this application is still linear.

3.4 PE RF Composer (PRC)

PRC composes baseline PE RF from learned OP RFs. PRC is called by SPADE to estimate fused PE RFs at compile-time. The composed baseline PE RFs are also used by SODA to project PE RFs at runtime for resource balancing. PE RF composition for a fused PE e consists of two steps: one is to construct I/O rate function $\mathbf{g}_e$ given the functions $\mathbf{g}_k$ from each fused operator $k \in K_e$. The other is to compute the CPU usage fraction function f_e given f_k of each fused operator. For the first step, the vector function $\mathbf{g}_e^i$ can be computed backwards for each output port i. For the example of a fused PE in Figure 2(a), given the function $\mathbf{g}_1 : r_1^s = a_1 r_1^r + a_2 r_2^r$ for OP1, $\mathbf{g}_2 : r_2^s = b r_3^r$ for OP2, and $\mathbf{g}_3 : r^s = c_1 r_1^s + c_2 r_2^s$ for OP3, the function for the PE is thereby $\mathbf{g}_e : r^s = c_1 a_1 r_1^r + c_1 a_2 r_2^r + c_2 b r_3^r$. The same approach can be applied on non-linear rate functions or loop topology where the output To compute the function f, it simply sums all fused OP RFs and the communication overhead. Equation 6 shows the composition.

$$f_e(\mathbf{r}^r, \mathbf{t}^r) = \sum_{k\in K_e} f_k(\mathbf{r}_k^r) + \sum_{i\in I_e} u^r(r_i^r, t_i^r) + \sum_{i\in O_e} u^s(\mathbf{g}_e^i(\mathbf{r}^r)^T, \mathbf{g}_e^i(\mathbf{t}^r)^T) \quad (6)$$

3.5 Node Performance Learner (NPL) and PE RF Adjuster (PRA)

NPL is responsible for profiling the computing node capacity as well as the CPU contention due to multi-threading. The results from NPL are used by PRA to adjust the baseline

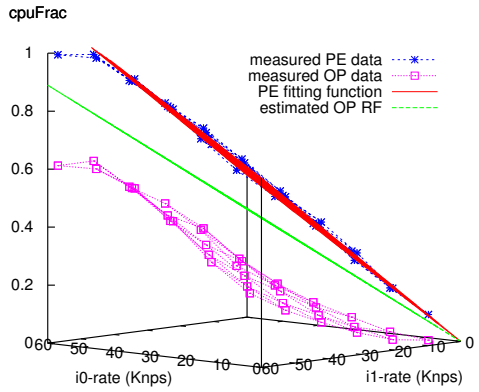


Figure 15: Join OP *RF* recovered from the unfused PE in join-cf1.1, on machine type 2

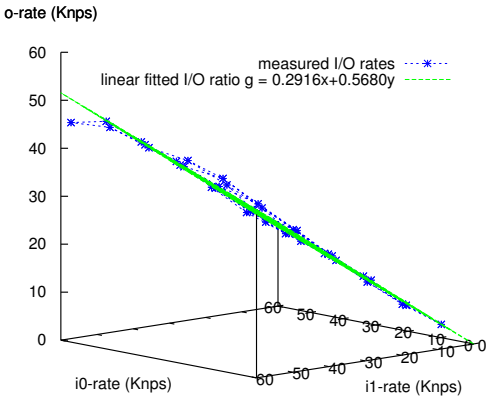


Figure 16: Join OP rate function recovered from the unfused PE in join-cf1.1, on machine type 2

PE *RF*s to the actual run-time environment. Furthermore, the results are used by ORN to normalize the learned OP *RF*s to the baseline OP *RF*s. However, the general problem of node capacity modelling is a hard one. This is because the execution time of a program varies greatly over different hardware. Besides CPU frequency, micro-architecture plays a role in how fast a program can run. Multiple levels of caches and look-up tables in the system increases the unpredictability of program execution time. Another important factor is the CPU contention on multi-threaded systems. To make things worse, CPU contention can vary depending on micro-architecture and operating system-level load balancing.

