Run Time Approximation of Non-blocking Service Rates for Streaming Systems

Jonathan C. Beard
Roger D. Chamberlain

Jonathan C. Beard and Roger D. Chamberlain, “Run Time Approximation of Non-blocking Service Rates for Streaming Systems,” in *Proc. of IEEE 17th International Conference on High Performance Computing and Communications* (HPCC), August 2015, pp. 792-797.

Dept. of Computer Science and Engineering
Washington University in St. Louis

2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

Run Time Approximation of Non-blocking Service Rates for Streaming Systems

Jonathan C. Beard*†
*ARM Research - Austin, TX
Email: jonathan.beard@arm.com

Roger D. Chamberlain†
†Dept. of Computer Science and Engineering
Washington University in St. Louis
Email: roger@wustl.edu

***Abstract*—Stream processing is a compute paradigm that promises safe and efficient parallelism. Its realization requires optimization of multiple parameters such as kernel placement and communications. Most techniques to optimize streaming systems use queueing network models or network flow models, which often require estimates of the execution rate of each compute kernel. This is known as the non-blocking "service rate" of the kernel within the queueing literature. Current approaches to divining service rates are static. To maintain a tuned application during execution (while online) with non-static workloads, dynamic instrumentation of service rate is highly desirable. Our approach enables online service rate monitoring for streaming applications under most conditions, obviating the need to rely on steady state predictions for what are likely non-steady state phenomena. This work describes an algorithm to approximate non-blocking service rate, its implementation in the open source RaftLib [2] framework, and validates the methodology using streaming applications on multi-core hardware.**

I. Introduction

Stream processing (or data-flow programming) is a compute paradigm that enables parallel execution of sequentially constructed kernels. This is accomplished by managing the scheduling of kernels and the flows of data (called *streams*) from one sequentially programmed kernel to the next. Queueing behavior naturally arises between two independent kernels. Selecting the correct queue capacity (buffer size) is one parameter (of many) that can be critical to the overall performance of the streaming system. Doing so often requires information, such as the service rate of each kernel, not typically available at run time (online). Complicating matters further, many analytic methods used to solve for optimal buffer size require an understanding of the underlying service process distribution, not just its mean. Both service rate and process distribution can be difficult to determine online without degrading application performance. This paper proposes and demonstrates a heuristic that enables online service rate approximation of each compute kernel within a streaming system.

Optimizing the queueing network that models a streaming application [11] can be performed using analytic techniques. Ubiquitous to many of these models is the non-blocking service rate of each compute kernel. Classic approaches assume a stationary distribution. One only has to look at the variety of data presented to any common application to realize that the assumption of a persistent homogeneous workload is naive. With the popularity of cloud computing we also have to assume that the environment an application is executing in can change at a moments notice, therefore we must build applications that can be resilient to perturbations in their execution environment. We focus on low overhead instrumentation that will enable more resilient stream processing applications by informing the runtime when conditions change. To the best of our knowledge there have been no other low overhead approaches to determine the online service rate of a compute kernel executing within a streaming system.

II. Background & Related Work

At its core, this work is about low-overhead instrumentation of software systems. Early software instrumentation tools include call graph analyzers such as `gprof` [6]. Other instrumentation tools such as TAU [14] provide low overhead instrumentation and visualization for MPI style systems. What these tools don't provide is a tool for reporting buffer performance during execution, our instrumentation does.

Modern stream processing systems can dynamically re-optimize in response to changing conditions (workload and/or computing environment). To re-optimize buffer allocations there are generally two choices: empirical search or analytic queueing models. Queueing models are desirable for this purpose since they can divine a buffer size directly, eschewing unnecessary buffer re-allocations. Utilizing these models dynamically, however, requires dynamic instrumentation. Tools such as DTrace [3] and Pin [12] can provide certain levels of dynamic information on executing threads. Our approach differs from the above in that we are specifically targeting methods for estimating online service rate in a low overhead manner.

