Scalability and State: A Critical Assessment of Throughput Obtainable on Big Data Streaming Frameworks for Applications With and Without State Information

Shinhyung Yang, Yonguk Jeong, ChangWan Hong, Hyunje Jun, and Bernd Burgstaller(✉)

Department of Computer Science, Yonsei University, Seoul, Korea
{shinhyung.yang,bburg}@yonsei.ac.kr

Abstract. Emerging Big Data streaming applications are facing unbounded (infinite) data sets at a scale of millions of events per second. The information captured in a single event, e.g., GPS position information of mobile phone users, loses value (perishes) over time and requires sub-second latency responses. Conventional Cloud-based batch-processing platforms are inadequate to meet these constraints.

Existing streaming engines exhibit low throughput and are thus equally ill-suited for emerging Big Data streaming applications. To validate this claim, we evaluated the Yahoo streaming benchmark and our own real-time trend detector on three state-of-the-art streaming engines: Apache Storm, Apache Flink and Spark Streaming. We adapted the Kieker dynamic profiling framework to gather accurate profiling information on the throughput and CPU utilization exhibited by the two benchmarks on the Google Compute Engine.

To estimate the performance overhead incurred by current streaming engines, we re-implemented our Java-based trend detector as a multi-threaded, shared-memory application in C++. The achieved throughput of 3.2 million events per second on a stand-alone 2 CPU (44 cores) Intel Xeon E5-2699 v4 server is 44 times higher than the maximum throughput achieved with the Apache Storm version of the trend detector deployed on 30 virtual machines (nodes) in the Cloud. Our experiment suggests vertical scaling as a viable alternative to horizontal scaling, especially if shared state has to be maintained in a streaming application. For reproducibility, we have open-sourced our framework configurations on GitHub [1].

1 Introduction

The increasing demand for Big Data streaming has become prevalent in Cloud-based applications where data streams are characterized by sub-second latency, high density at high velocity, statefulness and near real-time response requirements. Social interactions from existing services such as Twitter and Facebook, real-time click-streams from e-commerce Cloud platforms and GPS position information from mobile applications qualify as such data. Traditionally,

D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 141–152, 2018.
https://doi.org/10.1007/978-3-319-75178-8_12

MapReduce-based batch processing was applied with Big Data streaming applications. In pursuit of programming abstractions tailored specifically for streaming applications, and to support sub-second event response times, the Aurora and Apache Storm Big Data streaming platforms rapidly became popular for businesses and with the academic community.

Today's prominent Big Data streaming engines are programmed in Java, Scala or related programming languages targeting the Java virtual machine (JVM). The hardware abstraction provided by the JVM facilitates deployment in the Cloud. Users are provided with high-level programming primitives to compose streaming applications as a set of nodes (actors) connected by FIFO data channels. The resulting stream-graph topologies can then be readily deployed on the underlying, Cloud-based streaming engine. It is the sole responsibility of the underlying streaming engine to orchestrate a given stream graph topology on a set of Cloud nodes. The programmer is only required to provide high-level configuration parameters such as the number of nodes or virtual machines (VMs).

To assess the efficiency of streaming applications on current state-of-the-art streaming engines, and to determine the cost of the provided programming abstractions, this paper makes the following contributions.

1. We created a Java-based trend detection benchmark for Wikipedia user click-streams. This benchmark was implemented for the Apache Storm and Flink streaming engines. We employed the Yahoo streaming benchmark [10] as our second real-world streaming benchmark.
2. We adopted the Kieker dynamic profiling framework [5] for Spark Streaming and the Apache Storm and Flink streaming engines. To the best of our knowledge, this is the first detailed evaluation of the throughput and CPU utilization of two real-world benchmarks on the before-mentioned streaming engines run on the Google Compute Engine. From our measurements we conclude that CPU resources are under-utilized with current Big Data streaming engines.
3. We re-implemented our Java-based trend detector as a multi-threaded application in C++. Through manual performance optimizations such as the adoption of lock-free data-structures, it was possible to maintain shared state and raise the throughput by a factor of 44x to 3.2 million events per second on a stand-alone shared-memory multicore server. From this result two conclusions can be drawn: (1) the cost of current stream programming abstractions is non-negligible, and (2) vertical scaling on a multi-CPU, multicore computer benefits from the high bandwidth of chip interconnects and can thus be preferable to (pre-mature) horizontal scaling.