A challenge in modeling the CPU demands of a program is that this demand will vary depending on the specific CPU being used. In order to construct a general model that can be used across all nodes in a heterogeneous cluster, SODA uses *MIPS* as a measure of machine capacity. The *MIPS* used here is based on the processor Bogomips [18] reported by the Linux kernel and adjusted for the multi-context run-time environment. However, our experiences indicate that Bogomips is not a reliable measure of machine capacity and we could achieve better accuracy by applying domain specific capacity profiling techniques. Accordingly, we propose to use the maximum throughput achieved from running certain streaming mico-benchmarks, instead of CPU MIPS, to model the node capacity. The node specification database (NSD) saves the maximum data rate for each machine type based on the results from the streaming micro-benchmarks. This information is later used by PRA and ORN. In PRA, the input rates of the PE *RF* are multiplied by the normalized maximum data rate of the machine the PE will be placed on. In ORN, the input rates of the OP *RF* are divided by the normalized maximum data rate of the machine that the PE is running on.

Designing benchmarks for different types of streaming applications is still an on-going work. Figure 9 shows a comparison of three machine types using func3-cf4 as our micro-benchmark. *MIPS* for machine type 1 is 11773.4, machine type 2 is 9946.59, and machine type 3 is 13057.4. Since Bogomips is a metric loosely related to CPU frequency, the type-2 node has the lowest MIPS count. First, we observe that the SPADE measured CPU fraction for Regex1 is equal on all three machine types. This is because when the PE is taking 100% of the CPU resources, the splitting of CPU usage amongst the fused operators is the same for different machine types. We also observe that the maximum data processing rate (bytes per second) per MIPS is not the same for certain pairs of machine types. A type-2 node is able to process more data per MIPS than the other two types. We know that type-1 and 3 nodes have the same Intel architecture with different CPU speeds but type-2 nodes have the AMD architecture. From this study, we found that the *MIPS* metrics that SODA have been using to model node capacity may be useful for the same machine architecture with different CPU speeds. However, it can be misleading for machines with different architectures.

Placing multiple PEs on the same node/core may affect the performance if they are sharing caches, memories, and other resources. Our results show that running two PEs on the hyper-threading machines (type-1) will affect the *RF*s, but multi-core machines (type 2) do not show such contention. Figure 21 shows the PE CPU fraction measurement of func1-cf1 on machine type 1 that has 4 processors. Figure 25 shows the results on machine type 2 with dual-core processors. "func1-cf1-b" is the result when the PE is running on one processor and the other three processors are each pinned with a 100% CPU load program. Compared to the results from an idle machine, the CPU usage of the PE increased almost by 50% for processing at the same rate. Legend notation "-zx" means that all four processors are pinned with a program using 0.x fraction of the CPU. The program maintains a busy loop that wakes up periodically and writes to 10M of memory space. To simulate the worst case contention, the program uses 10M residential memory space and makes sure to clear the processor cache of the CPU during context switch. We adjust CPU utilization of the program by varying the sleeping period. Whichever processor the PE is scheduled, it always shares resources with our contention thread. On a type-2 machine, we see that increasing CPU demand of the other process does not change the PE *RF*. In this case, PE *RF* adjustment is not needed. However, sharing multiple threads on the hyper-threading node will affect the PE *RF* (Figure 21). The contention observed from hyper-threading node may be caused by increasing cache misses on level-2 caches and stall cycles [6]. An analytical model for hardware context switching [16] has suggested that the number of threads that a CPU can support with linear growth of

performance is limited. In this paper, we suggest to use an idle machine or dual-core machines with at most two threads on the same processor for accurate measurement. The study of load contention in the general case is left as future work.

3.6 Demonstration

In this section, we present two simple applications to illustrate our OP *RF* recovery and PE *RF* estimation algorithms. The PE *RF*s that are estimated from the recovered OP *RF*s will be compared against the real PE measurements.

3.6.1 Regex Application

Recall the func2-cf2 application, where operators Regex1 and Regex2 are fused and running on a single node. Figure 18 shows the OP *RF*s of Regex1 and Regex2, recovered from a fused PE using our OP *RF* recovery algorithms. Concretely, using Algorithm 2, a curve is fitted to the PE measurements and the saturated rate point is projected as (18.16,1). Then, additional curves are fitted to the OP measurements and the OP *RF*s are recovered by plugging the saturated rate point into these curves.

