Workload Characterization for Operator-Based Distributed Stream Processing Applications

Xiaolan J. Zhang
University of Illinois, Urbana–Champaign, IL, USA
xzhang29@crhc.illinois.edu

Sujay Parekh
IBM T.J. Watson Research Center, Hawthorne, NY, USA
sujay@us.ibm.com

Bugra Gedik
IBM T.J. Watson Research Center, Hawthorne, NY, USA
bgedik@us.ibm.com

Henrique Andrade
IBM T.J. Watson Research Center, Hawthorne, NY, USA
hcma@us.ibm.com

Kun-Lung Wu
IBM T.J. Watson Research Center, Hawthorne, NY, USA
klwu@us.ibm.com

ABSTRACT

Operator-based programming languages provide an effective development model for large scale stream processing applications. A stream processing application consists of many runtime deployable software processing elements (PE) that work in flows to process incoming messages. Operators (OP) are logical building blocks hosted by PEs. One or more OPs can be fused into a PE at compile-time. Performance optimization for our streaming system includes compile-time fusion optimization and runtime PE-to-host deployment. One of the goals of an optimized stream application is to use minimal computing resource to sustain maximal message throughput.

Characterizing the resource usage of PEs is critical for performance optimization. During compile-time optimization, OP-level resource usage is used to predict the resource usage of fused PEs. When starting an application, PE-level resource usage is used as an initial estimation by the scheduler. In this paper, we propose an efficient workload characterization approach for data stream processing systems. Our method includes the procedures for obtaining reusable OP-level resource usage information from profiling data and recomposing OP-level profiles to predict PE-level resource usage. We present several techniques to overcome measurement errors from the OP data collection. The impact of hardware speed and multi-threading contention on hyper-threading and multi-core machines are also studied. We show that our method can be applied to several streaming applications and the prediction of the PE CPU resource usage is within 15% of the actual CPU usage.

Keywords: stream processing system, workload characterization, profiling

DEBS'10, July 12-15, 2010, Cambridge, UK.

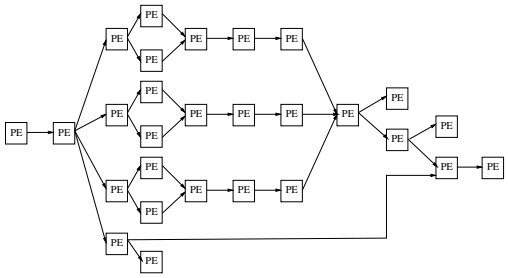


Figure 1: An example stream processing application.

1. INTRODUCTION

There has been increasing interest in systems that can process large volumes of heterogeneous data in near-realtime. 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 data. Examples of such stream processing systems include NiagaraCQ [12], Borealis [6], TelegraphCQ [11], STREAM [9] and System S [7]. In order to be able to keep up with the input data rates, performance, scalability, and efficient use of available resources are key considerations in such systems.

This paper is about characterizing and predicting the resource usage of stream processing applications deployed in a heterogeneous cluster environment. Our work is in the context of the System S distributed streaming platform [7, 18] where a streaming application 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*. An example is shown in Figure 1. 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.

Stream processing applications in System S are developed using the SPADE [8, 16] composition language, which allows the application to be constructed by combining sim-

pler, reusable building blocks known as operators. SPADE comes with a set of built-in operators (mostly to provide relational algebra operations in a streaming context) and also allows for the implementation of user-defined operators to extend the language. Similar to the PE dataflow graph, the operators are organized in a *logical* dataflow graph. The SPADE compiler automatically assembles the physical PE-level graph from this OP-level graph, through a process called *fusion*, where multiple operators are combined to form a PE. While fusion eliminates inter-operator communication overhead, it also increases the amount of processing to be done by a single PE. For optimal performance, the fusion decisions must carefully balance the overhead reduction with the increase in processing requirements, and match it to the available capacity on processing nodes in a cluster. These decisions require the ability to predict PE resource usage given the constituent operators.

A second motivation for our work is to provide a key input for runtime resource management of streaming applications. These resource management decisions in System S are performed by the SODA optimizing scheduler [27] which dynamically determines the placement of PEs from submitted jobs onto the cluster nodes, and the share of each node's resources received by each PE according to the demand of streams. These allocations must respect computation, communication, and memory constraints imposed by the cluster hardware. At its core, this is a combination of bin packing and flow balance problems. To make good decisions, the scheduler must be able to accurately predict the "size" of the PEs (in terms of their resource usage). Although the scheduler is capable of learning a PE's resource usage behavior as it executes, it still needs PE size estimates when an application is first submitted to the system. The same resource usage information used by the SPADE compiler can be used for this purpose.

The contribution in this paper includes a unified framework for obtaining and using operator- and PE-level resource models for predicting resource usage for both the compiler and runtime scheduler. The resource usage models for both PEs and OPs are called *resource functions* (*RF*) and are described in more detail in Section 2.2. Two main problems are tackled in our solution: how to obtain these OP-level *RF*s, and how exactly they should be combined to yield the final PE-level *RF*s. For the OP-level *RF*s, we aim at constructing consistent, reusable OP *RF*s from PE- and OP-level profiling data collected by System S on running streaming applications. By *consistent*, we mean that the resource model should be the same regardless of how the operator is fused with other operators during the profiling step (which may introduce interference in the measurements). By *reusable*, we mean that the resource model is useful for predicting the outcome of another, arbitrary fusion involving that operator. We refer to such OP *RF* as the *baseline* OP *RF*, and the process as OP *RF* recovery. Obtaining such *RF*s is a challenge because the source metrics are obtained in a specific fusion configuration running on a specific hardware. We show that a naïve approach to constructing resource models from the existing SPADE instrumentation may yield inaccurate OP *RF*s due to the approximations performed by the profiling system. We propose a way to "patch" these inaccurate *RF*s as a post processing step that does not require changes to the existing infrastructure or introduce additional overhead. On the problem of PE *RF* prediction, in addition to "adding up" the constituent OP baseline *RF*s, we show that it is necessary to factor in issues such as multi-threading and contention.