Work by Lancaster et al. [10] and Chamberlain and Lancaster [4] laid out logic that could ostensibly make online service rate determination possible. They suggest measuring the throughput into a kernel when there is sufficient data available within its input queue(s) and no back-pressure from its output queue(s). This logic works well for FPGA-based systems where hardware is controlled by the developer. For multi-core systems, however, this logic is too simplistic. The aforementioned work assumes that the measurements of a non-blocked service rate are all equal (i.e., the full service rate is observed at every sample point). In reality things like partially full queues result in less than accurate service rate estimates using this procedure. Anomalies such as cache behavior and clock variations can exacerbate understanding of the true service rate of a compute kernel. Making matters worse are context swaps that occur when one thread is observing another. In reality, sampling the service rate of a compute kernel looks like Figure 1 where multiple outliers and noise confound understanding of the true service rate.

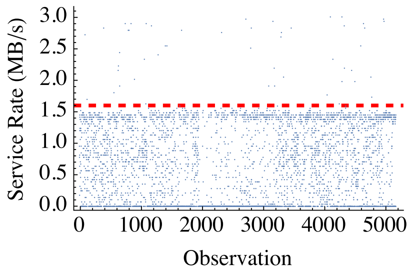

Fig. 1. Direct observations of the service rate, using the logic of [4], for a nominally fixed rate micro-benchmark kernel. The x-axis is the increasing observation index with time, the y-axis represents the actual data rate observed at each sample point. The red dashed line is the nominal service rate.

Central to accurate estimation of service rate is observing non-blocking reads and writes by the server. While executing, the probability of observing a non-blocked read or write to a queue in general is very low for high performance systems (i.e., those systems whose compute kernels have high utilization). The equations below (a modification of the equations given by Kleinrock [9]) give the probabilities for the simplified case where each server's process is Poisson and only a single in-bound and out-bound queue are considered (also known as an $M/M/1$ [8] queue; Table I lists variable definitions).

TABLE I
NOMENCLATURE USED FOR QUEUEING EQUATIONS.

Symbol	Description
μ_s	mean service rate
ρ	server utilization
C	capacity of output queue
T	sampling period of monitor
k	items needed by server during T

$$k = \lceil \mu_s T \rceil$$

$$Pr_{\text{READ}}(T, \rho, \mu_s) = \rho^k$$

$$Pr_{\text{WRITE}}(T, C, \rho, \mu_s) = \begin{cases} 1 - \rho^{C-k+1} & C \geq \mu_s T \\ 0 & C < \mu_s T \end{cases}$$

In general the shorter the service time, the lower the probability of observing a non-blocking read or write. Lengthening the observation period, T, decreases the probability that blocking will not occur during the observation period whereas shorter periods increase the probability of observation (i.e., no blocking during the period).

III. Monitoring Mechanism

The simple act of observing a rate can change the behavior being observed. To minimize this effect, our instrumentation scheme (implemented within RaftLib) uses a separate monitoring thread. The benefit of this arrangement is that it moves the instrumentation out of the critical path of the application. The drawback is that this increases the sensitivity to timing precision and the probability of noise within each observation.

To minimize the overall performance impact, the data necessary to estimate the service rate is split between the queue itself and the monitor thread. This has the benefit of transmitting data only when absolutely necessary. The queue itself is visible to three distinct threads: the monitor thread and the producer/consumer threads at either terminus of the queue. The only logic to consider within the queue itself is that necessary to tell the monitor thread if an operation was blocked and to increment a item counter on each read and write. The monitor thread reads these variables (which will be called tc from this point forward) and resets them upon copying their values.

The monitor thread samples at a fixed interval of time T. When the monitor thread samples tc, it has no way of knowing if the server at either end only performed complete executions or partial ones. The only thing it can be certain of is that the data read are non-blocking if the boolean value is set appropriately. This means that tc can represent something less than the actual service rate. Also contained within the tc are effects not-representative of average behavior; these include: caching effects, interrupts, memory contention, faults, etc.

IV. Service Rate Monitoring

Online estimation of service rate requires four basic steps: fixing a stable sampling period T, sampling only the correct states (expounded upon below), reducing and de-noising the data, then estimating the non-blocking service rate. The system has a finite number of states which are useful in estimating the non-blocking service rate. The most obvious states to ignore are those where the in-bound or out-bound queue is blocked (see Lancaster et al.). The others, as mentioned in Section III, are data unrepresentative of the non-blocking service rate. Symbols used in this section are summarized in Table II.

TABLE II
NOMENCLATURE USED FOR SECTION IV.

