Application Design and Profiling of Stream Processing

Veronica Gil-Costa, Jair Lobos
Universidad Nacional de San Luis, CONICET
San Luis, Argentina
Email: {jlobos,gvcosta}@unsl.edu.ar

Mauricio Marin
DIINF, University of Santiago, Chile
Center for Biotechnology and Bioengineering, Chile
Email: mauricio.marin@usach.cl

***Abstract*—Stream processing has recently caught the attention of many researchers and engineers, mainly because of the continuous growth of information generated by users. Stream processing platforms allow processing and analyzing real time data, which helps to make decisions faster. In this work, we determine the most relevant tasks performed by the distributed stream processing platform named S4. To this end, we develop a pool of benchmark applications and we make a profiling of their execution using the S4 platform. Results show that the most relevant operations are related to the control and manipulation of threads.**

Keywords—stream processing; S4; profiling.

I. INTRODUCTION

The world has become fully connected. There is a large number and variety of data resources available from hardware and/or software systems. There are numerous industries where everyday processes and interactions with customers generate millions of events that produce traces, with information regarding user's activity. These traces contain valuable information for understanding and optimizing processes. In addition, those traces can be used to detect anomalies, to predict the behavior and trends of customers, among other activities that can improve the productivity of a company or institution.

The events are collected from users actions form a continuous amount of data stream. Some examples can be found in: market analysis; telecom call detail records; video surveillance systems; vital signs of a patient in a medical system; intrusion records system networks; the behavior in a system of Web 2.0, among others. In all these applications it is necessary to collect, process and analyze the data stream, and then generate results or produce some specific actions. An important feature of these applications is that the analysis must be done in real time.

There are many stream processing platforms such as SPC (Stream Processing Core) [2], Storm [3], Esc [4] and D-Stream [5]. Some research works like TimeStream [13], StreamCloud [14][15] and CEC [16], have endeavored to present solutions to the problems of load balancing and fault tolerance of the stream processing process.

Recently, a general-purpose distributed platform designed to analyze massive data processing called S4 (Simple Scalable Streaming System) was proposed by L. Neumeyer, et. al. [1]. The S4 world-view is that streams are passed through a graph (DAG) formed by processing elements (PEs), which are connected each other in a downstream manner. Each PE performs a given primitive operation on the received stream and generates output streams. Data is routed through the PEs by means of keys, which are specified by users.

In this work, we develop and test a set of applications covering different computation/communication aspects using the S4 platform. We aim to understand the flow of events processed in those applications with different data streams, to determine which are the most repetitive and costly tasks executed by the S4. We obtain relevant metrics by developing prototypes for each application, which are used as benchmark to detect bottlenecks in both communication and computation operations. The performance information obtained in this work can be used to propose improvements to the stream processing platform itself. By determining the relevant operations executed by the S4 platform and their costs, it is possible then to introduce these costs into a simulation model as the ones presented in [6] to design and test new algorithms without affecting the actual platform running in production. To this end, we developed a pool of benchmark applications built with different characteristics including processing and communication complexities in order to determine the most relevant operations.

Additionally, the results obtained though the benchmark applications can be used in elastic stream processing programming environments [4], where developers can detect possible bottlenecks of PEs, make decisions and take action in advance. In this case, the knowledge obtained by executing the pool of benchmark applications, can aid to determine which PEs should be replicated.

This paper is structured as follows. Section 2 describes stream processing and the S4 platform. Section 3 briefly describes profiling and the tool used in this work. Section 4 describes the pool of benchmark applications used to detect the most relevant operations. Section 6 shows the results. Finally, Section 7 presents the conclusions and future work.

II. STREAM PROCESSING

In this section we discuss the main properties of stream processing, when stream processing makes sense, and how it fits into big data architectures. We also describe the S4 stream processing platform, used to test the benchmark applications presented in this paper.