Table 1: Definitions

u	CPU usage fraction (cpuFrac) of a PE/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 (bytes per sec.) of a PE/OP input port.
$\mathbf{r}^r$	Vector of rates for a set of ports.
t^r	Tuple count rate (tuples per sec.) of a PE/OP input port.
$\mathbf{t}^r$	Vector of rates for a set of ports.
r^s	Tuple data rate (bytes per sec.) of a PE/OP output port.
$\mathbf{r}^s$	Vector of rates for a set of ports.
t^s	Tuple count rate (tuples per sec.) of a PE/OP output port.
$\mathbf{t}^s$	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 an 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 function that specifies the relation of CPU usage and the input data rates of a PE/OP.
$\mathbf{g}$	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/OP.

We present an evaluation of our approach using *functors* and *aggregate*[1] streaming operators. As we strive to find a simple possible method to use for computing potentially large workloads, we face limitations in terms of handling complex PEs and operators, and we discuss them in detail.

Our methodology addresses a general workload characterization problem for distributed streaming processing systems. Although many previous continuous streaming query systems run on a single-server, such as NiagaraCQ [12], TelegraphCQ [11], and STREAM [9], today's systems are designed for the capability of scaling up the computation on many computing nodes to support high data-rate processing. Borealis [6] supports distributed execution of query operators and runtime load balancing but not compiler-time optimization like the "fusion" supported by System S. Our approach has the potential to apply to Borealis- and System S-like distributed systems.

The organization of the paper is as follows. Section 2 introduces the basic concepts of resource function, SPADE operator fusion and OP-level metric collection. Section 3 presents our OP workload characterization framework. An use case from a stock analysis application is shown in Section 4. Section 5 discusses some of the previous related work. Finally, Section 6 concludes the paper.

2. BASIC CONCEPTS

We introduce the concept of operator fusion, resource functions, and SPADE profiling in this section. Table 1 summarizes all the notations that are used in this paper.

[1]Functors are operators used to apply functions to stream tuples, e.g., filtering, projection, arithmetic transformations, etc. Aggregates are operators used for carrying out group-by aggregations.

2.1 Operator Fusion

System S applications are developed using a stream-oriented operator-based language, compiler and toolkit, called SPADE. The SPADE language provides a set of type-generic built-in operators, such as *functor, aggregate, join, sort, barrier, punctor*, and also allows users to define their own operators. Operators have zero or more input and output ports, where an output port produces a data stream that can be connected to input ports of another operator(s). The data stream consists of a flow of discrete data packets, called tuples. *Source* operators have no input ports (they are designed 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.

The SPADE compiler creates PE-level physical data flow graphs that can be deployed on the runtime environment from the OP-level logical dataflow graphs. The operation of combining some operators into a deployable PE is called *fusion* [15]. 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 tuple input on one of the input ports. 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 tuple. The next tuple can be an arrival at OP2. New arrivals are blocked when the PE is busy with processing previously arrived tuples. 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 via a function call to that operator, with the appropriate parameters.

Fusing a few communicating OPs into a PE can save computing resources. For example, sending tuples across PE containers involves additional CPU cost for communication, such as marshaling and unmarshaling the data. Within a PE, tuples are passed by reference and invocation of fused operators is just a function call. Thus, fusing small operators together can eliminate significant overhead relative to placing each operator in its own PE. Figure 3 shows that two units of sending and receiving overheads are saved in the fused PE for the OP graph in Figure 2(a).

2.2 PE and Operator Resource Functions

The workload a PE or OP is subjected to is characterized by a *resource function* (*RF*). Let u be the CPU usage fraction for a PE or OP, $\mathbf{r}^r$ be the vector of input stream data rates (bytes per second) of the input ports and $\mathbf{r}^s$ be the vector of output stream data rates for the output ports. Then, the *RF* is defined by Equations 1-2. The functions f and $\mathbf{g}$ capture the effect of the input rates on the CPU usage u and the output data rates respectively. For PEs or OPs where the workload is linear to the input rates, f is a scalar, $\mathbf{g}$ is a matrix.

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

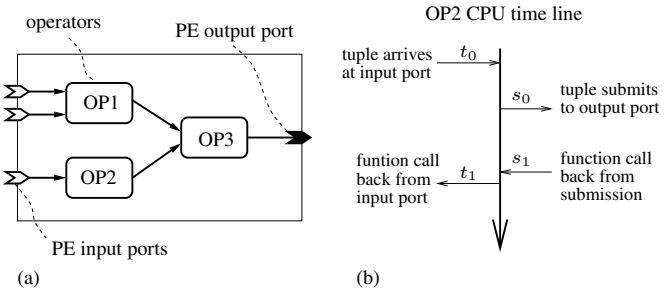


Figure 2: (a) An example fused PE. (b) Operator profiling statistics collection.

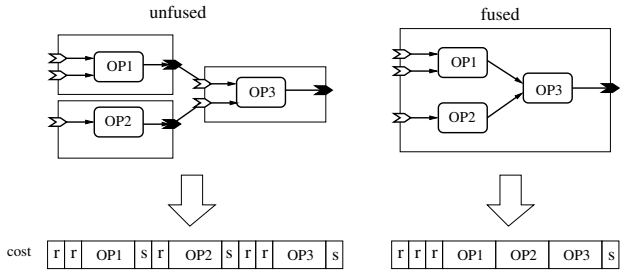


Figure 3: Fusion saves PE communication overhead. "r" is the receiving overhead. "s" is the sending overhead.

