A Performance Analysis of System S, S4, and Esper via Two Level Benchmarking

Miyuru Dayarathna[1] and Toyotaro Suzumura[1,2]

[1] Department of Computer Science, Tokyo Institute of Technology, 2-12-1 Ookayama, Meguro-ku, Tokyo 152-8552, Japan
dayarathna.m.aa@m.titech.ac.jp, suzumura@cs.titech.ac.jp
[2] IBM Research - Tokyo

Abstract. Data stream processing systems have become popular due to their effectiveness in applications in large scale data stream processing scenarios. This paper compares and contrasts performance characteristics of three stream processing softwares System S, S4, and Esper. We study about which software aspects shape the characteristics of the workloads handled by these software. We use a micro benchmark and different real world stream applications on System S, S4, and Esper to construct 70 different application scenarios. We use job throughput, CPU, Memory consumption, and network utilization of each application scenario as performance metrics. We observed that S4's architectural aspect which instantiates a Processing Element (PE) for each keyed attribute is less efficient compared to the fixed number of PEs used by System S and Esper. Furthermore, all the Esper benchmarks produced more than 150% increased performance in single node compared to S4 benchmarks. S4 and Esper are more portable compared to System S and could be fine tuned for different application scenarios easily. In future we hope to widen our understanding of performance characteristics of these systems by investigating in to the code level profiling.

Keywords: stream processing, data-intensive computing, workload characterization, performance analysis, benchmarking, systems scalability.

1 Introduction

Stream processing [16] (which is also called Complex Event Processing [5]) has emerged as an exciting new filed to support online information processing activities. These software process data on-the-fly, in-memory without requiring to store data in secondary storage. There have been extensive studies for characterizing the workload and performance implications of computing systems. However, there has not been sufficient amount of such studies carried out in the area of stream computing. In this paper we work on stream processing software performance characterization using System S, S4, and Esper; which are currently three prominent stream processing software in the field. Decision to

K. Joshi et al. (Eds.): QEST 2013, LNCS 8054, pp. 225–240, 2013.

choose these three software was stimulated due to their unique architectural designs. While System S is developed following a manager and worker model; S4 has a decentralized and symmetric architecture and follows Actors model [9]. Esper is completely different from System S and S4 because it is just a component for stream processing [4]. However, Esper provides a complete software suite for stream processing which has been used by popular software vendors for their event processing back-ends. Furthermore, current implementation of S4's operators are purely based on Java and also Esper is a pure Java library, whereas System S allows for both C/C++ and Java versions. We checked the software's licenses and got confirmed that they allow for publishing performance comparisons.

In achieving the aforementioned objectives we created 70 different experiment scenarios using three real-world application benchmarks and a micro benchmark. Performance characteristics such as Job throughput, CPU usage, Memory Usage, Network I/O were observed in arriving at conclusions about the design of stream processing system architectures.

2 Related Work

There have been several previous studies on characterizing performance of stream processing systems. Mendes *et al.* have conducted a performance evaluation of three event processing systems by running several micro-benchmarks [8]. Their intention was to provide a first insight in to the performance of event processing systems. We try to delve more deep in to the performance characteristics of such systems. They have used Esper similar to us. However, their study has been conducted in single node settings. Different to them, we implemented all the benchmarks using Esper in distributed settings using Java Messaging Service (JMS) [12].

Suzumura *et al.* made a performance study considering ETL (Extract-Transform-Load) scenario of System S [14]. However, our intention is completely different from their work. While they evaluate performance of System S in the context of ETL applications; we aim for identifying the characteristics of stream processing systems in a more general context. The works done by Parekh *et al.* and Zhang *et al.* study methods for characterizing resource usage profiles and characterizing the resource usage of Processing Elements (PEs) respectively [19][11]. Both these works are based on System S and their aim is to provide solid foundation for modeling and predicting resource usage of stream programs which is different from our motivation of performance characterization.

Arasu *et al.* described a stream data management benchmark [3] which has been originally used by members of Aurora [1] and STREAM [15] projects to compare performance characteristics of Data Stream Management Systems (DSMS) (i.e., Stream Processing Systems). Recently Zeitler *et al.* implemented the Linear Road benchmark on SCSQ DSMS [18]. However, in this study our approach does not concentrate on single concrete benchmark. Rather we use a collection of applications which is a distinguishing point of our work from their's. Yet we see Linear Road as a possible avenue for extending our work.

On their paper introducing S4 [9], Neumeyer *et al.* have conducted two experiments (online and offline) on S4 with a Streaming Click-Through Rate computation application. While they conducted their experiments maximum at 20000 input events per second rate; we conduct the experiments with several magnitudes higher input data rates to compare performance of S4 with respect to System S.

3 An Overview of Stream Processing Software

System S, S4, and Esper are three popular stream processing software in use. We provide brief introduction to each of them below.