A. Streaming Processing and Big Data

Big data is commonly defined as the three Vs: Volume, Velocity and Variety [17]. It is used to describe the exponential growth and availability of structured and unstructured data. A more recent, definition states that "Big Data represents the

Information assets characterized by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value" [18].

On the other hand, stream processing is used for fast data requirements, which includes tacking the velocity of processing a huge variety of data in real time. Therefore, both big data and stream processing can complement each other.

Stream processing was first used for finance problems. Today, it is used in almost all industries where stream data is generated by human activities or automatically by sensors. Events are generated on-line in unpredictable time instants. The union of events forms a continuous stream of information that may have dynamic variations in intensity of traffic. In this context, the process used to store and organize/index events in a conveniently way to then process them in batch can be very costly given the huge volume of data and the amount of computational resources required for processing them. But even if this is feasible, it is often desirable or even imperative to process the events as soon as they are detected to deliver results in real time.

In particular, stream processing corresponds to a distributed computing paradigm that supports the process of gathering and analyzing large volumes of heterogeneous data streams to assist decision making in real time.

Stream processing appears as result of the rigorous data management, which is increasingly demanding because of the information generated by business and scientific applications, which are fully linked to the technological progress. It is also related to the advance in hardware and software databases; the management of large amount of data in distributed systems; the use of techniques such as signal processing, statistics, data mining and optimization theory.

Stream processing aims to process data in real time and in a fully integrated way, to provide information and outcomes for consumers and/or end users. Also, it aims to integrate new information to support decision making in the medium and long term.

The high volume of event flows coming from different data sources makes it impossible to store this information, such as model-based on data warehouse where all the data is stored and then to make the appropriate processing and analysis.

Stream processing applications requires fulfilling certain performance requirements in terms of latency and throughput. Specifically, processing must keep up with the rate of incoming data, while it provides a high level of quality of analysis of results as fast as possible. Additionally, the application components and infrastructure must be fault-tolerant.

B. *S4 - Simple Scalable Streaming System*

S4 acronym for "Simple Scalable Streaming System" is a system of general purpose, distributed, scalable, which allows applications to process data flows continuously without restrictions [1]. S4 is inspired by MapReduce [7], designed in the context of data mining and machine learning algorithms of Yahoo! Labs for on-line advertising systems.

In S4, each event is described as a pair (key, attribute). PEs are the basic units and messages are exchanged between them. The PEs can send messages or post results. PEs are allocated in the so-called processing nodes (PNs) servers. The PNs are responsible for: a) receiving incoming events, b) routing the events to the corresponding PEs and c) dispatching events through the communication layer. The events are distributed using a hash function over the key of the events. Furthermore, the communication layer uses Zookeeper [8], which provides management and automatic replacement clusters if a node fails.