While accurate PE *RF*s are crucial to the performance of the scheduler, obtaining *RF*s is a challenge, because in general the PE logic can be arbitrary. For long-running PEs, it is conceivable to learn the *RF*s over time and make better resource allocation decisions as the learning improves. However, for new PEs, there is a bootstrapping question of how to obtain reasonable initial *RF*s to allow the scheduler to perform a good initial resource allocation. In this paper, we try to predict PE *RF* using previously measured OP *RF*.

We only consider linear *RF* as the majority of our current applications are composed mainly of linear operators or operators that can be approximated using linear functions. We find that using linear models for communication and computation overheads can provide good performance and the model is simple enough to compute for potentially large workloads.

Generally, we find that memory utilization is not directly related to data rates. While characterizing PE memory usage is important, the profiling infrastructure currently does not provide memory usage information. Developing effective memory models is ongoing work.

2.3 Operator- and PE-Level Profiling

SPADE provides a profiling system [15] 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, and enable us to obtain initial resource functions of fused PEs for optimized 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 *submit time*, $s_1 - s_0$, of all downstream operators. Note that these times are in terms of the elapsed CPU time for the corresponding thread (as opposed to wall-clock time). The profiling infrastructure of SPADE keeps track of the tuple receiving rate t^r in terms of the number of tuples received per second (tps) 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 to a *basecost* b. The total CPU usage fraction (cpuFrac) u of an operator k is computed as:

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

where M_k is the set of additional threads used by an operator, c^r is the average process time, c^s is the average submission time, I_k is the set of operator input ports, O_k is the set of operator output ports. Essentially, Equation 3 counts the net CPU process time by subtracting the portion used by downstream operators. Using the profiling 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, this estimate can be inaccurate, as discussed in Section 3.3.

An additional source of metrics is the System S runtime which collects PE-level metrics. This uses the Linux built-in `/proc` system to collect per-thread CPU usage information, which allows us to determine per-PE CPU usage. The runtime also reports information on inter-PE dataflow, such as tuple rates and their sizes.

3. METHODOLOGY

We now describe a unified methodology to tackle both problems of OP *RF* recovery and PE *RF* prediction based on the System S and SPADE metrics, focusing on CPU usage. Our methodology is shown in Figure 4 along with the information flows in the context of the existing System S components (depicted by ovals). 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 OP workload characterization is to collect raw metrics from the System S infrastructure. Since the OP data is obtained with different fusion configurations, we need to process the data so that it can be reused in other configurations. Therefore, 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. To reuse the data (for PE *RF* prediction), normalized OP *RF*s are combined at *PE RF Composer* (PRC), which is a component that can be used by either the SPADE compiler for PE fusion or by the scheduler 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

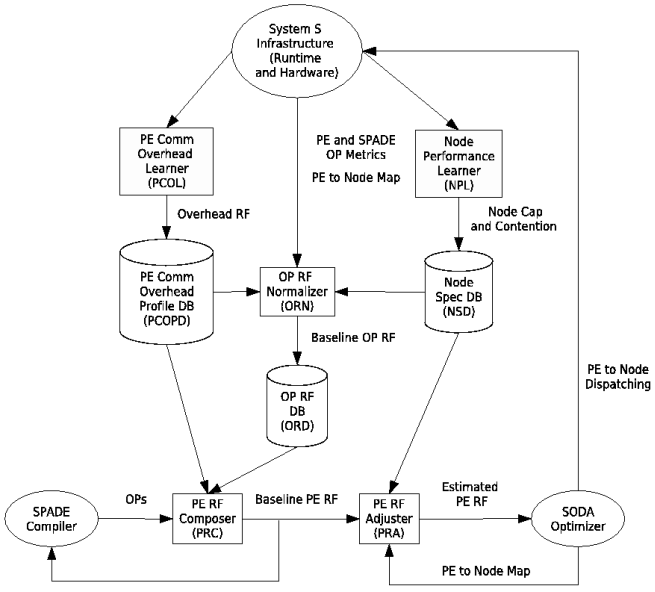


Figure 4: OP workload characterization infrastructure for System S.

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, when making 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, the scheduler 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 the next section, we will take a look at how this methodology can be used in practice.

3.1 Experiment Setup

First, we introduce our experimental environment and notational conventions used in the examples shown in the paper. All results were collected on three types of machines. Machine type-1 has an Intel Xeon 3GHz hyper-threading CPU, 1MB cache, and 6GB memory. Machine type-2 is an AMD Opteron 2.6G dual core CPU, 1MB cache, and 8GB memory. Machine type-3 is an Intel Xeon 3.4GHz hyper-threading CPU, 1MB cache, and 6GB memory. By default, the data is collected from machine type-1. The SPADE profiling system was run at a sampling ratio of 0.01 (sampling 1 tuple in every 100 tuples). The 98% confidence interval for the same training data is within ±6% of CPU fraction values and ±1% of I/O rates, respectively.

To illustrate the basic ideas, we use a simple application called Regex that performs streaming regular expression matching on its input data. Later, a combined functor and aggregate application named VWAP (volume weighted average price) is shown as a testing benchmark and the code is shown in the appendix. VWAP can be considered a subgraph of a more complete stock market analysis application [8]. Figure 5 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 Regex3), each of which performs regular expression operations on each input tuple. Depending on the number of functor OPs contained in an application, we get three versions of the same application: func1, func2, and func3. VWAP is a combined example with two functors (TradeFilter, VWAPSum) and an aggregate (VWAPAggreg). All OPs we focus on in this paper are single-threaded, except for the source OP that has a driver thread of its own. None of the other OPs has a basecost. The CPU fraction of PE/OPs in our examples is a real number ranging from 0 to 1.