Now we show that our recovered OP *RF*s from func2-cf2 can be used to predict the PE *RF*s of unfused Regex1 and Regex2 operators in func3-cf3. Figure 22 shows the estimated PE *RF* of Regex1 using Equation 6. Similarly, Figure 26 shows the estimated PE *RF* of Regex2. The measured PE *RF* from running func3-cf3 is plotted for comparison. Throughout the range of rates up to the saturated rate, the difference between our prediction and the actual measurement is smaller than 5% for Regex1. For Regex2, the error stays within 10% for most part of the comparison, until we reach the small region covering higher rates where non-linear effects are observed.

3.6.2 VWAP Application

We now study a larger application that contains functor and aggregate operators. Unlike the previous example, in this one some of the operators are performing data filtering and data reduction depending on predefined conditions. The example, named as VWAP, is part of a financial trading application and consists of three operators: TradeFilter (a functor), VWAPAggreg (an aggregate), and VWAPSum (a functor). TradeFilter filters out tuples that do not represent trading activity (such as quotes). VWAPAggreg finds the maximum/minimum of the trading prices on a sliding window of size 4 and step 1. VWAPSum performs arithmetic operations on tuple data fields to create a volume weighted average price.

Figures 19, 23, and 27 show the OP *RF*s recovered from an unfused configuration running on a type-2 machine. All OP *RF*s are normalized using the source rate for ease of comparison. Figure 20 shows the output tuple rate of each operator relative to the source rate. Figure 24 shows the predicted and actual PE *RF*s for the fused PE containing TradeFilter and VWAPAggreg operators. Figure 28 shows the predicted and actual PE *RF*s for the fused PE containing VWAPAggreg and VWAPSum operators. For TradeFilter and VWAPAggreg combined, our prediction exactly matches the actual PE measurements throughout the full range of input rates. For combined VWAPAggreg and VWAPSum, our prediction over-estimates by at most 15% CPU fraction at the highest rate.

4. RELATED WORK

Studying the performance of parallel programs on multi-core systems is receiving growing interest as multi-core processors become prevalent. Performance studies using hardware counters on simultaneous multi-threading (SMT) systems can be found in [6]. Their results showed that hyper-threading contention accounts for an average 69% increase of level-2 cache misses and 1.5 times more stall cycles for some benchmarks. On the contrary, multi-core execution context did not contribute to performance loss in most cases. SPADE measurement of multi-threading contention agrees with their main results. Further study is still needed to model the impact of multi-threading/multi-core contention on the resource function of independent threads with varying data sets. The architecture of Intel multi-core processors and Linux SMP schedulers were discussions in [17, 1]. Using queuing theory, [2] provided a deterministic model to estimate the executing time for a parallel program on a symmetry multi-processor system. However, the parameters that were used in the model are still hard to estimate in our system.

Many efforts have been made to improve the throughput of streaming systems by using data and task parallelism. StreamIT [13] is a stream processing system built mainly for signal processing. [10] explored the potential of parallelism on dataflow graphs. MapReduce technique introduced by [7] provided another programming model to process large amounts of static data (pre-existing on disks) with implicit parallelism. System S is explicitly parallelized (the processing graph is modified explicitly when more PEs are allocated), and we are also working on implicit parallelism functionality for real-time data flows. Our approach in this paper focuses on learning and predicting the resource consumption of each operator and processing element for varying data rates.

Some profiling tools have been developed to measure run-time resource consumption for general-purpose software programs. The authors in [15] compared existing performance analysis tools, and PAPI [5] provided a cross-platform interface for software programs to use performance hardware counters. Profiling and optimization for executing general programs on a single machine has been extensively studied in the past. An operating system level profiling and execution optimization tool was introduced in [21] to improve the execution for programs on a single machine. [14] proposed a run-time optimization system with hardware-driven profiling system. In SPADE, we try to profile our streaming program in a distributed execution environment without the aid of additional kernel patches.

5. CONCLUSION

In this paper, we have outlined a first empirical approach to constructing quantitative resource usage models for basic operators in streaming systems. The dataflow architecture suggests the use of rate-driven resource models for both CPU and I/O rates, which we have found to be effective in practice for a variety of streaming operators. The first main chal-

lenges addressed here is constructing normalized, reusable operator *RF*s, wherein the node-specific information is suitably factored out of the collected metrics to yield *RF*s that can be reused for predicting that OP's resource usage in other scenarios. The second challenge addressed is about composing these *RF*s into predictions on *RF*s for fused PEs. These PE-level *RF*s are utilized both for compile-time fusion optimizations as well as runtime resource allocation optimizations.