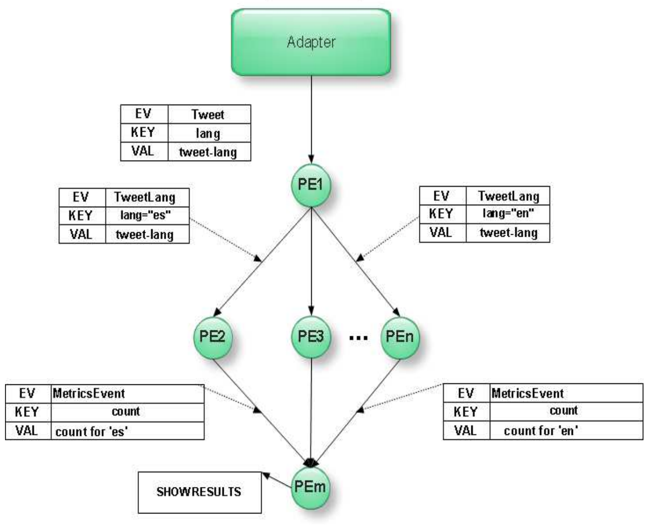


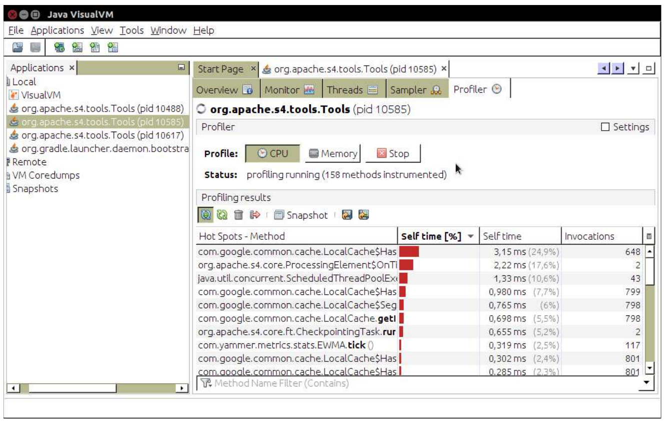
Figure 1. S4 application design (lang-count example).

To run an application with S4, we need to deploy an Adapter application. Adapters are S4 applications that can convert external stream into stream of S4 events. Figure 1 shows a simple Tweet language count for Twitter. In this example, input events contain a language descriptor for a tweet from Twitter. The Adapter gathers tweets from twitter and filters only the language descriptor. Then, the Adapter sends an event to PE1. PE1 listens for Tweet events with all possible keys. For each possible key, PE1 emits a new event of type TweetLang. PE2-n listen for TweetLang events emitted with the key lang. For example, PE1 emits an event with key lang=”es”. PE2 receives all events of type TweetLang keyed lang=”es”. If the PE corresponding to the emitted key exists, the PE is called and the counter of language is incremented. Otherwise, a new PE is instantiated and linked to the new key. Whenever a PE increments its counter, it sends the update count to the PE called PEm and this show the results.

III. PROFILING AND TOOLS

A profiler generates the division of the logical structure of the applications so that user can understand how a particular run of the application is performed using relevant information regarding execution time and memory usage.

Using a system profiler we can obtain a model to predict scaling factors as characteristic functions of the applications and hardware parameters [9]. Currently, there are several tools available to perform system profiles. S4 requires a JAVA profiler, among which we can highlight: Profiler4j [10] jvisualvm [11] and Java Profiler Tool [12]. By using these tools and the applications described in the next section, we intend to obtain a S4 profile to determine which are the most costly and the most relevant operations executed by the S4 stream processing platform.


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
org.apache.s4.tools.Tools (pid 10488)
org.apache.s4.tools.Tools (pid 10585)
org.apache.s4.tools.Tools (pid 10617)
org.gradle.launcher.daemon.bootstra
Remote
VM Coredumps
Snapshots
Start Page
org.apache.s4.tools.Tools (pid 10585)
Overview
Monitor
Threads
Sampler
Profiler
Settings
CPU
Memory
Stop
Snapshot
Method Name Filter (Contains)

org.apache.s4.tools.Tools (pid 10585)

Profiler

Profile: CPU Memory Stop

Status: profiling running (158 methods instrumented)

Profiling results

Hot Spots - Method	Self time [%]	Self time	Invocations
com.google.common.cache.LocalCache$Has		3,15 ms (24,9%)	648
org.apache.s4.core.ProcessingElement$OnTi		2,22 ms (17,6%)	2
java.util.concurrent.ScheduledThreadPoolEx		1,33 ms (10,6%)	43
com.google.common.cache.LocalCache$Has		0,980 ms (7,7%)	799
com.google.common.cache.LocalCache$Seg		0,765 ms (6%)	798
com.google.common.cache.LocalCache.**getI**		0,698 ms (5,5%)	798
org.apache.s4.core.ft.CheckpointingTask.**rur**		0,655 ms (5,2%)	2
com.yammer.metrics.stats.EWMA.**tick** ()		0,319 ms (2,5%)	117
com.google.common.cache.LocalCache$Has		0,302 ms (2,4%)	801
com.google.common.cache.LocalCache$Has		0,285 ms (2,3%)	801

Figure 2. Snapshop of the jvisualvm tool.