Symbol	Description
T	sampling period
tc	sum of non-blocking reads during T
S	windowed set of items tc
S'	Gaussian filtered set of S
q	95^{th} quantile of S'
$\bar{q}$	population averaged q
d	bytes per data item

A. Sampling Period Determination

Each queue within a streaming application has its own monitor thread. As such, each T is queue specific. An initial requirement is a stable time reference across all utilized cores. The timing method described by Beard and Chamberlain [1] is employed. This provides a stable and monotonically increasing time reference whose latency on our test systems is approximately $50-300$ ns across the cores. Despite a relatively stable time reference, two trends complicate matters. First, as service time decreases, the probability of observing a non-blocking queue transaction decreases as well. Second, noise from the system and timing mechanism dominate for very small values of T making observations unusable.

Modern computing systems introduce some level of noise into the measurements [1]. Choosing a a longer sampling period (T) reduces the impact of the noise. We wish, however, to observe kernel executions that are unimpeded by their environment (no blocking due to upstream or downstream effects). This goal stands in juxtaposition to noise reduction since shorter sampling periods increase the probability of observing non-blocked periods of execution.

Figure 2 shows how the empirically observed sampling period varies with desired sampling period, T, starting with the minimum latency ($\sim$300 ns) of back to back timing requests then iterating over multiples of that latency. The monitor thread tries to find the largest time period T (moving to the right in Figure 2) while minimizing observed queue blockage during the period. As is expected, the noise is less significant compared to the period as T increases.

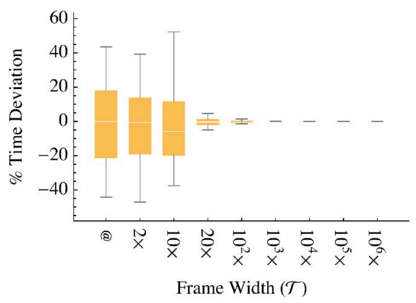

Fig. 2. Observations of T variation using the timing mechanism of [1]. The @ symbol represents the minimum resolution of the timing mechanism ($\sim$300 ns for this example), subsequent box and whisker observations are the indicated multiple of @. The trend indicates wider time frames (up to the approximate time quanta for the scheduler) give more stable values of T.

B. *Service Rate Heuristic*

Once a stable T has been determined, the next step is estimating the online service rate. The head and tail of each queue store counts of non-blocking transactions, tc, as well as the size of each item copied, d. The instrumentation thread samples tc from the head and tail every T seconds. We will use an estimate of the maximum, well-behaved tc to estimate the service rate of interest. (well-behaved is articulated below). For simplicity, the discussion that follows will consider only actions that occur at the head of the queue with the understanding that equivalent actions can occur at the tail as well.

Algorithm 1 summarizes the process described below. This description presumes an implementation of a streaming mean and standard deviation (see Chan et al. [5]) through the $updateStats()$, $updateMeanQ()$ and $resetStats()$ methods. The mechanics of the $QConverged()$ function are described later.

```
$stream \leftarrow tc$;
$output \leftarrow$ output stream;
$S \leftarrow \{\}$;
while True do
    $tc_{current} \leftarrow pop(stream)$;
    $S' \leftarrow \{\}$;
    for $i \leftarrow gauss_{radius}$,
    $i < |window| - gauss_{radius}, i++$ do
        $val \leftarrow$ Dot( $S[i - gauss_{radius};;i + gauss_{radius}], GaussianFilter$ );
        push( $S'$, $val$ );
    end
    $\mu_{S'} \leftarrow$Mean($S'$);
    $\sigma_{S'} \leftarrow$StandardDeviation( $S'$);
    $q \leftarrow$NQuantileFunction( $\mu_{S'}$, $\sigma_{S'}$, .95 );
    updateStats( $q$ ); if QConverged() then
        push( $output$, getMeanQ() );
        resetStats();
    end
end
```

Algorithm 1: Service rate heuristic.