The remainder of this paper is structured as follows. In Sect. 2, we present the constituents of the Yahoo benchmark and how the Kieker framework was incorporated to obtain dynamic profiling information. Section 3 introduces our trend detector for the streaming APIs and the low-level C++ version. We present our experimental evaluation in Sect. 4, discuss the related work in Sect. 5 and draw our conclusions in Sect. 6.

2 Yahoo Streaming Benchmark

The purpose of the Yahoo streaming benchmark [10] is to determine the performance of three state-of-the-art Big Data streaming engines: Apache Storm, Apache Flink, and Apache Spark Streaming. The benchmark constitutes a Cloud-deployment of an advertising analytics pipeline. Events arrive through Kafka, the JSON format is deserialized, and events are filtered, projected and joined. Windowed counts of events per campaign are stored in the Redis in-memory database. The Yahoo streaming benchmark consists of three Cloud components: (1) the Kafka distributed data queue, (2) the analytics pipeline expressed for one of the three before-mentioned streaming engines, and (3) the Redis in-memory database.

The Yahoo streaming benchmark as provided on GitHub [10] is configured to run on a single (Cloud) node. It was a non-trivial, time-consuming process to adapt this single-node configuration to multiple nodes. To obtain detailed dynamic profiling information, we incorporated the Kieker dynamic profiling framework as a system daemon on each Cloud node. We developed the system daemon to automatically launch at each boot and it starts to sample per-core CPU utilization every 500 ms. Sampled data is stored locally on each Cloud node and from this raw data we analyze performance of Cloud streaming applications. To make our results reproducible, we have open-sourced these configurations on GitHub [1].

2.1 Kafka Distributed Streaming Queue

Apache Kafka 0.8.2 is deployed as the default data queue with the benchmark. Kafka is a subscription-based distributed streaming queue platform. Data generators written in the Clojure programming language subscribe to the Kafka platform as producers. A streaming application will subscribe to the Kafka platform as a consumer. Kafka works as a Cloud-based global data-queue which hides underlying details and only exposes a few interfaces to producers and consumers. The queue is constructed as a Kafka cluster which consists of one or more Kafka broker servers. In this benchmark, we deploy a Kafka cluster of five Kafka broker servers where each broker server occupies an entire Cloud node.

2.2 Anatomy of Streaming Engines

Streaming engines are the major targets in this streaming benchmark. Because of the high arrival rate of tuples from Kafka at the streaming engine, the throughput of the Yahoo streaming benchmark is solely constrained by the throughput of the streaming engine itself. The streaming engines compared in this experiment are Apache Storm version 0.9.7, Spark Streaming version 1.6.2, and Apache Flink version 1.1.3. With the Storm configuration, Storm Nimbus does bookkeeping and the orchestration of the entire platform. More than one Storm supervisor instance is assigned to a Nimbus instance and runs a subset of the target stream topology. Similarly, the Flink platform is operated by two types of managers, Job Managers and Task Managers. A Job Manager is responsible for allocating subsets of the target stream topology to Task Manager entities. Apache Spark Streaming is an additional layer built upon the Apache Spark platform. This enables stream processing using traditional batch processing of Apache Spark. A Spark cluster is managed by the Spark Master. A Spark Master may have multiple Spark Slaves. Each Slave instance may be assigned to execute a subset of the target stream application.