Figure 6 illustrates several different PE fusion and node placement configurations for a particular OP graph. The boxes with OP labels are operators. The source and sink OPs (solid half circles) are also presented in the figures. Dashed boxes that group one or more operators represent PEs. Solid boxes that contain PEs represent nodes that only run the PEs and the typical daemons used by a regular Linux OS installation. An application uses a certain fusion and placement configuration as specified by the OP graph in the configuration. For example, notation "func3-cf3" means the three functors in Regex (OP graph Figure 5(a.2)) are unfused and placed in the same way as the three operators in Figure 6(c). The same application, func3, can also be configured as in cf4, which fuses all OPs together with additional sinks. This application configuration is denoted "func3-cf4".

3.2 PE Communication Overhead Learner (PCOL)

As discussed in Section 2.1, the cost of receiving and sending tuples between PEs is an integral part of the overall PE CPU resource usage, so it is important to obtain a good measure of this overhead. An earlier study [15] 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. The model assumes the CPU resource consumption is linear to the data rates. The metrics are 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 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 ran 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 the tuple size increases, the TCP network interface becomes the

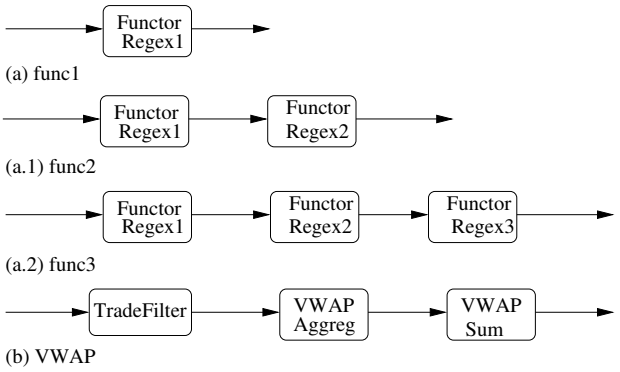


Figure 5: Example SPADE operator graphs. (a) func1, (a.1) func2, (a.2) func3, (b) VWAP.

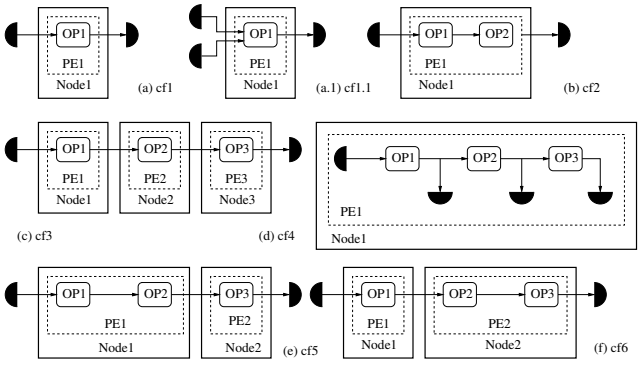


Figure 6: Example fusion and PE-to-node mapping scheme for operator graphs. (a) cf1, (a.1) cf1.1, (b) cf2, (c) cf3, (d) cf4, (e) cf5, (f) cf6.

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 the output port overhead $u^s(r^s, t^s)$ by Equation 5.

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

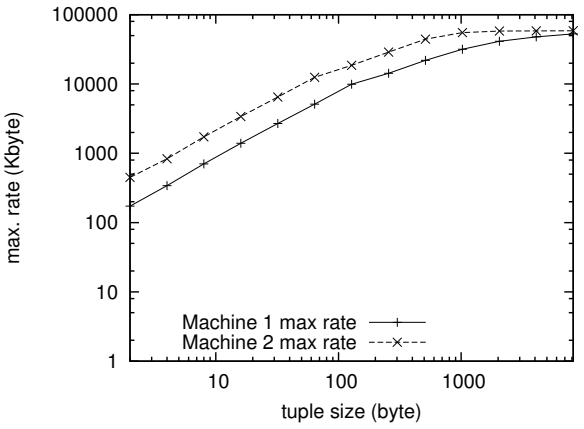
$$u^s(r^s, t^s) = u_{src}(\frac{r^s}{t^s})\frac{r^s}{r_{max}(\frac{r^s}{t^s})} \quad (5)$$

3.3 OP RF Normalizer (ORN)

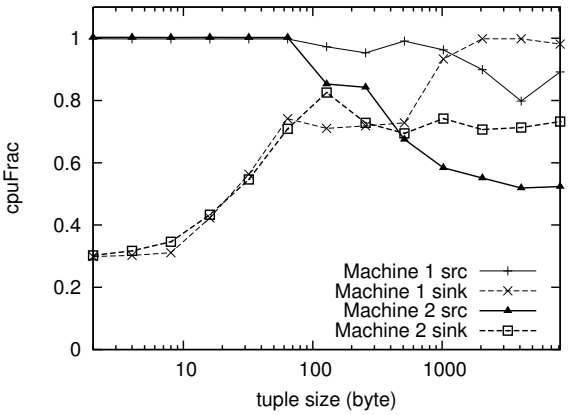
This step tackles the construction of the baseline OP *RF* using data from the System S runtime and the SPADE profiling infrastructure. The *RF*s are stored in the ORD for easy access by other system components. We first discuss our formulation of the OP *RF*s. Next, we show that the SPADE profiling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

3.3.1 Operator RF

In general, the OP *RF*s take the same form as the PE *RF*s presented in Section 2.2, capturing the input rate-dependent aspect of the operator's resource usage. There are other factors such as the operator parameterization (e.g., window



(a) Maximum data rate.



(b) CPU fraction at maximum rate.

Figure 7: PE communication overhead profiling.

size for the join operator[2]) 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 different parameters that are used to configure an operator's internal 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 RFs 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.

[2]A window stores a set of tuples that must be processed together in an operator.

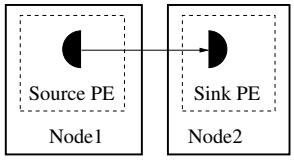


Figure 8: Experiment setup to profile a computing node.