While sampling tc, the timing thread creates a list S, ordered by entry time. S is maintained as a sliding window of size w. If S is of sufficient size, it is expected that the set S tends toward a Gaussian distribution ($\mathcal{N}(\mu_S, \sigma_S)$), as it is a list of sums of non-blocking transactions. S, however consists of many data that are not necessarily indicative of non-blocking service rates. These elements arise from the following conditions: (1) the monitor thread observed only a partial firing of the server (i.e., the server had the capability to remove j items from the queue but only $< j$ items were evident when retrieving tc); (2) the monitor thread clears the queue's current value of tc during a firing (i.e., the counter maintaining tc is non-locking because locking it introduces delay); (3) outlier conditions as discussed in Section II which are not indicative of normal behavior.

The underlying distribution of S without outliers tends towards a Gaussian, therefore a Gaussian discrete filter is used to shape the data in S so that it is sufficiently well-behaved (de-noised) for estimating the maximum. The filtered data make up the set S'. Equation 2 describes the kernel, where $x \leftarrow [-2, 2]$ is the index with respect to the center. Through experimentation, a radius of two was selected as providing the best balance of fast computation and smoothing effect.

$$GaussianFilterKernel(x) \leftarrow \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}} \tag{2}$$

Once filtered, S' is used to estimate the maximum. Since we must still account for outliers, rather than explicitly use the maximum, we estimate the maximum via the 95^{th} quantile of S'. Operationally, we use the sample mean, $\widehat{\mu_{S'}}$, and standard deviation, $\widehat{\sigma_{S'}}$, to estimate $\mathcal{N}(\mu_{S'}, \sigma_{S'})$, and the quantile is of course

$$q = \widehat{\mu_{S'}} + 1.64485\ \widehat{\sigma_{S'}}. \tag{3}$$

Stability is gained by using the streaming mean of successive values of q_i (e.g., Figure 3). Where $\bar{q}$ is the averaged, estimated maximum non-blocking transaction count tc, assuming only one queue for simplicity, the service rate is simply $\frac{\bar{q}\times d}{T}$. This, however also assumes that the underlying distribution generating tc is at least stable over the observation period. As with all online estimates, $\bar{q}$ becomes more stable with more observations (e.g., Figure 4).

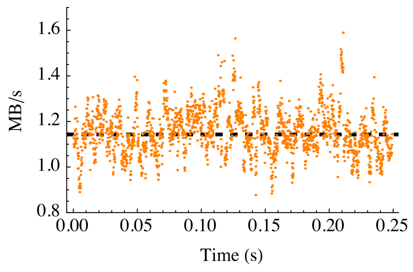

Fig. 3. Plot of the values of q with increasing time. Each value of q is the result of a computation of Equation 3. The dashed line across the y-axis represents the set or expected service rate.

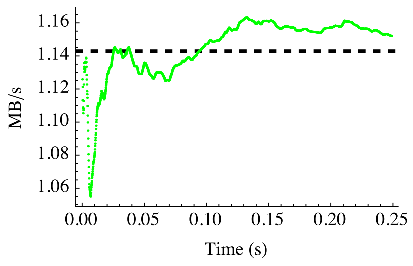

Fig. 4. An example of convergence of $\bar{q}$ with increasing time. Data is from a single queue tandem server micro-benchmark, observing the departure rate from the queue to the server with the set service rate marked as a dashed line.

Convergence of $\bar{q}$ to a "stable" value is expected after a sufficiently large number of observations. In practice, with μs-level sampling, convergence is rarely an issue. Determining when $\bar{q}$ is stable is accomplished by observing σ of $\bar{q}$. Minimizing the standard deviation is equivalent to minimizing the error of $\bar{q}$. With a finite number of samples, it is unlikely that $\sigma(\bar{q})$ will ever equal to zero, however observing the rate of change of the error term to a given tolerance is a typical approach. To accomplish this in a streaming manner, a similar approach to that taken previously is used with differing filters to approximate the relative rate of change over the window. A discrete Gaussian filter with a radius of one is followed by a Laplacian filter with discretized values (in practice, one combined filter is used). The kernel is given in Equation 4 with $x \leftarrow [-1, 1]$ and $\sigma \leftarrow \frac{1}{2}$. The minimum and maximum of the filtered $\sigma(\bar{q})$ are kept over a window $w \leftarrow 16$ where convergence is judged by these values all being within some tolerance (ours is set to 5×10^{-7}).

$$LaplacianGaussian(x) \leftarrow \frac{x^2 e^{-\frac{x^2}{2\sigma^2}}}{\sqrt{2\pi\sigma^5}} - \frac{e^{-\frac{x^2}{2\sigma^2}}}{\sqrt{2\pi\sigma^3}} \qquad (4)$$

An example of a stable and converged $\bar{q}$ is shown in Figure 5, where the data plot is of the dual filtered $\sigma(\bar{q})$ and the vertical line is the point of convergence. The time scale on the x-axis is the same as that of Figure 4 so that the stability point on Figure 5 matches that of Figure 4.