3 Trend Detector

Trend detection is a popular technique employed with real-world enterprises to discover user trends on Cloud services such as social media, e-commerce and search engines. It is important to note that user-generated data streams have to be analyzed by the Cloud streaming environment. Therefore, trend detection has to be implemented and operated in a Cloud environment using streaming operators.

Monitoring an incoming data stream of user-generated keywords, a trend detection algorithm analyzes the stream to detect irregularities in the occurrences of registered keywords over the most recent consecutive time-windows. Each and every uniquely distinguished keyword is given its own timebucket to store and update a series of occurrences and they are constantly evaluated to list the most trending keyword(s) in the system.

3.1 Java Trend Detector

We implemented a trend detector in Java for the Storm streaming engine. Based on the original approach from Twitter's trend detection [4], our implementation incorporates the point-by-point Poisson model. This Poisson model is employed to explicitly distinguish locally irregular occurrences of a particular keyword within the target time-series, where the overall count of the keyword is insignificant. The point-by-point model is especially applicable to find trending keywords from a small set of data. We improved this model by introducing a parallel reduction algorithm. With our approach, we employ trend-detection actors and aggregator actors as depicted in Fig. 1. They constitute n layers of actors such that 2^n trend-detection actors receive data streams from spout s_0. This layer of trend-detection actors is followed by $n-1$ levels of aggregators. Level k consists of 2^{n-k-1} aggregators, where k indicates the position of the aggregators in the topology.

In this parallel reduction, each trend-detection actor evaluates the trendiness of incoming keywords with its own set of timebuckets of unique keywords. That is, for a single stream of keywords that are equally distributed into 2^n trend-detection actors, there will be 2^n parallel evaluations of trendiness on local sets of timebuckets. This strategy was necessary in two aspects: first, in a Big Data streaming application, there is no guarantee on how many "sibling" instances of an actor exist in the stream graph topology. No exchange of information is

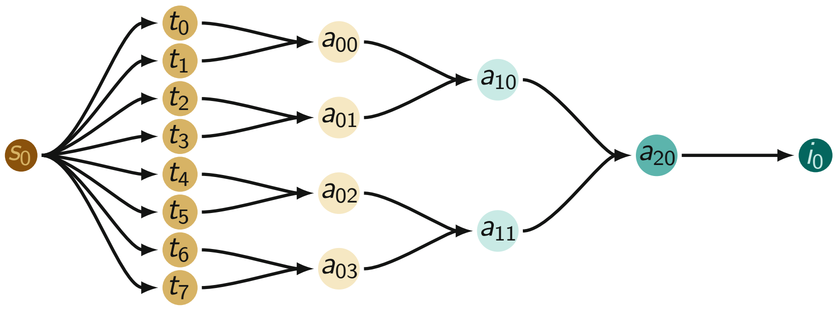


Fig. 1. Our Java-based trend-detector's topology of 3 layers. The topology is dynamically created at the beginning of the run-time with given number of layers for speculative parallel reduction.

allowed between actors except for the FIFO data channels connecting producers to consumers. Thus in such stream graph topologies it is not possible to share a single global set of timebuckets across all actors (actors cannot have shared state). Therefore each trend-detection actor is designed to keep its own set of timebuckets and evaluate it separately. Secondly, by parallel reduction, multiple actors can jointly conduct the evaluation.

The trend detection layer is followed by layers of aggregator actors. An aggregator actor accepts a tuple which contains a keyword and its trendiness from a pair of preceding trend-detection actors or aggregator actors. An aggregator determines a keyword of the highest trendiness from its local list. With the Java trend detector, the last layer consists of a single aggregator actor. The resulting keyword from this actor is considered as the most trending keyword.

In our design of the Java trend detector, actors have local state only. Thus trend detection is based on local decisions and hence semantically incorrect and speculative. (I.e., the parallel trend detector may not always compute exactly the same trends as the underlying sequential solution. Nevertheless, differences materialize only under certain adversary cases of event-distributions, which are outside of the scope of this paper.)