System S is an operator-based, large-scale distributed data stream processing middleware [7]. The project was initiated in 2003 and is currently under development at IBM Research [17]. System S uses an operator-based programming language called SPADE [6] for defining data flow graphs. SPADE has a set of built-in operators (BIOP) and also supports for creating customized operators (i.e., User Defined Operators (UDOP)) which allows for extending the SPADE language. Communication between operators is specified as streams. SPADE compiler fuses operators into one or more Processing Elements (PEs) during the compilation process. System S Scheduler (see Figure 1 (c)) makes the connection between PEs, when the application is run in a stream processing cluster. Out of the BIOPs used for implementing the sample programs Source creates a stream from data coming from an external source. Sink converts a stream into a flow of tuples that can be used by external components. Functor performs tuple-level manipulations (e.g., filtering, mapping, projection, attribute creation, transformation, etc.). Aggregate groups and summarizes incoming tuples. Split splits a stream into multiple output streams.

S4 is an open source operator-based stream processing system released by Yahoo Inc. in October 2010 [9]. S4 follows Actors model which makes it considerably different from System S. S4 has a decentralized and symmetric architecture where all nodes share the same responsibilities. Furthermore, S4 uses a pluggable architecture (see Figure 1 (a)) that keeps the design generic and customizable to a greater extent.

In S4 the computations are performed by PEs. Communication between the PEs is done in the form of data events. A sequence of events in S4 is defined as a stream. Event emission and consumption is the only mode of communication between PEs. Current version of S4 provides several PEs for standard tasks such as count, aggregate, join, etc. However, custom PEs can be easily created by extending the classes provided by S4 API. All PEs consume exactly the events which correspond to the values on which they are keyed.

Programming model of S4 has been created in such a way that developers write PEs in Java programming language, and the PEs are assembled into applications using Spring framework. Developers need to essentially implement input event handler `processEvent()`, and `output()` which implements the output mechanism [9].

Esper is a software component for stream processing developed by EsperTech Inc. [4]. The software is available in Java with the name Esper, and in.NET as NEsper. However, this paper uses only Esper since we wanted to compare performance of S4 with a Java based stream processing software. One of the important differences of Esper from System S and S4 is that it is just a software library. Therefore, important features of stream processing systems such as distributed processing, fault tolerance, etc. has to be coded manually. However, since Esper is a software library it can be easily integrated with variety of applications such as J2EE web servers, distributed caches, web browsers, etc.

Since Esper is just a software library we implemented the distributed event processing functionality by using Esper with ActiveMQ [12] which is an open source message broker implementation of the Java Messaging Service (JMS) specification.

Software architecture of the three stream processing software is shown in Figure 1.

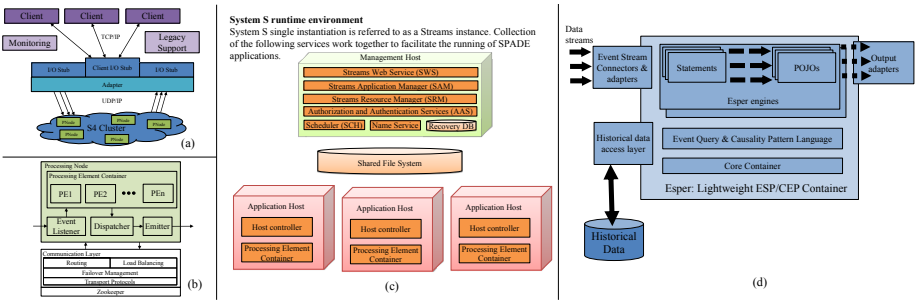

Fig. 1. Software architecture of the three stream processing software. (a) S4 Framework overview (b) A processing node of S4 (c) System S runtime environment (d) Esper's architecture.

4 Methodology and Performance Metrics

Our methodology is based on two level benchmarking. We use a micro-benchmark to get basic characterization of the three software's performance. Next, we use three different stream programs (Application-Specific Benchmarks) which are used for different purposes. These programs, the reasons for choosing them, and the features of their associated data sets are described below.

4.1 Sample Programs and Data Sets

Micro-benchmark. We use a three operator micro-benchmark program to get a basic understanding of the behavior of the stream processing software. This program's structure is shown in Figure 2 (d). It has only one functor (F1)

that increments an integer value it receives from the Source operator (S). The result from F1 gets stored in `results.dat` via Sink operator (SI). We chose this program for our study because of its simple nature which allows us to reveal the behavior of the stream processing systems during heavy work loads. We used a synthetic data set of two digit integer values (in CSV format) for our experiment. The data set contained 10 million records.

VWAP. Calculation of Volume-Weighetd Average Price (VWAP) is a real world application scenario of stream processing in the financial services domain. VWAP is calculated as the ratio of the value traded and the volume traded within a specified time horizon. This can be depicted as,

$$VWAP = \frac{\sum_{i=1}^{n}(P_i.V_i)}{\sum_{i=1}^{n}(V_i)}$$

where V_i represents each traded volume and P_i represents the corresponding traded price in a series of n transactions. Figure 2 (a) shows the data flow graph of the VWAP application used for performance evaluation. This application is part of a larger financial trading application described at [2]. The first functor (F1) filters the tuples for valid records and the aggregate operator (AG) keeps a tuple window of 4 based on the ticker id it receives from F1. Furthermore, it calculates the $P_i.V_i$ portion of the VWAP formula for each tuple. AG outputs a new tuple each time it receives a tuple from F1. Finally, F2 calculates the VWAP value and transfers to Sink operator (SI) which stores the result in `result.dat` file. We used a dataset that is available with System S's sample VWAP application; but magnified it to a larger data set of 1 million tuples.