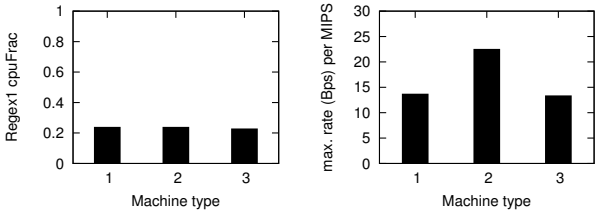


Figure 9: Comparison of node capacity on running func3-cf4.

3.3.2 SPADE OP-Level Profiling and Measurement Inaccuracy

For obtaining OP RFs, we run the application with training data at a range of input stream data rates. Note that the raw data measurement may be inaccurate due to profiling limitations. For accurate OP RF profiling, the measured CPU time $\hat{t}_c$ consumed by each operator should be close to the actual CPU time t_c. However, per-thread CPU usage obtained from the Linux OS only provides 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 can potentially be at a nanosecond scale. Hence, most of the measurements of process time and submit time in Figure 2(b) is zero. To work around this limitation, SPADE uses the following approximation. Instead of the CPU time, it measures the elapsed wall-clock 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 into the CPU time, 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 $\hat{t}_c$ may deviate from the actual t_c. This causes an inaccuracy in the operator's CPU usage measurement.

Specifically, CPU-bound operators may be underestimated because their CPU utilization are 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. 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 measured CPU time $\hat{t}_c$. Refer to Section 3.1 for the detailed explanation of the application configurations. The application func3-cf4 cannot sustain as high a saturated rate as func3-cf3 because cf4 fuses all operators into one PE with sink OPs and uses only one node. The measurement of

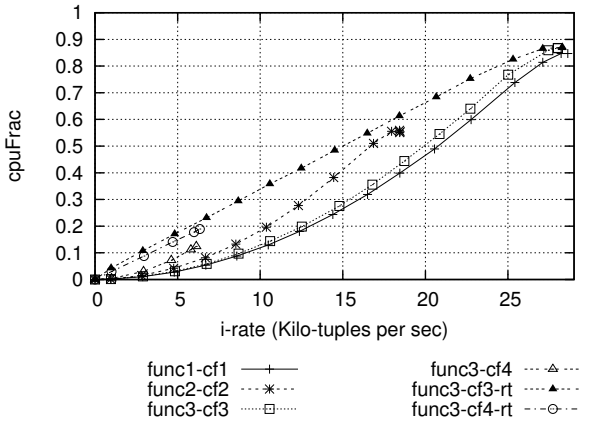


Figure 10: Comparison of the measured OP *RF*s of Regex1 in different configurations.

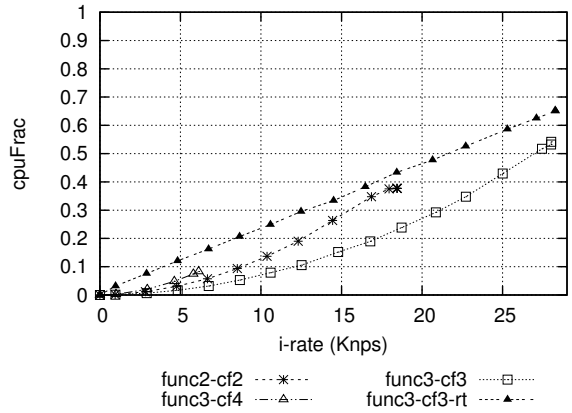


Figure 11: Comparison of the measured OP *RF*s of Regex2 in different configurations.

Regex1 on func3-cf3 and func1-cf1 are the same, indicating that the profiling information of the same OP collected from different applications stays the same. 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" in Figure 10. Since the PE thread containing Regex1 (in func3-cf3 and func3-cf4) runs on a dedicated node (the Linux OS daemon threads use negligible CPU resources and are most likely scheduled on another context in our multi-process machines), context switching rarely happens while the tuple is being processed. Therefore, the wall-clock result should be an accurate measure of the CPU usage time, and we see that the SPADE profiling results (without -rt) under-estimate the true value for all input rates below the saturated rate. The reason is that the PE thread blocks 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. This blocking time at PE input ports reduces the average CPU utilization of the whole thread. In this case, we can see that $\hat{t}_c < t_c$.

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 the highest, 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 Figure 10 and Figure 11 also verifies that these functors have linear *RF*s.

There are some approaches that may improve CPU time measurement precision but are not currently suitable for System S. One way is to use kernel APIs to access hardware counters, such as PAPI [10] to acquire better measurements. It requires installation of external patches to the Linux kernel, which is problematic for the Enterprise versions of the OS. An alternate way is suggested by our Regex1 example – use the elapsed wall-clock time measure. However, in general, a thread can be context-switched during tuple processing, or block due to resource sharing and synchronization. This erroneously inflates 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.

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 Equation (4)-5 and get the OP *RF* by $f = f_e - (u^r + u^s)$.

3.3.3 OP RF Recovery

There are two parts in OP *RF*: the CPU cost function and output rate function versus input data rate. 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 solely on the SPADE profiling metrics. As mentioned above, we assume linear *RF*s, which can be obtained from the raw metrics data using a linear regression.

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 a good fit. More advanced statistical techniques may be used as well, although we have not yet found it necessary in practice.

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**

Second, for applications with large numbers of operators, it may not be possible to even deploy or run the application unless the operators are first fused into a more manageable number of PEs. For such operators, the PE level metrics are not sufficient. Hence, we must rely on the OP-specific metrics collected by the SPADE profiling mechanism. The challenge here is whether 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 (Section 3.3.2) that at saturation, the profiling measurements accurately reflect the CPU usage. Hence, in the case of linear RFs, 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 entire PE graph, 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 would be saturated (step 1-5). 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 would have been 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-threaded 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 is $0.6 \times 0.8 + 0.4 = 0.88$, which is used by the profiling infrastructure to approximate the real OP CPU utilization, 80%. Thus, the profiling measure has a 10% error when it is used to compute the CPU fraction for that OP. Generalizing this to additional OP types is a work-in-progress.