3.2 C++ Trend Detector

Our goal for the C++ version of the trend detector is to fully utilize the underlying hardware of a single multicore node, while focusing on creating a semantically correct, non-speculative trend detector. In conducting an evaluation of the C++ trend detector, we assumed that all keyword tuples arrive in the right order between preceding and succeeding tuples in terms of creation time. That is because, if a tuple arrives too early or too late, it won't be placed in the right time window on which trend detection is performed.

As depicted in Fig. 2, one datagenerator is employed per CPU. The program targets a server with two Xeon E5-2699 v4 CPUs, where one CPU consists of

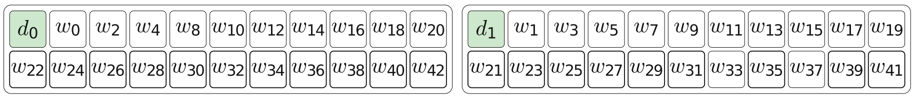

d_0 w_0 w_2 w_4 w_8 w_{10} w_{12} w_{14} w_{16} w_{18} w_{20}
w_{22} w_{24} w_{26} w_{28} w_{30} w_{32} w_{34} w_{36} w_{38} w_{40} w_{42}
d_1 w_1 w_3 w_5 w_7 w_9 w_{11} w_{13} w_{15} w_{17} w_{19}
w_{21} w_{23} w_{25} w_{27} w_{29} w_{31} w_{33} w_{35} w_{37} w_{39} w_{41}

Fig. 2. Thread-to-core allocation of the C++ trend detector on a server with two Xeon E5-2699 v4 CPUs. One datagenerator $d_{[01]}$ is assigned per CPU; the remaining cores are filled with worker threads w_*.

22 cores. Our design of the C++ trend detector leverages information about the hardware architecture. To prevent the OS from moving threads between cores, we pinned one datagenerator thread onto the first core of each CPU and the other cores are pinned with worker threads. (Pinning was done with the LIKWID-pin utility [8], which manipulates the CPU affinity of a program's threads.) At the worker thread creation stage, allocation of multiple worker threads will take turn between the two CPUs. The worker threads pinned onto the same CPU receive tuples in a round-robin fashion from the datagenerator inhabiting the same CPU through a dedicated queue.

To maximize throughput, we utilize B-Queues [9] as the system-level streaming queues between a datagenerator and its workers. A B-Queue is a lock-free single-producer, single-consumer queue, and we use one dedicated B-Queue from a datagenerator to each of its connected workers. We unrolled the innermost loop of the C++ datagenerator such that tuples are entered into each queue once per iteration.

Once a worker receives a keyword, it looks up the corresponding, dedicated timebucket for this keyword in a global hashmap. Then the keyword's timestamp of its creation time is inserted into the timebucket and the keyword's trendiness is evaluated periodically. The evaluated trendiness is then inserted into the global trending list and the most trending keyword at the current time is determined from the list. The global hashmap is a highly-contended shared data-structure causing serialization from lock contention. We overcame this problem by introducing a global lock-free hashtable [7].

3.3 An Example Data Set

As an example data set which qualifies as Big Data, we chose a snapshot of Wikipedia's traffic from the Amazon web services public data set page [2]. This data set contains 150 GB of hourly page traffic statistics collected from January 1, 2011 to March 31, 2011. This data set was employed for benchmarking both the Java and the C++ trend detector.

4 Experimental Results

In this paper, we ran three benchmarks. One is our C++ trend detector which was evaluated on a CentOS 7 server machine consisting of two Xeon E5-2699 v4 CPUs with 512 GB of RAM. The other two are Big Data streaming applications—the

Yahoo streaming benchmark and our Java trend detector (the counterpart of the C++ trend detector). To evaluate them correctly we referred to Yahoo's Cloud setup [3]. First, we configured 30 hypervised machines on the Google Compute Engine. In this setup, one hypervised machine has 16 virtual CPUs (vCPUs) with 24 GB of RAM. Each vCPU is a hyperthreaded core of an Intel Xeon processor running at 2.50 GHz. Nineteen hypervised machines are dedicated to "infrastructure" purposes: three Zookeeper nodes, one Redis in-memory database instance, five Kafka brokers constituting one Kafka cluster, and 10 Kafka producer nodes which feed tuple streams into the Kafka cluster. Eleven hypervised machines are dedicated to the actual application running on the streaming engine under evaluation. One coordinator is needed to manage an entire streaming engine and its workers. Thus we are left with 10 workers which run the streaming application itself.