CDR. Call Detail Record (CDR) is a piece of information produced by a telephone exchange containing details of a phone call passed through it. Telecommunication networks process massive amounts of CDR events, another application area for stream processing software systems. Furthermore, we wanted to test how well different stream processing software scale under massive data rates if programmers do not worry about producing optimized version of their codes. Considering these factor, we decided to use a four operator CDR processing application (shown in Figure 2 (b)) in our study to evaluate the three stream processing software. First functor (F1) splits the input tuples read by Source operator to different routes based on a hash value of call station ID (which identifies each user). The Aggregate operator (AG) calculates the total packet count by adding input and output packet values mentioned in the tuple. Finally, the result is stored in the disk via Sink (SI) operators. We used a synthesized data set with 2 million data tuples with each tuple having 22 fields for this experiment. The data set had information of 0.1 million users.

Twitter Topic Counter. As the third application specific benchmark we selected a Twitter hash tag count application. Twitter users can label the key words of their tweets using # (e.g., #car) which indicate to which conversations the messages relate to. As shown in Figure 2 (c) we developed a six operator SPADE program with similar functionality to the Twitter Topic Counter application (here onwards referred to as Twitter application) available with S4 and used both these applications during our performance evaluations. We used a real data set with 90507 tweets gathered from twitter on 4th June 2011 in the period 00:00-01:00 JST for this experiment.

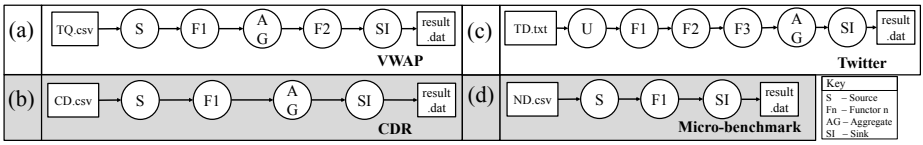

Fig. 2. Data flow graphs of sample applications

4.2 Experimental Setup

The experiments were conducted on 12 compute nodes each with AMD Opteron^{TM} Processor 242, 1600MHz 1MB L2 cache per core, 250GB hard drive. Seven out of twelve nodes had 8GB RAM (Nodes labeled as `sa0<n>`, n is from 1 to 7) while the remaining five (Nodes labeled as `sb0<m>`, m is from 1 to 5) had 4GB RAM. From the profiling results we obtained using Oprofile [13], and Nmon [10] we observed that the difference of main memory in the nodes did not affect our experiments. All the nodes were installed with Linux Cent OS release 5.4, IBM Infosphere Streams Version 1.2, and JRE 1.6.0; and were connected via 1Gigabit Ethernet. We used S4 version 0.3.0.0 and Esper version 4.5.0. We set both initial Java heap sizes and Maximum Java heap sizes to 3GB to avoid CPU time being spent on increasing Java heap memory during the experiments.

4.3 Performance Metrics

We used job throughput, CPU usage, network I/O, and memory usage of each sample application (i.e., job) as the metrics for the evaluations.

- Job Throughput : Measures number of input tuples processed by each stream processing system. On System S we measured this in Tuples per second while on S4 and Esper it was measured in Events per second.
- CPU usage (%): Overall CPU utilization during the experiments. This metric provides information on which nodes are busy processing data hence allows for identifying stream processing system's node level bottlenecks.

- CPU usage by process (%): Measures what processes contribute to overall CPU utilization and the amount of their contribution. This allows us to identify where the performance bottlenecks exist (i.e., in stream processing system level or OS level) and which processes are responsible for such behavior.
- Memory Usage (MB) : The amount of main memory (RAM) use during application execution. It is essential to maintain high percentage of free main memory for proper functioning of stream applications. Memory usage is one of the important metrics for this study because unlike most batch processing tasks stream processing keeps all the required data in main memory during the program execution.
- Network I/O (KB/s) : We measured network I/O of each node of stream processing system cluster during experiments to quantitatively understand what kind of communication overhead exists between the nodes.

4.4 Objectives and Methodology

Measurement of throughput was done in two different ways for S4, Esper, and System S. In S4, and Esper we measured time required to process a specific amount of events while in System S we measured time between processing first tuple and a last tuple in order to increase the accuracy of the results.

By measuring throughput our intention was to understand how well the applications scale in both stream processing systems. Scalability of current S4 applications are characterized by S4 runtime. The runtime determines what number of PEs to be created to suit for a particular node allocation under different workloads. We cannot explicitly allocate PEs for different nodes. Hence we utilized the automatic node allocation for S4. However, SPADE allows for allocating a node pool (a collection of hosts in a stream processing cluster) and attaching different operators to specific nodes programmatically. We used this feature to distribute workload of System S jobs among nodes. The Esper applications we developed had to be manually allocated for different nodes giving us more finer grain control over their distributed execution. Furthermore, we augmented the sample programs shown in Figure 2 in such a way that the modified versions allow for attaching more nodes with the job. The objective was to use more nodes than the operators (PEs) available in the data flow graph and allocate multiple operators to each node. E.g., If we used 12 nodes we need a data flow graph with at least 12 operators which is not possible with any sample program layout shown in Figure 2. How the augmentation of data flow graphs of sample applications developed for System S has been carried out is shown in Figure 3.