One aspect of our approach is to specifically tackle the inaccuracy in the SPADE OP-level profiling metrics for fused operators. We also presented a general technique to recover OP *RF*s from unfused PEs for those OPs that cannot be recovered accurately in fused form using SPADE metrics. Our two-pronged approach effectively increases the efficiency and accuracy of OP and PE resource profiling. In an end-to-end application of our approach on real SPADE applications, we first obtain OP *RF*s, then obtain estimated PE *RF*s from fusing those OPs. We find that the PE *RF* are within 15% CPU fraction compared to actual measurements from the fused PE. From a methodology perspective, we find that additional contention introduced by a multi-threaded machine may require additional modeling, whereas multi-core machines exhibit less interference and so are easier to handle.

This paper presents an initial attempt to tackle a hard and complex problem. We believe it warrants further investigation into tackling more complex fused PEs and an even greater diversity of operators. One specific area is dealing with multi-threaded operators, especially in fused configurations. From the hardware perspective, accounting for additional contention on multi-threaded processors or having a large number of active threads on a multicore is an interesting open question.

6. REFERENCES

[1] J. Aas. *Understanding the Linux 2.6.8.1 CPU Scheduler*. Silicon Graphics, Inc. (SGI), Feb. 2005.

[2] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.*, 22(1):94–136, 2004.

[3] 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 *DMSSP '06*, pages 27–37, New York, NY, USA, 2006. ACM.

[4] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu. On optimizing the aggregation/join architectural pattern for high-performance data stream processing. In submission, 2008.

[5] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14:189–204, 2000.

[6] M. Curtis-Maury. *Improving the Efficiency of Parallel Applications on Multithreaded and Multicore Systems*. PhD thesis, Virginia Tech, Mar. 2008.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[8] B. Gedik, H. Andrade, and K.-L. Wu. A code generation approach to optimizing high-performance distributed data stream processing. In submission, 2008.

[9] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *SIGMOD '08*, pages 1123–1134, New York, NY, USA, 2008. ACM.

[10] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII*, pages 151–162, New York, NY, USA, 2006. ACM.

[11] G. Jacques-Silva, J. Challenger, L. Degenaro, J. Giles, and R. Wagle. Towards autonomic fault recovery in system-s. In *ICAC '07*, page 31, Washington, DC, USA, 2007. IEEE Computer Society.

[12] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *SIGMOD '06*, pages 431–442, New York, NY, USA, 2006. ACM.

[13] A. A. Lamb. *Linear Analysis and Optimization of Stream Programs*. PhD thesis, Massachusetts Institute of Technology, May 2003.

[14] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA '99*, pages 136–147, Washington, DC, USA, 1999. IEEE Computer Society.

[15] S. Moore, D. Cronk, K. S. London, and J. Dongarra. Review of performance analysis tools for mpi parallel programs. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 241–248, London, UK, 2001. Springer-Verlag.

[16] R. H. Saavedra-barrera, D. E. Culler, and T. V. Eicken. Eicken. analysis of multithreaded architectures for parallel computing. In *In Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1990.

[17] S. Siddha. Multi-core and linux kernel. Intel Inc., 2007.

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

[19] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, and K.-L. Wu. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware '08*, Dec. 2008.

[20] K.-L. Wu, P. S. Yu, B. Gedik, K. Hildrum, C. C. Aggarwal, E. Bouillet, W. Fan, D. 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 *VLDB*, pages 1185–1196, 2007.

[21] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. *SIGOPS Oper. Syst. Rev.*, 31(5):15–26, 1997.

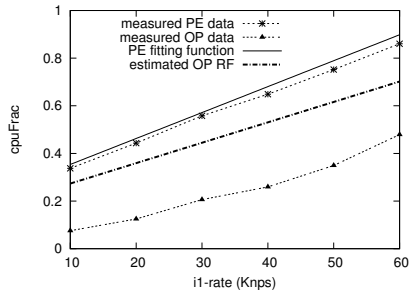


Figure 17: The 2-D slicing of Figure 15 at input 1 rate of 30Knps

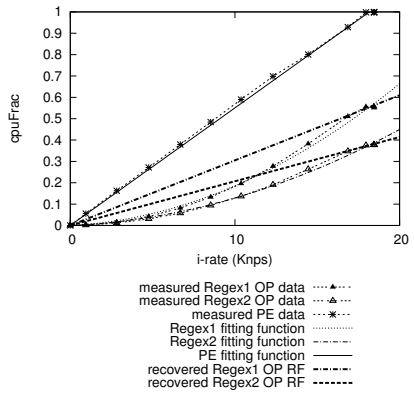


Figure 18: Regex1 and Regex2 OP *RF*s recovered from the fused PE in func2-cf2, on machine type 1