3.4 PE RF Composer (PRC)

PRC composes the baseline PE RF from learned OP RFs. PRC is used to estimate fused PE RFs at compile-time. The composed baseline PE RFs are also used by the scheduler 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 the 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 $\mathbf{g}_e : r^s = c_1 a_1 r_1^r + c_1 a_2 r_2^r + c_2 b r_3^r$. 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 PE RFs to the actual runtime environment. Furthermore, the results are used by ORN to normalize the learned OP RFs to the baseline OP RFs. However, the general problem of node capacity modeling 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 TLB tables in the system increase 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.

In order to construct a general model that can be used across all nodes in a heterogeneous cluster, the scheduler uses *MIPS* as a measure of machine capacity. The *MIPS* used here is based on the processor Bogomips [26] reported by the Linux kernel and adjusted for the multi-context runtime environment. However, our experiences indicate that Bogomips is not a reliable measure of machine capacity and

better accuracy can be achieved by applying domain specific capacity profiling techniques. Accordingly, we propose to use the maximum throughput achieved from running certain streaming micro-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 would 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 on-going work. Figure 9 shows a comparison of three machine types using func3-cf4 as our micro-benchmark. *MIPS* for type-1 machine is 11773.4, type-2 machine is 9946.6, and type-3 machine 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 profiling 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 division 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 whereas type-2 nodes are AMD based. We conclude that the *MIPS* metrics used by the scheduler to model node capacity *may* be useful for the same machine architecture with different CPU speeds. However, it is misleading for machines with different architectures.

Placing multiple PEs on the same node/core may affect the performance since they are sharing caches, memories, and other resources. Our results show that running two PEs on hyper-threading machines (type-1) affects the *RF*s, but on true multi-core machines (type-2), the *RF*s do not show the effects of such contention. Figure 12 shows the CPU fraction used by the func1-cf1 PE at different input data rates, when running on an idle machine of type-1 (4-way hyper-threading). Figure 13 shows the results on machine type 2 with dual-core processors. In each case, the func1-cf1-b line shows the usage when three of the four hyper-threads (or multi-cores, depending on the CPU) have a program pinned to them that is 100% busy. To study the effect of contention with other programs, we run a program that consumes CPU as well as overwrites large parts of the CPU cache. In the figures, the "-zx" denotes the case where we pin a program that uses $0.x$ fraction of the CPU to each hyper-thread/core. The program executes a loop that wakes up periodically and writes to 10MB of memory space, which should pollute the cache. The CPU utilization of the program is adjusted by varying the sleep duration. Thus the PE constantly competes for resources with at least one CPU-intensive program, regardless of which core it runs on. 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 hardware threads on the hyper-threading node affects the PE *RF* (Figure 12). Compared to the results from an idle machine, the CPU usage of the PE increased almost by 50% for the same input rate. The contention may be caused by increasing cache misses on level-2 caches and stall cycles [13]. An analytical model for hardware context switching [23] has suggested that the number of threads that a CPU can support with linear growth of performance is limited. Based on our studies, we will consider using 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 general cases is left as future work.

4. DEMONSTRATION

We use two applications to demonstrate our methodology. First recall the func2-cf2 application, where operators Regex1 and Regex2 are fused and running on a single node. Figure 14 shows the OP *RF*s of Regex1 and Regex2, recovered from a fused PE using our OP *RF* recovery algorithms. Concretely, using Algorithm 2, we fit a curve to the PE measurements which is projected to give the saturated rate point at $(x = 18.16, y = 1)$. 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. The 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 15 shows the estimated PE *RF* of Regex1 using Equation 6. Similarly, Figure 16 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. An expanded version of this paper [29] also shows our approach working for a streaming join with 2 inputs.

The second example, VWAP, contains both functor and aggregate OPs. VWAP is the core processing part of a stock trading application [8] as seen in the appendix. Each operator is defined by the keyword `stream`. Fusion of PEs can either be specified by the keyword `partition`. Node assignment can also be specified by the keyword `node`. Unlike the Regex examples, VWAP operators may select input data depending on predefined conditions. VWAP 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 memory maximum/minimum of the trading prices on a sliding window of size 4. VWAPSum performs arithmetic operations on tuple data fields to compute a volume weighted average price. The baseline OP *RF*s in VWAP are recovered from an unfused configuration running on a type-2 machine using Algorithm 1 (the figures are in the extended version [29]). All OP *RF*s are normalized using the source rate for ease of comparison. Figure 17 shows the output tuple rate of each operator relative to the source rate. It shows that linear rate functions can be used to model the selecting probability of each OP on input training data. Figure 18 shows the predicted and actual PE *RF*s for the fused PE containing TradeFilter and VWAPAggreg operators. Our prediction exactly matches the actual

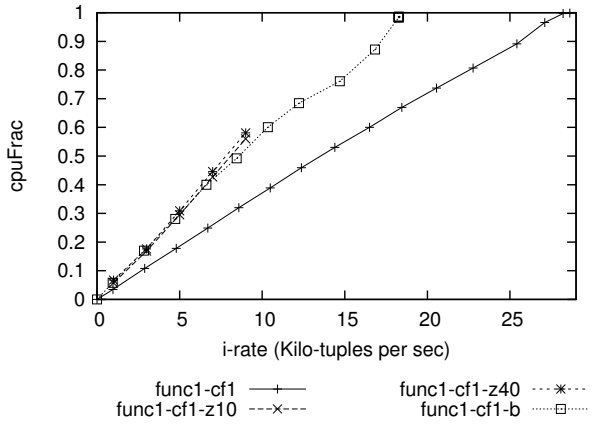


Figure 12: CPU utilization for the PE in func1-cf1, on type-1 machine (hyper-threading) and in a multi-threaded environment.