Apart from the four augmented sample applications we used an optimized version of the CDR application (shown in Figure 3 (c)) in order to observe how introduction of multiple sources affects the throughput of a stream program. However, we used this program only for this purpose but used normal augmented version of CDR application (in Figure 3 (b)) in order to make the comparison a fair one. In order to supply four source files to CDR Optimized data flow graph the data file used for CDR was splitted equally in to four files using Linux's

`split` command and resulting files were checked to make sure that the first and last packets were in the correct format. Note that we do not do any optimization of the workload across different nodes in our experiments in order to make a fair comparison between the three systems.

As shown in Figure 3 we introduced a split operator (SP) for each sample program that splitted the data stream from source operator to several sub graphs. The splitting was based on a hash value produced for each incoming packet. We made sure the packets are evenly splitted among sub-graphs by choosing an appropriate filed to hash and by generating the input data file with proper distribution of tuples. Furthermore, we introduced two Functor operators (FS (at the beginning) and FE (at the end)) which recorded the time of receiving start packet and a desired last packet.

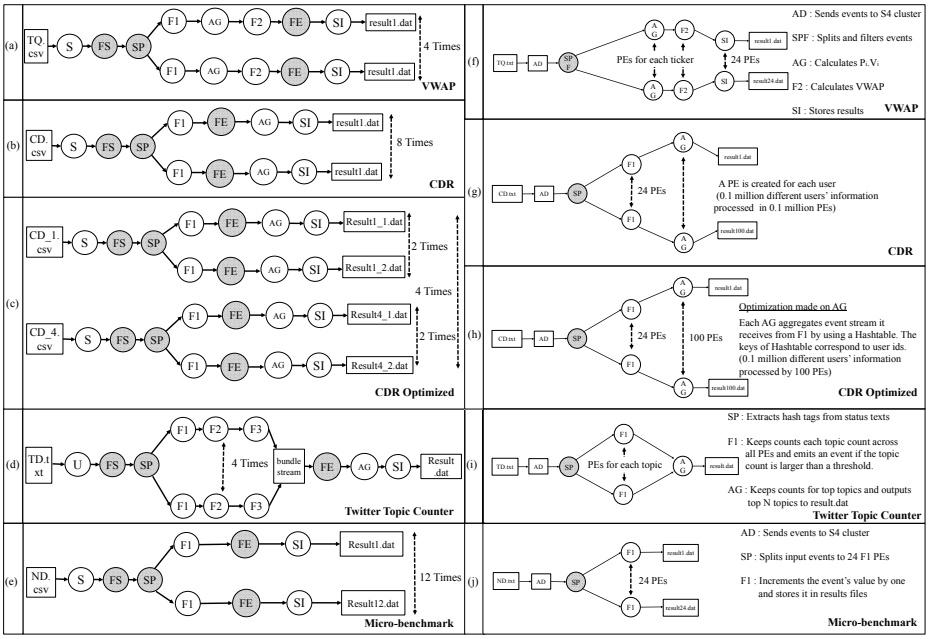

Fig. 3. Augmentation of sample application data flow graphs on System S and S4. Sub-figures (a) to (e) correspond to System S application while (f) to (j) correspond to S4.

In the case of VWAP we used a data set of 1 million data tuples and we attached last packet with trade ticker "GMD". The time it took to receive this packet at one of the four PEs was taken as the total runtime and throughput was calculated. For CDR the input data file had 2 million data tuples. We measured the time it takes to receive the last packet (i.e., 2 millionth packet) by one of the FEs. The same method was employed to measure the runtime of CDR optimized version.

The data flow graph of Twitter application is significantly different from other augmented versions because we wanted to keep the similarity of program design with S4 version of Twitter application. It should be noted that in CDR, CDR Optimized, and Twitter applications measurement of the last packet was done before the Aggregate operators. It is because Aggregate operators output only a subset of tuples they receive. The measurements of the throughput was conducted on the corresponding Aggregate operators on S4 versions; hence this ensured that the measurements were taken on the same operator on both System S and S4.

How the sample applications have been implemented on S4 is shown in Figure 3 (f) to (j). For data flow graphs shown in Figure 3 (g), (h), (i), and (j) the storing of results was done without use of a Sink (SI) PE because we can store the results by using the code of last operator. However, we defined a separate SI PE for VWAP since data output involved an additional step of conversion to CSV format. Furthermore, as shown in Figure 3 (g) and (h) we developed two versions of CDR for S4 since the original version shown in Figure 3 (g) resulted in inefficient use of system resources by spawning 0.1 million PEs. The solution was to reduce the number of PEs to 100 by using a hash table on each PE with user id as the key. All the PEs were distributed equally to all nodes during the experiments that ran on S4.