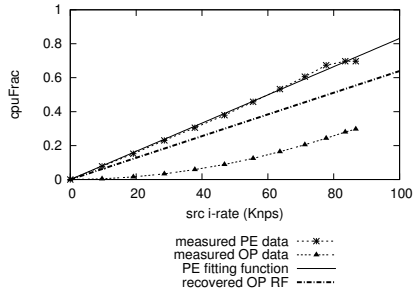


Figure 19: TradeFilter OP *RF* recovered from the unfused PE in vwap-cf3, on machine type 2

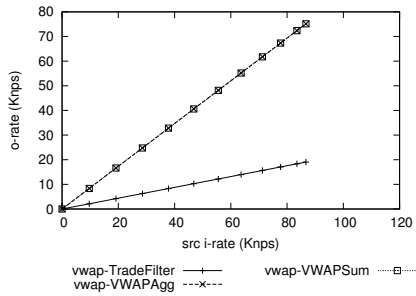


Figure 20: I/O rate ratios for VWAP OP

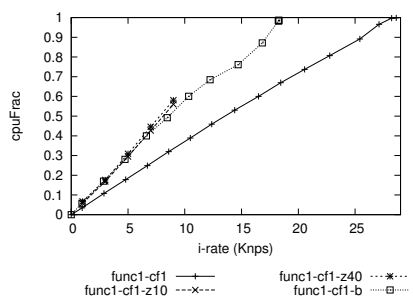


Figure 21: CPU utilization for the PE in func1-cf1, on machine type 1 and in a multi-threaded environment

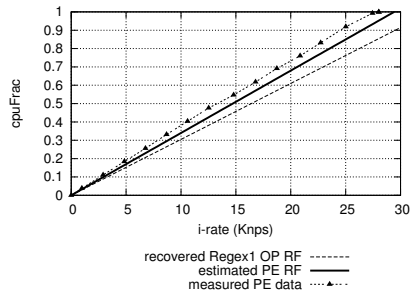


Figure 22: Estimated PE *RF* for unfused Regex1 in func3-cf3, on machine type 1

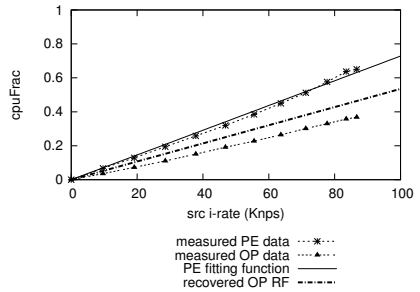


Figure 23: VWAPAggreg OP *RF* recovered from the unfused PE in vwap-cf3, on machine type 2

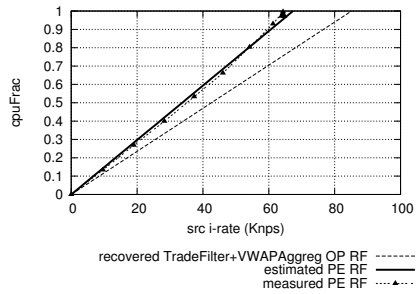


Figure 24: Estimated PE RF for fused TradeFilter and VWAPAggreg in vwap-cf5, on machine type 2

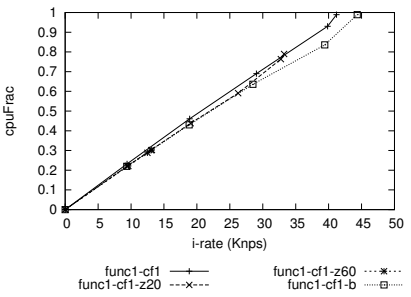


Figure 25: CPU utilization for the PE in func1-cf1 on machine type 2, in a multi-threaded environment.