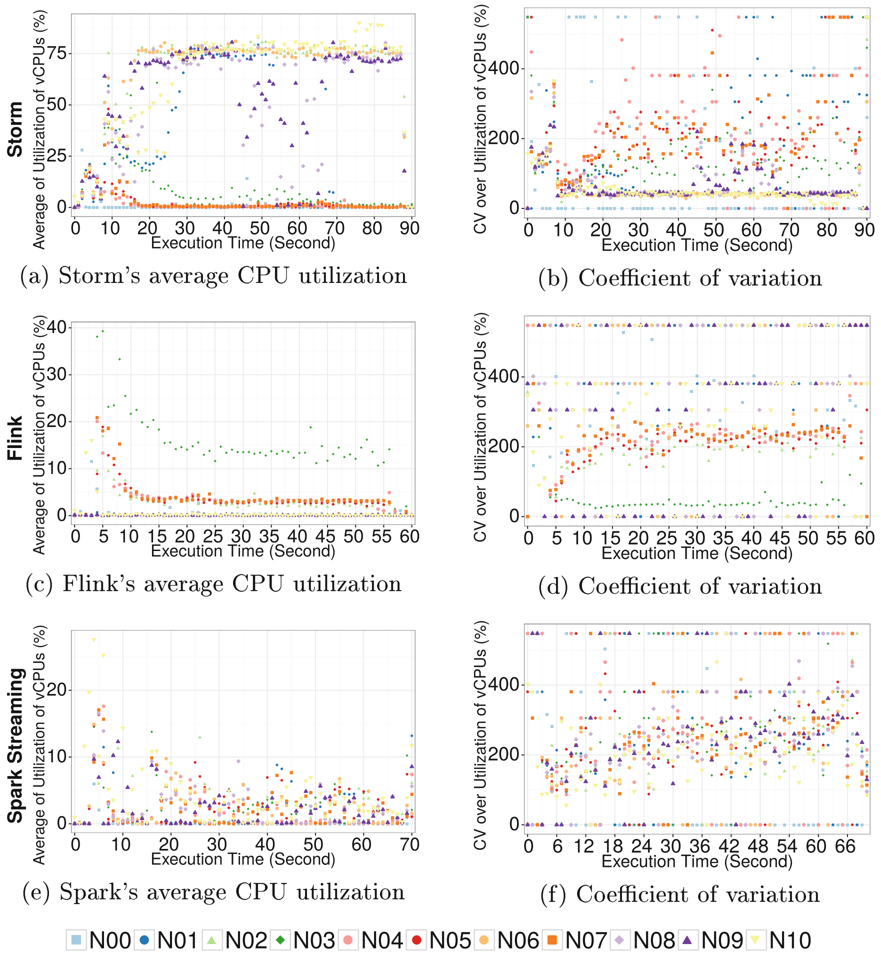
We measured the CPU utilization of our Big Data streaming benchmarks using the Kieker dynamic profiling framework. We configured the Kieker framework's periodic sampler facility to measure per-core CPU utilization of 11 streaming engine nodes every 500 ms during the execution of the benchmark to sample at the double frequency of per-second sampling rate according to the Nyquist-Shannon sampling theorem. Measurements from the 16 vCPUs of a single hypervised machine node are averaged to denote per-node and per-second CPU utilization during the period of the benchmark execution. From this refinement we produced two graphs: Average (AVG) and Coefficient of Variation (CV) graphs. Each AVG graph shows per-node CPU utilization of hypervised machines which run the streaming engine. Each CV graph shows the degree of sparseness among per-second utilization of each hypervised machine.