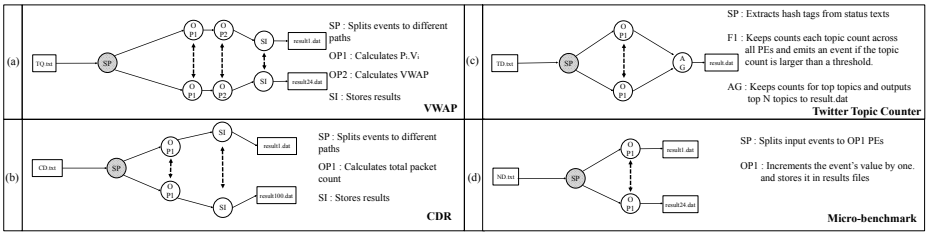

Fig. 4. Augmentation of sample application data flow graphs for Esper

Augmented versions of the Esper applications are shown in Figure 4. Each operator denoted by OP (e.g., OP1) is attached to a JMS server running on the same node.

For each augmented sample program we allocated different number of operators per node as the number of worker nodes involved in the processing vary. Each version of the sample program was run three times for both System S, Esper, and S4. Average value of the running times was taken for calculating the throughput to improve the accuracy of end results. Furthermore, we ran both Nmon and Oprofile on each node associated with the experiment well before one of the experiments during three experiment runs begin. The two daemons were shutdown after the experiment completed keeping enough delay from the experiment end time.

5 Performance Evaluation

5.1 Job Throughput

The throughput results obtained from running the four sample applications and the micro-benchmark based on System S, S4, and Esper are shown in Figure 5 (a), (b), and (c) respectively. Note that a tuple in System S corresponds to an event in S4 and Esper. Since most research literature on System S uses the term tuple to represent a data event we use the same terminology in our work. The performance curves on Figure 5 (b) shows that S4 achieves sub-linear scalability for CDR, VWAP, and micro-benchmark because doubling the number of nodes did not result in a corresponding doubling of the system's performance. We have drawn two different curves for CDR on S4. The curve marked with CDR corresponds to the data flow graph shown in Figure 3 (g) which is a naive unoptimized version while the curve CDR Optimized is obtained by running the optimized program shown in Figure 3 (h). The latter was created to avoid inefficient resource usage as described in Section 4.4. Twitter application indicated super-linear scalability because in each case shown in Figure 5 (b) increasing the number of nodes increased the system performance more than twice.

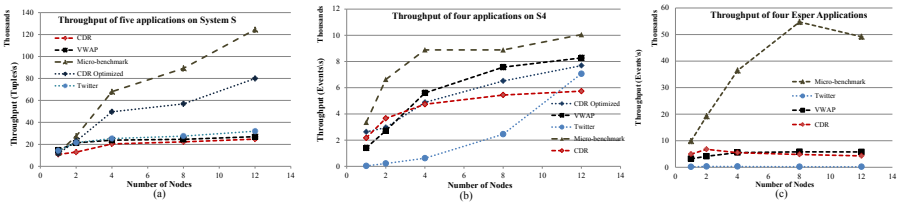

Fig. 5. Throughput comparison of sample applications. (a) System S Applications (b) S4 Applications (c) Esper Applications.

Out of the System S versions of these applications; micro-benchmark and CDR Optimized version produced almost linear throughput curves. CDR, VWAP, and Twitter applications indicated sub-linear scalability (saturated). However, considering the throughput characteristics obtained from optimized version of CDR, it is apparent that having a single source operator in the data flow graph results in such sub-linear scalability.

Esper reported more than 1.5 times higher performance for single node applications compared to S4. This indicates that Esper is much suited for single node event processing scenarios. However, the distributed versions of the Esper applications did not scale well compared to S4 or System S. We believe the reason for such less scalability was because of the overhead associated with object serialization.

In order to get more details of System S, S4, and Esper (on single node) we used Nmon and Oprofiler tools. We wanted to get profile information of Esper

on single node setting because it reported best performance on single node. The results are discussed below.

5.2 CPU Usage

CPU utilization of manager nodes and worker nodes of both S4 and System S are shown in Figure 6. We refer the node on which the S4 adapter component run during the experiment as Manager node and the remaining nodes as Worker nodes in the case of an S4 cluster. The node on which Streams Application Manager (SAM) [7] resides is referred as Manager node and the other nodes are taken as Worker nodes.