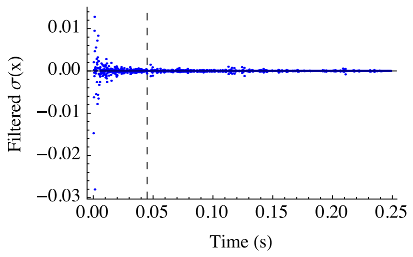

Fig. 5. Plot of the filtered standard deviation of $\bar{q}$, the point of convergence is indicated by the vertical dashed line.

Once convergence is achieved, it is a simple matter to restart the process described above, and begin the search again. Figure 6 shows a sample run where the average service rates are known (solid blue y-axis grid lines). The x-axis grid lines (dashed vertical lines) show points of convergence to stable solutions after subsequent restarts of the approximation algorithm.

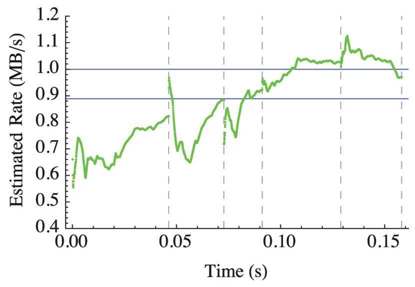

Fig. 6. Example of $\bar{q}$ adapting to two service rates during execution of a micro-benchmark. The instrumentation captures the departure rate from a single queue to a compute kernel.

V. Experimental Setup

The hardware used for all empirical evaluation are listed in Table III. All binaries are compiled with the "-O2" compiler flag using the GNU GCC compiler (version 4.8.3). The streaming framework used is the RaftLib C++ template library [2].

In order to assess our method over a wide range of conditions, a simple micro-benchmark consisting of two threads connected by a lock-free queue is used. Each thread consists of a while loop that consumes a fixed amount of time in order to simulate work with a known service rate. The amount of

TABLE III
SUMMARY OF HARDWARE USED FOR EMPIRICAL EVALUATION.

ID	Processor	OS	Memory
1	2 × AMD Opteron 6136	Linux 2.6.32	64 GB
2	2 × Intel E5-2650	Linux 2.6.32	64 GB
3	2 × Intel Xeon X5472	Darwin 13.4.0	32 GB
4	2 × Six-Core AMD Opteron 2435	Linux 3.10.37	32 GB
5	Intel Xeon CPU E3-1225	Linux 3.13.9	8 GB

work, or service-rate, is generated using a random number generator sourced from the GNU Scientific Library. Service time distributions are set as either exponential or deterministic. Parameterization of the distributions is selected using a pseudo random number source. The exact parameterization range and distribution are noted where applicable.

In addition to the micro-benchmarks described above, two full streaming applications are also explored. The first, matrix multiply, is a synchronous data flow application that is expected to have relatively stable service rates. The second is a string search application that has variable rates. Ground truth service rates for each kernel are determined by executing each kernel offline and measuring the rates individually.

Matrix Multiply: Matrix multiplication is central to many computing tasks. Implemented here is a simple dense matrix multiply ($C = AB$) where the multiplication is broken into multiple dot-product operations. The dot-product operation is executed as a compute kernel with the matrix rows and columns streamed to it. This kernel can be duplicated n times. The result is then streamed to a reducer kernel which reforms the output matrix C. This application differs from the micro-benchmarks in that it uses real data read from disk and performs multiple operations on it. As with the micro-benchmarks, it has the advantage of having a continuous output stream from both the matrix read and dot-product operations. The data set used is a $10,000 \times 10,000$ matrix of single precision floating point numbers produced by a uniform random number generator.