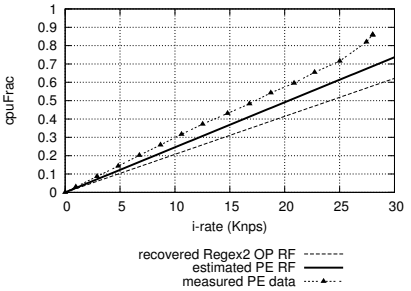


Figure 26: Estimated PE *RF* for unfused Regex2 in func3-cf3, on machine type 1

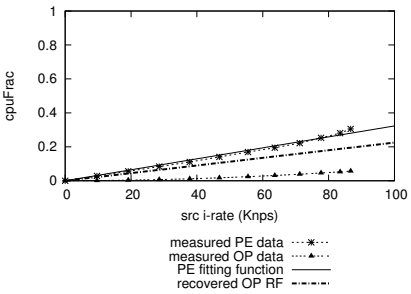


Figure 27: VWAPSum OP *RF* recovered from the unfused PE in vwap-cf3, on machine type 2

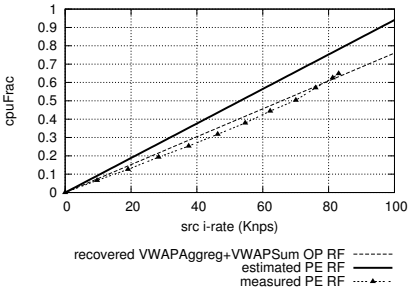


Figure 28: Estimated PE *RF* for fused VWAPAggreg and VWAP-Sum in vwap-cf6, on machine type 2

APPENDIX

A. SPADE APPLICATION SOURCE CODE

A.1 func3-cf4

```
[Application]
regex

[Typedefs]
typespace regex

[Nodepools]
nodepool pool[4] := ()

[Program]
stream Source1(dateTime:String)
    := Source()["file:SourceData.dat",nodelays,
    csvformat,throttledRate=1000]{}
-> partition["pe0"], node(pool, 0)

stream Regex1(dateTime:StringList)
    := Functor(Source1)[]
{ regexMatch(dateTime, "([0-9]*)-([0-9]*)-([0-9]*) (.*)") }
-> partition["pe1"], node(pool, 1)

stream Regex2(date:String, time:String, seq:Long)
    := Functor(Regex1)[]
{ dateTime[3]+"-"+
  select(toInteger(dateTime[2])-1,
  "JAN", "FEB", "MAR", "APR", "MAY", "JUN",
  "JUL", "AUG", "SEP", "OCT", "NOV", "DEC")+
  "-"+dateTime[1], dateTime[4], seqNum()
}
-> partition["pe2"], node(pool, 2)

stream Regex3(schemaFor(Regex2))
    := Functor(Regex2)[]
{  date, regexReplace(time,"00","0",true), seq }
-> partition["pe3"], node(pool, 3)

stream DummySink(schemaFor(Regex3)) := Functor(Regex3)[false] {}
-> partition["pe0"], node(pool, 0)
```

A.2 aggs-cf1

```
[Application]
aggregator

[Typedefs]
typespace aggregator

[Nodepools]
nodepool pool[2] := ()

[Program]
vstream stockReportStream(
   symbol       : String,
   dateTime     : String,
   closingPrice : Float,
   volume       : Integer)

vstream aggregatedData(
   symbol       : String,
   recordCnt    : Integer,
   totalTuples  : Integer,
   minPrice     : Float,
   maxPrice     : Float,
   avgPrice     : Float,
   minVolume    : Integer,
   maxVolume    : Integer)

stream Source1(schemaFor(stockReportStream))
   := Source()["file:stock_report.dat",
      csvformat,throttledRate=1000] {1, 2, 3-4}
-> partition["pe0"], node(pool, 0)

stream Aggreg1 (schemaFor(aggregatedData))
   := Aggregate(Source1 <count(10),count(1), pergroup>)
      [symbol]
  {Any(symbol), Cnt(), MCnt(), Min(closingPrice), Max(closingPrice),
   Avg(closingPrice), Min(volume), Max(volume)}
-> partition["pe1"], node(pool, 1)

stream Sink1(schemaFor(aggregatedData)) := Functor(Aggreg1)[false] {}
-> partition["pe0"], node(pool, 0)
```