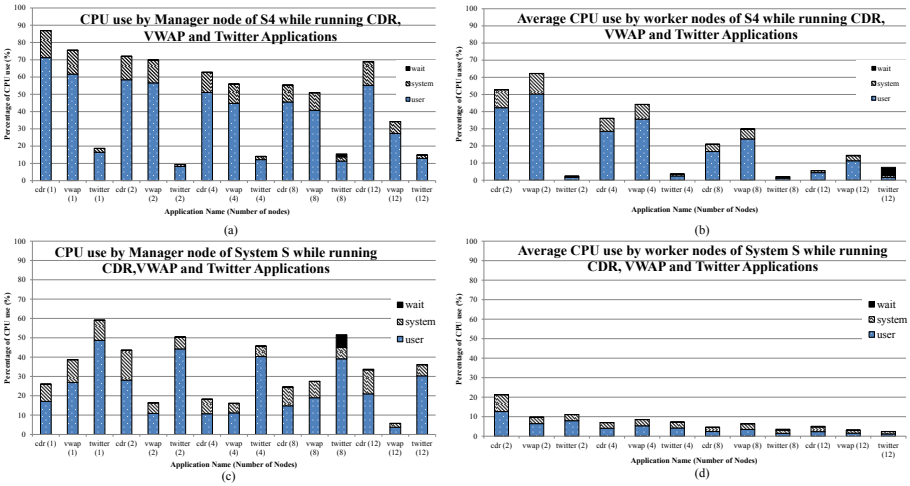

Fig. 6. CPU usage on different nodes of System S and S4 while running CDR, VWAP, and Twitter applications

To increase the legibility of results and to distinguish between application-level and micro-level benchmarks we separately list the results of micro-benchmark experiments' CPU utilization in Figure 7.

It was clear that System S's workers used (in Figure 6 (c), (d)) less CPU compared to S4 with VWAP and CDR applications. However, in the case of Twitter which operated relatively low input data rate S4 workers and manager node reported relatively less CPU usage compared to their System S counter parts. Furthermore, in System S a considerable amount of processing has been performed by the system processes rather than by user processes as can be observed from S4.

Esper on single node reported average CPU usage of 83.4%, 72.1%, 66.2%, 50.8% for VWAP, Twitter, CDR, and micro-benchmark respectively.

We used Oprofile to identify what happens in system and user processes in the case of the CDR and micro-benchmark experiments with 12 nodes for both

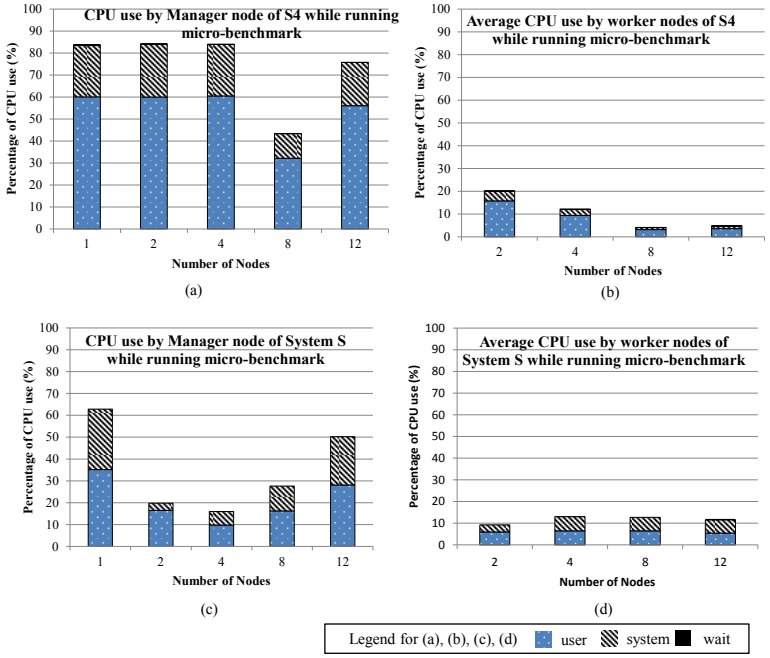

Fig. 7. CPU utilization by nodes when running micro-benchmark on System S and S4

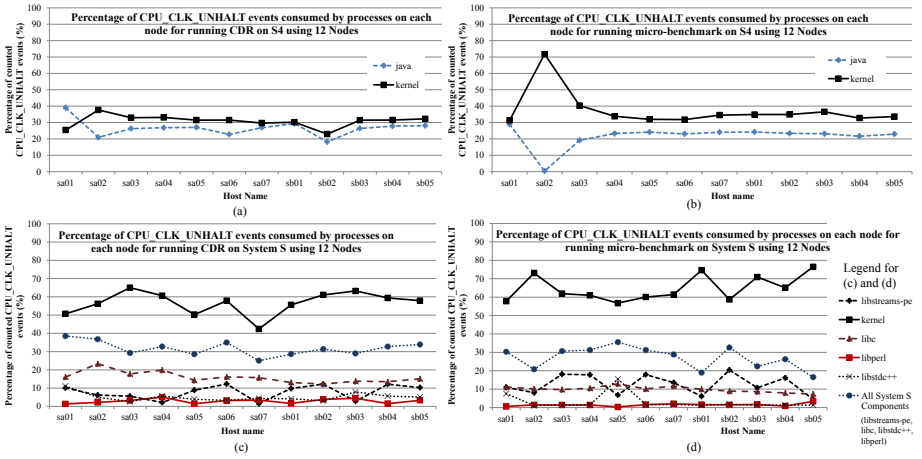

Fig. 8. Breakdown of CPU usage by processes for CDR and Micro-benchmark

System S and S4. The results are shown in Figure 8. Each graph of Figure 8 shows the share of `CPU_CLK_UNHALT` events used by different processes. It can be observed that in both experiments involved with S4 (Figure 8 (a), (b)), in most of the nodes the Java virtual machine (java) had used slightly less than 30% of `CPU_CLK_UNHALT` events which is little less than the amount consumed by the operating system's kernel. However, in the case of System S when running both CDR and micro-benchmark the System S components had consumed roughly 20% to 30% `CPU_CLK_UNHALT` events. However, unlike the case with S4 nodes' kernel consumed 50% to 60% `CPU_CLK_UNHALT` events. This gives an explanation for why we saw considerable amount of processing conducted by system processes in Figure 6.