Figure 2 shows the environment of the jvisualvm tool when measuring the CPU utilization. The CPU Profiling is used to test the performance of the application and it gives detailed information about the total execution time and the number of calls for each method. In the same way, the Memory Profiling is used to analyze the memory usage, by showing the total number of executed methods or objects and the amount of bytes assigned to each one.

IV. BENCHMARKS APLICATIONS

This section describes a pool of applications developed to run on the S4 platform to perform tests in order to obtain a system behavior profile. These benchmark applications will help us to determine the most costly and most relevant operations. Each application has different levels of complexity on the tasks performed and different levels of communications. The applications described below can be classified into the following categories: 1) High communication, 2) dynamic creation of processing elements, 3) high, medium and low computation. Though this classification, we obtain the S4 operation costs for each type of benchmark application.

A. *Ping-Pong*

Figure 2 shows a basic communication structure between two processing elements. PE A, named “sender”, generates a new message and sent it to PE B, named “receiver”. Finally B replies this message to A. This benchmark program uses different messages sizes. Each event uses messages between 8 and 256 characters size. This application is classified as low computation but with high communication between the processing elements.

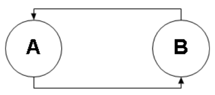


Figure 3. Ping-Pong application.

B. *Router*

The next application is called “Router”. This application generates random values in the Adapter module. The Adapter sends messages to the PE R, which determines if this value is an even or an odd number. If the received number is even, R sends a message (an event) to the Processing Element named Even. Otherwise, if the number is odd, R sends the event to the Processing Element named Odd. Both Odd and Even PEs make a count of the received elements. When all messages are dispatched, each PE sends its results to the Processing Element named Res, and this PE shows the final results. Figure 3 shows the flow of events and the PEs of this application. This application is classified as low computation but with high communication between the Adapter and the PEs.

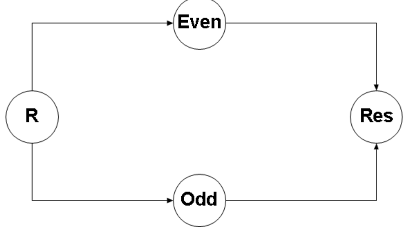


Figure 4. Router application.

C. *Counter of Tweets and Re-Tweets*

This application works with the Twitter API to get tweets from “Twitter’s global stream of Tweet data”. This application is connected to the data repository to extract tweets, which are processed by the Adapter. The Adapter receives the tweets, creates an event and sends them to the PE named T. This last PE classifies the tweet as “No-Re-Tweet” when it corresponds to message that has been posted for the first time, or “Re-Tweet” when it corresponds to a message that has been re-posted by other users.

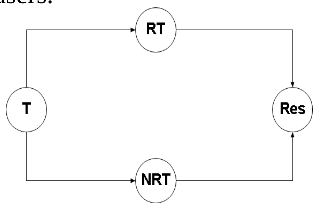


Figure 5. Counter of Tweets and Re-Tweets application.

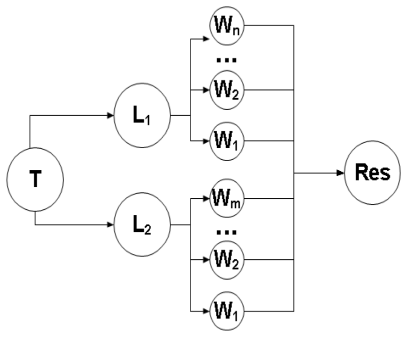


Figure 6. Language word counter application.