A.3 aggt-cf1

```
[Application]
aggregator

[Typedefs]
typespace aggregator

[Nodepools]
nodepool pool[2] := ()

[Program]
vstream stockReportStream(
   symbol       : String,
   dateTime     : String,
   closingPrice : Float,
   volume       : Integer)

vstream aggregatedData(
   symbol       : String,
   recordCnt    : Integer,
   totalTuples  : Integer,
   minPrice     : Float,
   maxPrice     : Float,
   avgPrice     : Float,
   minVolume    : Integer,
   maxVolume    : Integer)

stream Source1(schemaFor(stockReportStream))
   := Source()["file:stock_report.dat",
   csvformat,throttledRate=1000] {1, 2, 3-4}
-> partition["pe0"], node(pool, 0)

stream Aggreg1 (schemaFor(aggregatedData))
   := Aggregate(Source1 <count(10), pergroup>) [symbol]
   {Any(symbol), Cnt(), MCnt(), Min(closingPrice), Max(closingPrice),
   Avg(closingPrice), Min(volume), Max(volume)}
-> partition["pe1"], node(pool, 1)

stream Sink1(schemaFor(aggregatedData)) := Functor(Aggreg1)[false] {}
-> partition["pe0"], node(pool, 0)
```

A.4 join-cf1.1

```
[Nodepools]
nodepool pool[2] := ()

[Program]

stream Source1(
   bidderName : String,
   productName : String,
   bidPrice : Float)
:= Source()["file:auction_bid.dat", csvformat, throttledRate=10000]
{}
-> partition["pe0"], node(pool, 0)

stream Source2(
   productName : String,
   offerPrice : Float)
:= Source()["file:product_match.dat", csvformat, throttledRate=5000]
{}
-> partition["pe0"], node(pool, 0)

stream Join1(
   bidderName : String,
   productName : String,
   bidPrice : Float,
   matchingPrice: Float)
:= Join(Source1 <count(30)>; Source2 <count(15)>)
   [$1.productName = $2.productName & $2.offerPrice <= $1.bidPrice ]
   {$1.bidderName, $2.productName, $1.bidPrice, $2.offerPrice}
-> partition["pe1"], node(pool, 1)

stream Sink1(
   bidderName : String)
 := Functor(Join1) [false]{}
-> partition["pe0"], node(pool, 0)
```

A.5 vwap-cf3

```
[Application]
vwap

[Typedefs]
typespace vwap

[Nodepools]
nodepool pool[4] := ()

[Program]

stream TradeQuote(
  ticker   : String,
  date     : String,
  time     : String,
  ttype    : String,
  price    : Double,
  volume   : Double,
  vwap     : Double,
  askprice : Double,
  asksize  : Double)
    := Source()["file:TradesAndQuotes.csv.long",
                nodelays, csvformat, throttledRate=10000]
         {1-3, 5, 7-9, 11, 15-16}
-> partition["pe0"], node(pool, 0)

stream TradeFilter (
    date       : StringList,
    timestamp  : Long,
    ticker     : String,
    ttype      : String,
    price      : Double,
    volume     : Double,
    myvwap     : Double,
    vwap       : Double)
      := Functor(TradeQuote)[ ttype="Trade"]
          { regexMatch(date, "([0-9]*)-([A-Z]*)-([0-9]*)"),
  timeStampToMicroseconds(date,time),
            ticker, ttype, price, volume,
            price*volume, vwap }
-> partition["pe2"], node(pool, 1)

stream VWAPAggregator (
    ticker    : String,
    cnt       : Integer,
    minprice  : Double,
    maxprice  : Double,
    avgprice  : Double,
    svwap     : Double,
    svolume   : Double)
      := Aggregate(TradeFilter < count(4), count(1) >)[ticker]
           { Any(ticker), Cnt(ticker), Min(price), Max(price), Avg(price),
             Sum(myvwap), Sum(volume) }
-> partition["pe3"], node(pool, 2)

stream VWAPSum (
    cminprice  : Double,
    cmaxprice  : Double,
    cavgprice  : Double,
    cvwap      : Double)
      := Functor(VWAPAggregator)[true]
          { minprice*100.0d, maxprice*100.0d, avgprice*100.0d,
            (svwap/svolume)*100.0d }
-> partition["pe4"], node(pool, 3)

 stream DummySink (
    cminprice  : Double,
    cmaxprice  : Double,
    cavgprice  : Double,
    cvwap      : Double)
  := Functor(VWAPSum)[false]{}
-> partition["pe1"], node(pool, 0)
```