5.3 Memory Usage

Memory consumption of different nodes while running the four augmented applications on S4 and System S are shown in Figure 9. It can be observed that worker nodes of System S consumed less memory compared to S4 workers. Twitter application introduced relatively less data transfer rate among the nodes since it only injected 90507 events. This is a reason for why S4 nodes indicate less memory use compared to other applications. Huge amount of memory consumption can be observed on manager node (sa01). It should be noted that the node named sa01 was used as the master node in all the experiments carried out in this paper. Memory usage during Esper run on single node was 7.7GB, 5.5GB, 6.1GB, and 7.3GB micro-benchmark, Twitter, VWAP, and CDR respectively.

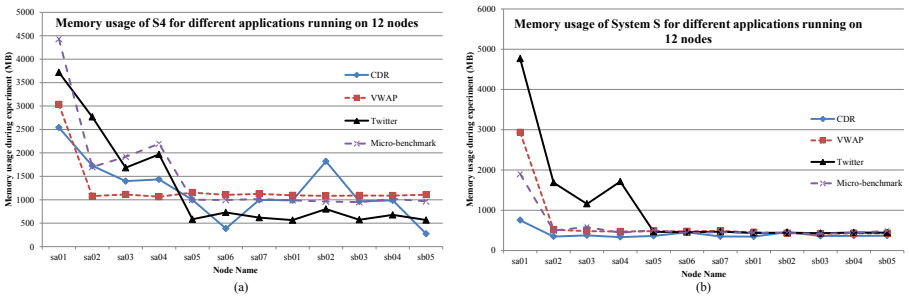

Fig. 9. Memory consumption of sample applications on System S and S4

5.4 Network I/O

The network I/O happening between nodes of a stream processing system is a key criterion that determines its performance. Figure 10 shows the network I/O of different nodes when running CDR application on both S4 and System S. Note that we compare only the external communications happening through Ethernet port (`eth0`) rather than through the loop back interface (`lo`). When considering

the `eth0-write` at master nodes and `eth0-read` at worker nodes it is apparent that the amount of data transferred between the master and the worker nodes is considerably larger in S4 compared to System S. For the data shown in Figure 10 the average `eth0-write` rate of Manager node of S4 is 60% larger than the average `eth0-write` rate of System S's Manager node. Moreover, average rate of `eth0-read` at worker nodes of S4 is 120% larger than average `eth0-read` at worker nodes of System S. However, both System S and S4 applications used the same 2 million data set. This indicates that S4's data transfer protocol consumes more memory bandwidth compared to System S. The network I/O for eight nodes while running CDR for both S4 and System S (shown in Figure 10 (c) and (f)) confirms the aforementioned fact.

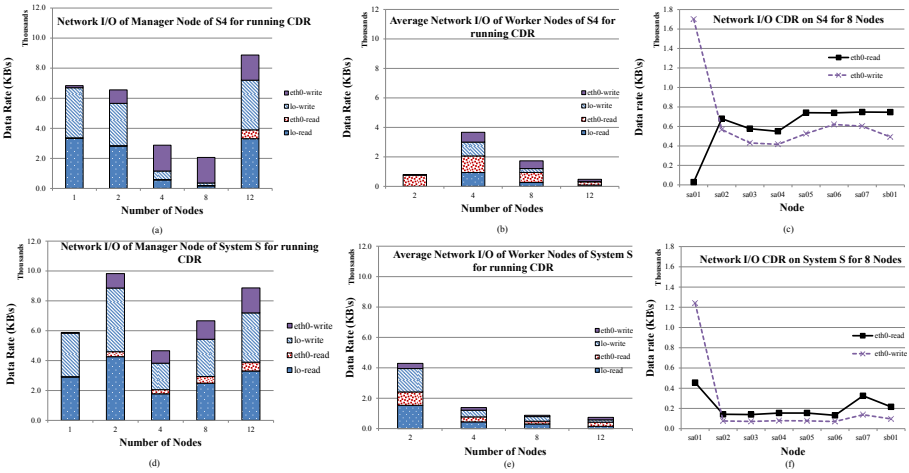

Fig. 10. Network I/O for Manager node and Worker nodes while running CDR on S4 and System S

6 Discussion

The results we obtained by running ten different applications (totaling 70 applications considering 1,2,4,8,12 node scenarios including System S and S4 source code optimization) on System S, S4, and Esper gave us sufficient insight to their internals. It became clear from the throughput comparison made on Figure 5 that the stream programming models that allow/require programmers to write optimized codes should be used carefully to maximize the throughput. E.g., A SPADE program written with single source operator might not scale well in different hardware configurations. Also a S4 application that generates huge numbers of PEs for incoming events cannot scale well with limited `PEContainer` queue size. Yet introduction of multiple source operators resulted in a 3.2 times

speedup for CDR application on System S and a 1.34 times speed up for reducing 0.1 million Aggregator PEs of S4 to 100 PEs (See Figure 5) which indicated possible avenues for performance improvements in different stream programming models.