Once the classification process is finished, the message is sent to the corresponding PE (RT or NRT), which extracts the list of hashtag (represents an idea, it is considered as metadata), and stores the five most frequent hashtags. This information is sent to the Processing Element named Res. The Res PE summarizes the results received. Figure 4 shows the corresponding diagram for this benchmark application. This application is classified as *high communication* and as *medium computation*.

D. *Language word counter*

This application works with Twitter tweets. The differences with the previous application are that the event classification process is done by tweet language instead of No-Re-Tweet or Re-Tweet classification; and this application counts the number of words that has a tweet. With this process we stress the system because an additional PE is generated for each new word entered into the system. Namely, dynamic PEs are created every time a new word is found inside a tweet. Figure 5 shows the diagram for this benchmark application. The Processing Element named T, classifies the events into two groups L1 and L2, corresponding to languages Spanish and English. Each Processing Element used for language classification gets tweet and splits them into words. Each word is sent to its corresponding Processing Element W. If there is no PE for the received word a new PE is created. The words are counted in each PE and the results are sent to the Res PE. This application has a high communication cost and has dynamic creation of processing elements.

E. *Clasifficator for people's needs in a post-disaster scenario*

The classification process is composed of 4 steps: Recollecting, Filter, Relevance and Ranking. The recollecting step focuses on collecting data from the source to retrieve data from the Twitter API. The filter operator exploits a Naive Bayesian model to identify if tweets are objective or subjective. To create these models an automatic classification is performed through bags of positive and negative words. The bag of words were manually created by developers and validated by undergraduate students based on the information of other disaster tweets datasets. Objective tweets have a higher value, because they are more reliable than the subjective ones. If the tweet is subjective it is checked whether it is positive or negative in order to benefit the tweets based on the identified characteristics by applying weights constants in the ranking process.

The topic step is used to identify whether the information is coming from a trustworthy source or not. Trustworthiness is calculated in two dimensions: author information and tweet information. From the author side, information such as the number of tweets generated, the number of followers/followees and an account verification state are considered to calculate the reputation of the author. From the tweet side, the number of re-tweets, favorite marks, and the associated timestamp are exploited to calculate its reputation.

During the ranking step a normalization process of the obtained values is performed. This was computed every certain number of tweets, to get statistics such as the maximum number of followers, the number of favorites, etc. This data is used to normalize each tweet.

F. *DownStream Web Search Engine*

Typically, web search engines are composed by three services devised to quickly process user queries in an on-line manner: Front-Service (FS), Caching-Service (CS) and Index-Server (IS). In such systems a query submitted by a user goes through different stages. Initially, it is received by the FS, which redirects the query to the CS. The CS checks whether the same request has already been performed and verifies if the result (document IDs) are stored in the cache memory of the server. The CS can answer to the FS with a cache hit. In this case, the CS sends the query results to the FS, which builds the Web page with the query results and sends it to the user. Otherwise, if the CS sends a cache-miss to the FS, the FS re-routes the query to the IS, which will compute the top-k document results.

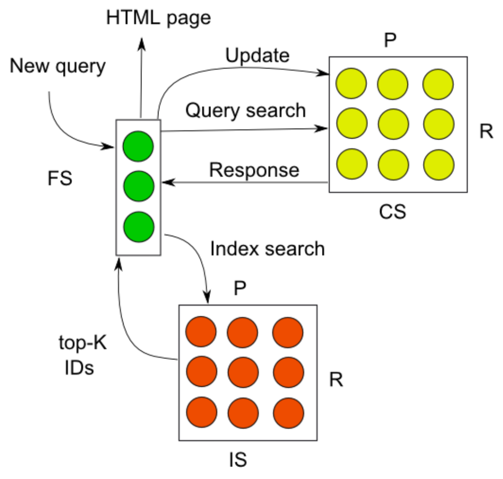


Figure 7. Components of a web search engine.