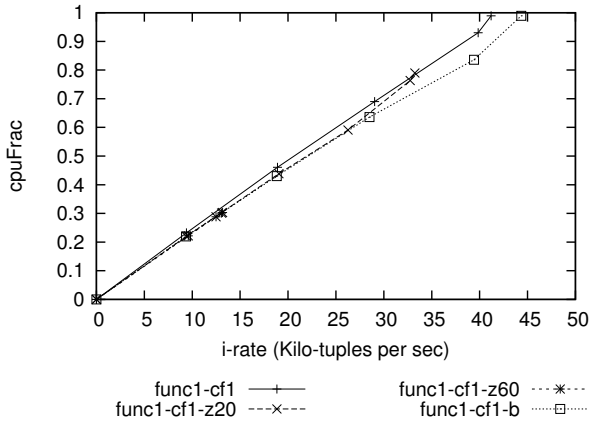


Figure 13: CPU utilization for the PE in func1-cf1 on type-2 machine (multi-core) and in a multi-threaded environment.

PE measurements throughout the full range of input rates. Figure 19 shows the predicted and actual PE *RF*s for the fused PE containing VWAPAggreg and VWAPSum operators. Our prediction over-estimates by at most 15% CPU fraction at the highest rate.

5. 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 the literature [13]. These 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 contention 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 characterize 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 discussed in previous studies [3, 25]. Using queueing theory, Adve et al. [5] provided a deterministic model to estimate the execution time for a parallel program on a symmetric multi-processor system. However, the parameters that were used in their model are difficult to estimate in our system.

Many profiling tools have been developed to measure runtime resource consumption for general-purpose software programs. PAPI [10] provides a cross-platform interface for software programs to use performance hardware counters. Many other profiling tools were developed using PAPI interface, such as PerfSuite [19], HPCToolkit [4], and TAO [24]. Linux kernels later than 2.6.31 have started to support performance counters. Perf [2] is a new tool that uses Linux performance counters. Our recent effort is to explore these tools to resolve the profiling limitation of SPADE. There are some sampling based profiling tools available on Linux, such as gprof [14] and OProf [1]. However, gprof does not support multi-threading. OProf requires superuser privilege that is not suitable in our application. PIN [20] is another general dynamic binary instrumentation tool that can be used for application profiling. Moore et al. [22] compared existing performance analysis tools.

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 by Zhang et al. [28] to improve the execution for programs on a single machine. Merten et al. [21] proposed a runtime optimization system with hardware-driven profiling system.

6. CONCLUSION

In this paper, we have described an empirical approach for constructing quantitative workload characterization for basic operators in streaming systems. The dataflow architecture of stream processing systems 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 challenge 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 optimization and runtime resource allocation optimization.

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 as seen in the empirical validation described in Section 4. We find that the predicted PE *RF* are within 15% CPU fraction compared to actual measurements from the fused PE. Hyperthreading machines induce additional contention that requires additional characterizing, while multi-core machines do not.

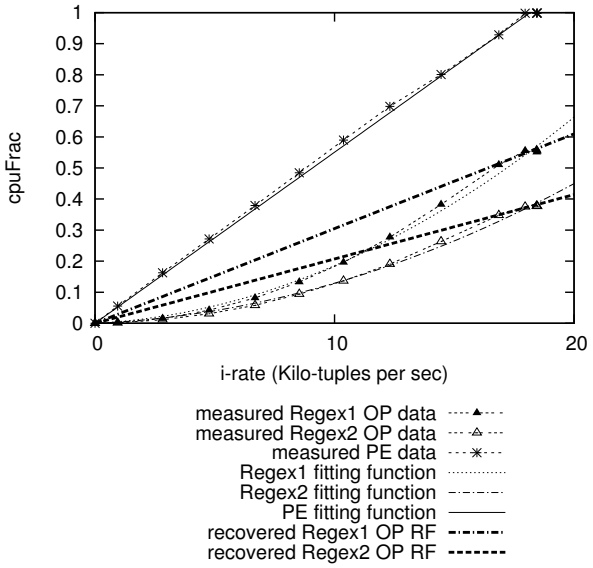


Figure 14: Regex1 and Regex2 OP *RF*s recovered from the fused PE in func2-cf2.

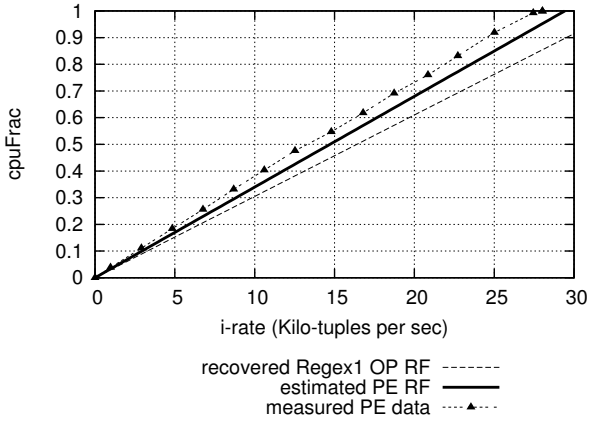


Figure 15: Estimated PE *RF* for unfused Regex1 in func3-cf3.

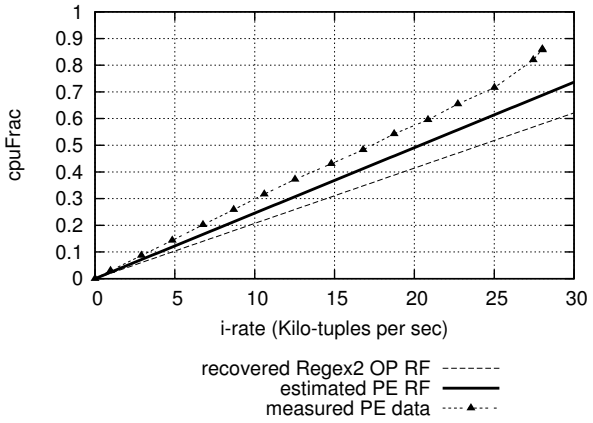


Figure 16: Estimated PE *RF* for unfused Regex2 in func3-cf3.