Rabin-Karp String Search: The Rabin-Karp [7] algorithm is classically used to search a text for a set of patterns. It utilizes a "rolling hash" kernel (replicated up to n times) to efficiently compute the hash of the text being searched as it is streamed in. The next kernel verifies the match from the rolling hash to ensure hash collisions don't cause spurious matches. The verification matching kernel can be duplicated up to j times. The final kernel simply reduces the output from the verification kernel(s), returning the byte position of each match. The corpus consists of 2 GB of the string, "foobar."

VI. RESULTS

The methods that we have described are designed to enable online service rate determination. Just how well do these methods work in real systems while they are executing? In order to evaluate this quantitatively, several sets of micro-benchmarks and real applications are instrumented to determine the mean service rate of a given server.

Each micro-benchmark is constructed with a tandem configuration of two kernels (A and B). The upstream kernel A provides a fixed distribution arrival process to kernel B. The service rate of kernel B is varied for each micro-benchmark (0.8 MB/s $\rightarrow \sim 8$ MB/s). The results comprise 1800 executions in total. The goal is to find the service rate of kernel B without *a priori* knowledge of the actual rate. Figure 7 is a histogram of the percent difference between the service rate estimated via our method and the "set" nominal rate.

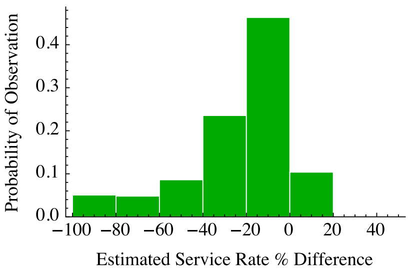

Fig. 7. Histogram of the probability of estimating the service rate of micro-benchmark kernel B. Each execution is a data point, with the percent difference calculated as $\left(\frac{\text{(observed rate} - \text{set rate)}}{\text{set rate}}\right) \times 100$. Not plotted are four outliers to right of the plotted data which are greater than 1000% difference.

We see in this histogram that generally the correspondence between estimated service rate and ideal service rate is reasonably good. When it errs, the estimate is typically low, which is consistent with previous empirical data, in which actual realized execution times are typically longer than nominal [1]. The majority of the results are within 20% of nominal.

We implement change by moving the mean of the distribution halfway through execution of kernel B We are interested in whether our instrumentation can detect this change, potentially enabling many online optimizations. An example with a wide switch in service rate is shown in Figure 8.

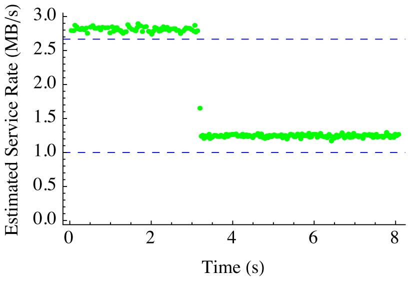

Fig. 8. Depiction of the ideal (drawn from empirical data) of the instrumentation's ability to estimate the service rate while the application is executing. Each dot represents the converged service rate estimate (y-axis). The top and bottom dashed lines represent the first and second phases.

In order to classify the dual phase results into categories, a percent difference (20%) from the manually determined rates for each phase is used. Approximately 14.7% of the data had nominal service rate shifts that were known to be less than the 20% criteria specified. Table IV shows the effectiveness of the heuristic in categorizing distinct execution phases of the micro-benchmarks. Here two observations can be made. First, the system correctly detects both phases more effectively in high utilization conditions, which are the conditions under which correct classification is likely to be more important. Second, the classification errors that are made are all conservative. That is, it is correctly detecting the final condition of the kernel, indicative of a conservative settling period for rate estimation.

TABLE IV
DATA FROM DUAL-PHASE MICRO-BENCHMARK.

Phases Detected	$\rho < 0.5$	$\rho \geq 0.5$
Neither phase	0.0%	0.3%
1st phase only	0.0%	0.0%
2nd phase only	56.6%	27.5%
Both phases	43.4%	72.2%

The real test of any instrumentation is how well it can handle situations beyond those that are carefully controlled. The matrix multiply application is executed on platform 2 from Table III with the number of parallel dot-products set to five. Only the reduce kernel is instrumented. Overall the results are not quite as clean as those of the micro-benchmark, but that is expected given the chosen kernel has an extremely low ρ. A majority (63%) are within the range of measurements observed using isolated instrumentation.