These services are deployed on a large set of processors forming a cluster of computers. They are implemented as arrays of P $\times$ R processors, where P indicates the level of data partitioning and R the level of replication of data. Hence, this architecture makes a high usage of partitioning and redundancy to enhance the query response time and throughput. For instance, each query is assigned a unique partition of the CS using a hash function, and different CS nodes can be associated with a partition to answer a query (see Figure 6).

Figure 7 shows a web search engine application designed for the S4 platform. The diagram represents the query flow through the web search engine components. Each component is composed by a set of different PEs. This application has the following structure: the FS is divided into three sub-services FS1, FS2 and FS3. Each group executes the tasks performed by the FS in different moments of the query processing process. In other words, the group named FS1, executes the tasks required to route an incoming query to the CS. The group named FS2, executes the tasks required to route the query to the IS, if no cache hit was reported, or the tasks required to build the query answer and send it back to the user, otherwise. The CS is divided into two groups CS1 and CS2. The first group detects cache hits and the second group updates the cache with query results. There is only one group of IS, because this service is used only once during the query process, to compute the top-k document results for queries. This application is classified as high in communication and high in computation.

V. RESULTS

In the following, we show results obtained by executing all benchmark applications described in the previous section. During each execution, we detect the most costly and relevant operations. The profiling execution was captured by the jvisualvm tool [11]. Results were obtained in a cluster of 16

64-bits CPUs Intel Q9550 Quad Core 2.83 GHZ and 4GB DDR3 RAM 1333 Mhz. Additionally, to verify the results, tests were also performed on an Intel I5-4200U 1.6 GHZ and 8GB DDR3 RAM 1600 Mhz.

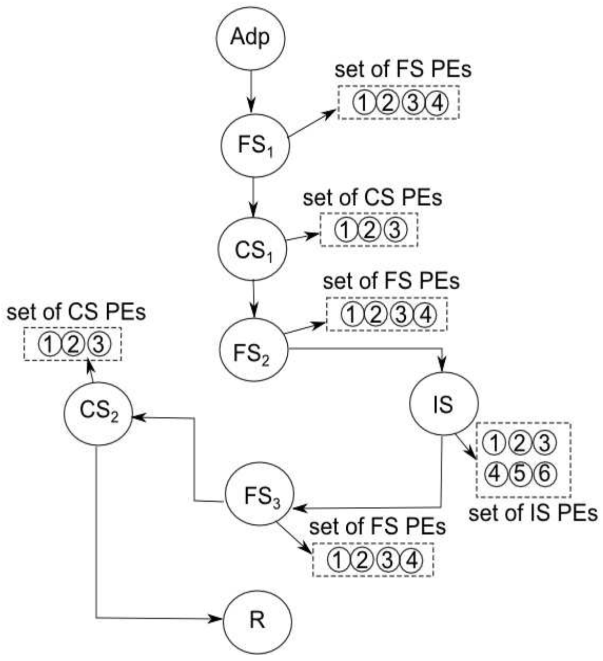


Figure 8. Web search engine diagram for the S4 platform.

TABLE I. COMMUNICATION OPERATIONS

Package	**Operations**		
	Class	***Method***	***Cost(ms)***
comm.staging	BlockingThread PoolExecutor Service	RunneableWithPermit Release.run	0.0019
		execute	0.0072625
comm.tcp	TCPEmitter	Init	--
		getPartitionCount	0.0005988
comm.tcp	TCPListener	EventDecoderHanler.messageReceived	0.0036462
comm.topology	**ZKRemote Stream**	**createStreamPaths**	**38.6**
		getPath	--
		getCollectionName	--
		addInputStream	**93.3**
		update	--
		refreshStreams	--
comm.topology	ZkClient	readData	--
		getChildren	--
comm.topology	ZNRecord	getSimpleField	--
		putSimpleField	--
comm.topology	PhysicalCluster	getNodes	--
comm.topology	Stream Consumer	Equals	--
		hashCode	--
s4.comm	DefaultHasher	hash	0.0002567

For the sake of simplicity, we present the average results obtained for each S4 operation using all benchmark application described in section IV. We cover all the S4 tasks.