While Java based stream processing system architectures are gaining considerable attention due to their portability, system designers and programmers have to think carefully before choosing the desired solution. E.g., A light weight input event rate job could be easily processed using S4 with few amount of PEs (E.g., Twitter application run on S4). However, a large scale application with high commercial importance such as the VWAP and CDR might produce millions of PEs since S4 dynamically generates PEs for each new data events it receives. As with any other JVM related optimizations, setting of maximum and minimum heap size values plays a key role in determining the performance of Java based stream processing systems. However, in the case of S4 and Esper based stream processing system administrators need to be vigilant about the characteristics of the data handled by their S4/Esper applications properly (See Figure 9 (a)).

Esper applications' throughput results indicate that scalability is one of the key challenges faced by JMS based distributed Esper applications. While there was a slight scalability advantage scaling from one node to two nodes for all the benchmarks (See Figure 5), the performance tend to degrade when the application is scaled to more nodes. We believe the reason for such behavior is slowness in network communication and serialization. However, the Esper micro-benchmark application had considerably higher performance compared to its S4 counterpart.

While we observed heavy use of network bandwidth by S4, by using optimized protocols and techniques such as Java New I/O, InfiniBand Remote Direct Memory Access the conditions could be improved.

7 Conclusion

In this paper we presented a performance study on three stream processing software. We used three popular stream processing software: IBM System S, Yahoo S4, and EsperTech's Esper. We ran three application benchmarks covering different domains of stream processing. We used a micro-benchmark to further clarify performance of the software systems. The study used job throughput, CPU usage, memory usage, and network usage of each node as the performance metrics. By analyzing the throughput and profiling results we observed that carefully designed stream applications result in high throughput. Another conclusion we arrived at is that choice of a stream processing system need to be made considering factors such as performance, platform independence, and size of the jobs. Furthermore, we understood the importance of key role played by operating system kernel in stream processing system's performance.

A stream processing system architecture that scales in terms of number of PEs is a further work that is inspired by this work. In future we hope to extend this work to a code level performance study on S4, specially to identify which components, code segments are most resource intensive.

Acknowledgments. This research was supported by the Japan Science and Technology Agency's CREST project titled "Development of System Software Technologies for post-Peta Scale High Performance Computing".

References

1. Abadi, D.J., et al.: Aurora: a new model and architecture for data stream management. The VLDB Journal 12, 120–139 (2003)
2. Andrade, H., et al.: Scale-up strategies for processing high-rate data streams in systems. In: ICDE 2009 (2009)
3. Arasu, A., et al.: Linear road: a stream data management benchmark. In: VLDB 2004, pp. 480–491 (2004)
4. EsperTech. Esper - Complex Event Processing (February 2012), http://esper.codehaus.org/
5. Etzion, O., Niblett, P.: Event Processing in Action (2011)
6. IBM. Ibm infosphere streams version 1.2.0.1: Programming model and language reference (February 2010)
7. IBM. Ibm infosphere streams version 1.2.1: Installation and administration guide (October 2010)
8. Mendes, M.R.N., Bizarro, P., Marques, P.: A performance study of event processing systems. In: Nambiar, R., Poess, M. (eds.) TPCTC 2009. LNCS, vol. 5895, pp. 221–236. Springer, Heidelberg (2009)
9. Neumeyer, L., et al.: S4: Distributed stream computing platform. In: KDCloud 2010 (December 2010)
10. Nmon. nmon for Linux (June 2011), http://nmon.sourceforge.net
11. Parekh, S., et al.: Characterizing, constructing and managing resource usage profiles of systems applications: challenges and experience. In: CIKM 2009, pp. 1177–1186 (2009)
12. Snyder, B., Bosanac, D., Davies, R.: ActiveMQ in Action (2011)
13. SourceForge. OProfile - A System Profiler for Linux (June 2011), http://oprofile.sourceforge.net
14. Suzumura, T., Yasue, T., Onodera, T.: Scalable performance of systems for extract-transform-load processing. In: SYSTOR 2010 (2010)
15. The_STREAM_Group. Stream: The stanford stream data manager. Technical Report 2003-21 (2003)
16. Turaga, D., et al.: Design principles for developing stream processing applications. In: Software: Practice and Experience (August 2010)
17. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.-L., Fleischer, L.K.: SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 306–325. Springer, Heidelberg (2008)
18. Zeitler, E., Risch, T.: Scalable splitting of massive data streams. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA 2010. LNCS, vol. 5982, pp. 184–198. Springer, Heidelberg (2010)
19. Zhang, X.J., et al.: Workload characterization for operator-based distributed stream processing applications. In: DEBS 2010, pp. 235–247 (2010)