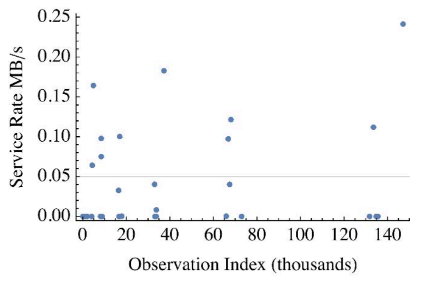

Fig. 9. Plot of the estimated service rates of the reduce kernel. The manually determined rate for this experimental setup ranged from 0.05 MB/s to 0.43 MB/s. Overall, a majority of the results, ~63%, are within this range.

The results for the Rabin-Karp application are also relatively good for a variable data rate application with low ρ. For this case $\sim 35\%$ of the estimates are within the range of rates determined by manual time averaged estimates. The others are quite close and we believe represent the transient behaviors of a real application. The time averaged online service rate estimates are within 5% of (just below) the manually determined average service rate; indicating that the variation in online observations are likely behaviors that only last for fractions of the overall application.

Low overhead instrumentation should have little, if any, impact on the execution of the application itself. Using the single queue micro-benchmark, execution time is measured with and without instrumentation. Using the GNU `time` command over dozens of executions, the average impact is only 1 - 2%. Impact to the overall system was equally minimal, load average increased only a small amount (by 0.1 on average).

VII. Conclusions & Future Work

We demonstrate an algorithm for approximating the online service rate of kernels in a streaming system. Overall our methodology works quite well. When the heuristic fails, it usually fails knowingly (e.g., no convergence is reached or non-blocking reads were not observed). It has been validated using micro-benchmarks and two full streaming applications. While evidence has been shown for the estimation of the service rate's central moment and its variance, efficient methods also exist for streaming computation of higher moments [13]. RaftLib currently supports the methods described in this paper. Future work, and extensions to RaftLib's instrumentation system, will include higher moment estimation.

Acknowledgments

This work was supported by Exegy, Inc., and VelociData, Inc. Washington University in St. Louis and R. Chamberlain receive income based on a license of technology by the university to Exegy, Inc., and VelociData, Inc.

References

[1] J. C. Beard and R. D. Chamberlain, "Use of a Levy distribution for modeling best case execution time variation," in *Computer Performance Engineering*, ser. Lecture Notes in Computer Science, A. Horváth and K. Wolter, Eds. Springer, Sep. 2014, vol. 8721, pp. 74–88.
[2] J. C. Beard, P. Li, and R. D. Chamberlain, "Raftlib: A C++ Template Library for High Performance Stream Parallel Processing," in *Proc. of Programming Models and Applications on Multicores and Manycores*. ACM, Feb. 2015, pp. 96–105.
[3] B. Cantrill, M. W. Shapiro, A. H. Leventhal *et al.*, "Dynamic instrumentation of production systems." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 15–28.
[4] R. D. Chamberlain and J. M. Lancaster, "Better languages for more effective designing," in *Proc. of Int'l Conf. on Engineering of Reconfigurable Systems & Algorithms*, Jul. 2010.
[5] T. F. Chan, G. H. Golub, and R. J. LeVeque, "Algorithms for computing the sample variance: Analysis and recommendations," *The American Statistician*, vol. 37, no. 3, pp. 242–247, 1983.
[6] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "gprof: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 120–126, 1982.
[7] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.
[8] M. Kendall and W. Buckland, *A Dictionary of Statistical Terms*. Edinburgh and London: Published for the International Statistical Institute by Oliver & Boyd, Ltd., 1957.
[9] L. Kleinrock, *Queueing Systems. Volume 1: Theory*. Wiley-Interscience, 1975.
[10] J. M. Lancaster, J. G. Wingbermuehle, and R. D. Chamberlain, "Asking for performance: Exploiting developer intuition to guide instrumentation with TimeTrial," in *Proc. 13th Int'l Conf. High Performance Computing and Communications*, Sep. 2011, pp. 321–330.
[11] S. S. Lavenberg, "A perspective on queueing models of computer performance," *Performance Evaluation*, vol. 10, no. 1, pp. 53–76, 1989.
[12] 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," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 190–200, 2005.
[13] P. Pébay, "Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments," *Sandia Report SAND2008-6212, Sandia National Laboratories*, 2008.
[14] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.