A. Communication Operations

In this section, we detect and evaluate the most commonly and/or costly operations used for communication. Note that although S4 communication classes use objects and methods from other libraries or packages, we focus only on those belonging to the S4 platform.

Table I shows the S4 communication operations used to create nodes and to obtain information about clusters. TCP communications are started, creating path for the streams, adding streams and using a hash function to determine the route of events. Communication operations not relevant because of their low costs, do not have time costs in the fourth column of Table I. The results presented are average times obtained with all benchmark applications.

Results show that the most expensive communication operations are **addInputStream,** which publishes interest in a stream, by a given cluster and **createStreamPaths** which creates a zookeeper node to produce and consume streams.

TABLE II. COMPUTATION OPERATIONS

Package	**Operations**		
	Class	***Method***	***Cost(ms)***
Core	**S4Boostrap**	**run**	**1762**
Core	App	init	412
		start	32.4
		createInputStream	94.6
Core	ProcessingElement	handleInputEvent	0.0866573
		isCheckpointable	0.0008995
		recover	0.026
		getInstanceForKey	0.0647487
		setApp	--
		setName	--
core	Stream	StreamEventProcessingTask.run	0.1644673
		put	0.0200624
core	DefaultCoreModule	loadProperties	--
		configure	17.5
		provideTmpDir	--
core	ReceiverImpl	checkAndSendIf NotLocal	0.0034892
		receive	--
base	Key	addStream	--
		get	0.0076047

B. Computation Operations

The most important S4 computation operations are related to the initialization of objects, creation of communication objects for the communication layer, and related to control and manipulation of PEs and events arriving to the PEs. Table II shows the most relevant transactions.

Table II, shows that the Bootstrap is the most expensive process in terms of time consuming. This operation loads the application into main memory and starts it execution. Hence, it takes a larger time compared to others operations. The application Initialization takes a couple of milliseconds. The methods related with the processing elements creation do not take a significant execution time.

Table II, shows that the Bootstrap is the most expensive process in terms of time consuming. This operation loads the application into main memory and starts it execution. Hence, it takes a larger time compared to others operations. The application Initialization takes a couple of milliseconds. The

methods related with the processing elements creation do not take a significant execution time.

TABLE III. S4 MOST USES METHODS (STATIC VS DYNAMIC)

Operation	***Static***	***Dynamic***	***Diff***
Receive message	0,00029299	0,00087835	-199,78%
Create PE	0,02425	0,0182	24,94%
Set Key	1,518	0,021	98,61%

TABLE IV. EXTERNAL TASKS

Package	**Operations**		
	Class	***Method***	***Cost(ms)***
java.net	SocketInputStream	read	83.901729
java.net	SocketOutputStream	socketWrite	0.244189128
java.util.concurrent	**ThreadPoolExecutor**	**getTask**	**2213.0740740**
java.io	ObjectStreamClass	lookup	0.0001040
java.io	BufferedOutputStream	write	0.0000272
sun.nio.ch	EPollArrayWrapper	poll	160.5876465
java.util.concurrent.locks	LockSupport	parkNanos	665.0635514
		park	47.1500057

The operations concerning stream process have no big impact on the S4 platform. Although, these operations do not take much time, they must take into consideration the method that allows connection to Zookeeper for communications, as well as the method that checks whether the cluster, which will be used for communication, is local or not.

Table III shows results for the operations concerning to the most used methods between static and dynamic PEs creation. The operation of receiving a message is most expensive for dynamic PE creations applications than statics applications. However creating a PE and setting a key for each processing elements is most expensive for statics applications.