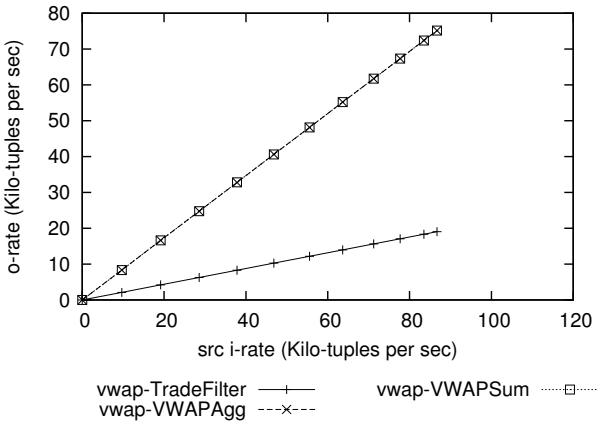


Figure 17: I/O rate ratios for VWAP OPs.

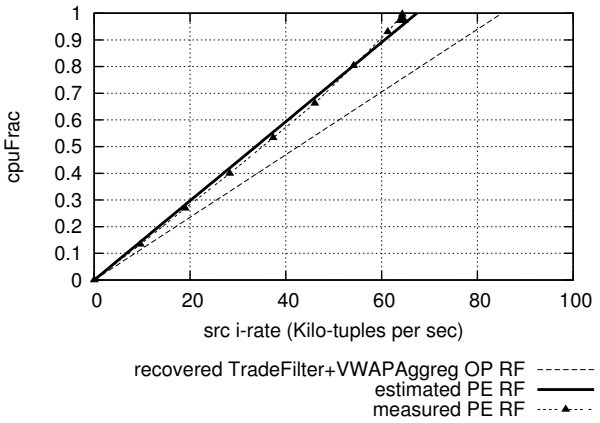


Figure 18: Estimated PE *RF* for fused TradeFilter and VWAPAggreg in vwap-cf5, on type-2 machine.

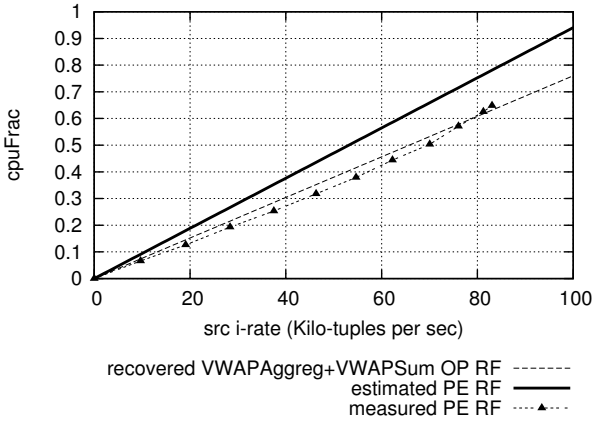


Figure 19: Estimated PE *RF* for fused VWAPAggreg and VWAPSum in vwap-cf6, on type-2 machine.

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

7. REFERENCES

[1] OProfile. `http://oprofile.sourceforge.net/`.

[2] Perfwiki. `http://perf.wiki.kernel.org/index.php/Main_Page`.

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

[4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[5] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Transactions on Computer Systems*, 22(1):94–136, 2004.

[6] Y. Ahmad, B. Berg, U. Cetintemel, M. Humphrey, J.-H. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, and S. Zdonik. Distributed operation in the borealis stream processing engine. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 882–884, New York, NY, USA, 2005. ACM.

[7] 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: Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, pages 27–37, New York, NY, USA, 2006. ACM.

[8] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu. Scale-up strategies for processing high-rate data streams in System S. In *International Conference on Data Engineering*, pages 1375–1378, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[9] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[10] 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.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 668–668, New York, NY, USA, 2003. ACM.

[12] J. Chen, D. J. Dewitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 379–390, New York, NY, USA, 2000. ACM.

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

[14] J. Fenlason and R. Stallman. GNU gprof: The GNU profiler. `http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html`.

[15] B. Gedik, H. Andrade, and K.-L. Wu. A code generation approach to optimizing high-performance distributed data stream processing. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 847–856, New York, NY, USA, 2009. ACM.

[16] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the System S declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1123–1134, New York, NY, USA, 2008. ACM.

[17] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Mendell, H. Nasgaard, R. Soule, and K.-L. Wu. SPL stream processing language specification. Technical Report RC24897, IBM Research, 2009.

[18] 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: the 2006 ACM SIGMOD International Conference on Management of Data*, pages 431–442, New York, NY, USA, 2006. ACM.

[19] R. Kufrin. Measuring and improving application performance with perfsuite. *Linux Journal*, 2005(135):4, 2005.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[21] 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: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 136–147, Washington, DC, USA, 1999. IEEE Computer Society.

[22] 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.

[23] 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.

[24] S. S. Shende and A. D. Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20:287–331, 2006.

[25] S. Siddha. Multi-core and Linux kernel. Intel Inc., 2007.

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

[27] 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: Proceedings of the 9th International Middleware Conference*, Dec. 2008.

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

[29] X. J. Zhang, S. S. Parekh, B. Gedik, H. Andrade, and K.-L. Wu. Performance modeling of operators in a streaming system. Technical Report RC24945, IBM Research, 2009.

APPENDIX

The SPADE source code of the application Volume Weighted Average Price (VWAP). Please refer to the SPADE language specification [17].

```
[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)
```