C. External Tasks

Table IV shows the external tasks used by the S4 platform. These methods are basically used for the manipulation, use of threads and invocations to methods of additional packages. Table IV shows the most relevant transactions obtained by running the benchmark applications. These methods are basically Java core packages (java.net, java.util and java.io) for manipulating stream. An external method to highlight is EpollArrayWrapper, which manipulates a native array of epoll event. Another important method is LockSupport, used for thread blocking with locks and synchronizations. The most costly operations is the getTask from ThreadPoolExecutor, which controls the blocking of tasks of the threads.

VI. CONCLUDING REMARKS AND FUTURE WORK

We presented a profiling study for the S4 Stream processing platform to determine the most costly and relevant operations. To the best of our knowledge, this is the first work concerning benchmark for streaming processing. We developed a pool of applications with different complexity. We used jvisualvm to make the profiling of the executions. Results show that the most relevant operations executed by the S4 platform are related to the creation of applications and the manipulation of events. The most costly operations are Thread control and Thread manipulation.

Future work includes the design and implementation of a simulator for the S4 platform, whose parameters will be set by the results obtained in this work.

ACKNOWLEDGMENT

This work was funded in part by Conicyt-Mincyt grant CH1204, Partially supported by CeBiB basal funds FB0001, Conicyt, Chile, and STIC-AMSUD RESPOND.

REFERENCES

[1] B. Robbins, A. Nair, and A. Kesari, “S4: Distributed Stream Computing Platform. Leonardo Neumeyer”, in ICDM 2010 pp. 170-177.

[2] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, “Spc: a distributed, scalable platform for data mining”, in DMSS&P, 2006, pp. 27–37.

[3] Storm. [Online]. Available: https://github.com/nathanmarz/storm/wiki, retrieved: March, 2015

[4] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, “ESC: Towards an Elastic Stream Computing Platform for the Cloud”, in CC, 2011, pp. 348–355.

[5] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: fault-tolerant streaming computation at scale”, in SOSP, 2013, pp. 423–438.

[6] V. Gil-Costa, J. Lobos, R. Solar, and M. Marin, “AMEDS-Tool: An Automatic Tool to Model and Simulate Large Scale Systems”, in Summer Simulation Multi-Conference, 2014, pp 20:1--20:8.

[7] H. Andrade, B. Gedik, and D. Turaga, “Fundamentals of Stream Processing Aplications Design, System and Analytics”, Cambridge University Press, 2014.

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: wait-free coordination for internet-scale systems”, in USENIXATC, 2010, pp 11–11.

[9] S. Graham, P. B. Kessler, and M. K. Mckusick, “Gprof: A call graph execution profiler”, in CC, 1982, pp.120-126.

[10] Profiler4j: http://profiler4j.sourceforge.net, retrieved: March, 2015

[11] Java Virtual Machine Monitoring, Troubleshooting, and Profiling Tool. http://docs.oracle.com/javase/7/docs/technotes/guides/visualvm/, retrieved: March, 2015

[12] Java Profiler Tool. http://www.semanticdesigns.com/Products/Profilers/JavaProfiler.html, retrieved: March, 2015

[13] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, “Timestream: reliable stream computation in the cloud”, in EuroSys, 2013, pp. 1–14.

[14] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, “Streamcloud: An elastic and scalable data streaming system”, in Trans. on PDS, vol. 23, no. 12, 2012, pp. 2351–2365.

[15] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, “Integrating scale out and fault tolerance in stream processing using operator state management”, in SIGMOD, 2013, pp. 725–736.

[16] Z. Sebepou and K. Magoutis, “Cec: Continuous eventual checkpointing for data stream processing operators”, in DSN, 2011, pp. 145-156.

[17] D. Laney, “3D Data Management: Controlling Data Volume, Velocity and Variety”, Gartner, 2001, pp. 1-4.

[18] A. De Mauro, M. Greco, and M. Grimaldi. “What is big data? A consensual definition and a review of key research topics”, in AIP, 2015 pp. 97–104.