To determine the efficiency of each streaming engine's orchestration of worker nodes, we generated a diagram where the actor allocation across the 10 worker nodes is depicted. Each hypervised machine is depicted in a unique color and all actor instances are included, to provide the complete picture of how a streaming engine orchestrated actor instances across hypervised machine nodes. Due to substantial differences with the programming interface, we did not produce actor orchestration diagrams for the Spark streaming engine.

4.1 Yahoo Streaming Benchmark

Big Data streaming engines require complicated Cloud configurations in which multiple hypervised machines collaborate to run different types of software components which have dependencies to other components. In this benchmark, the infrastructure consists of Zookeepers, Redis database nodes, the Kafka cluster, Kafka producers, and streaming engines. Thirty hypervised machines are required to execute a streaming application.

In Fig. 3c and e, CPUs are under-utilized. Most of the worker nodes are only utilizing 10% or less of their CPU resources. Although Flink has one outlier node depicted in green, its top CPU utilization is only 40%. On the other hand, Fig. 3a shows higher CPU utilization. Five worker nodes are utilizing more than 75% of their CPU resource. It is more clearly depicted in Fig. 4a. With the Storm

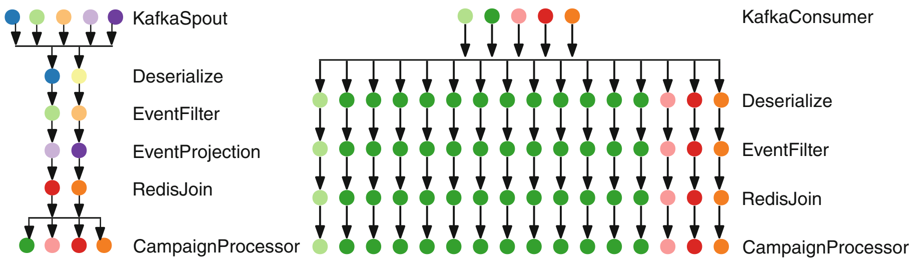


(a) Storm's average CPU utilization (b) Coefficient of variation

(c) Flink's average CPU utilization (d) Coefficient of variation

(e) Spark's average CPU utilization (f) Coefficient of variation

Fig. 3. CPU utilization of Big Data streaming engines with the Yahoo streaming benchmark



(a) Storm's actor allocation (b) Flink's actor allocation

Fig. 4. Stream graph topology and actor distribution for the experiment from Fig. 3 (employing the same color code)

configuration, all participating worker nodes are allocated with actor instances of the target topology. This is different from Flink's actor distribution diagram in Fig. 4b. Flink actors are only partly distributed on five worker nodes. Compared to Storm, this orchestration does not seem efficient. However, Flink in fact achieved higher throughput than Storm. The throughput of the Yahoo streaming benchmark with Flink is 282 141 tuples per second whereas Storm only achieved 24 703 tuples per second. In terms of throughput, Flink's orchestration works better, however the problem still remains that it did not utilize five worker nodes at all. In the evaluation of Cloud applications, this is clearly inefficient because energy resources to run worker nodes are wasted.

4.2 Trend Detector

In comparing the Java trend detector to the C++ trend detector, two key factors are to be considered: First, the Java trend detector runs on a Big Data streaming platform, which means there will be multiple worker nodes participating in an execution of the application. Second, by necessity of the streaming engine programming abstractions, the actors of the Java trend detector are restricted to local state only. In particular, each trend-detection actor has its own actor-local set of timebuckets, and the aggregator actors do not share global state. Local state reduces the communication overhead compared to a shared, global (distributed) store of timebuckets. Nevertheless—as pointed out before—this performance advantage comes at the cost of analysis precision.

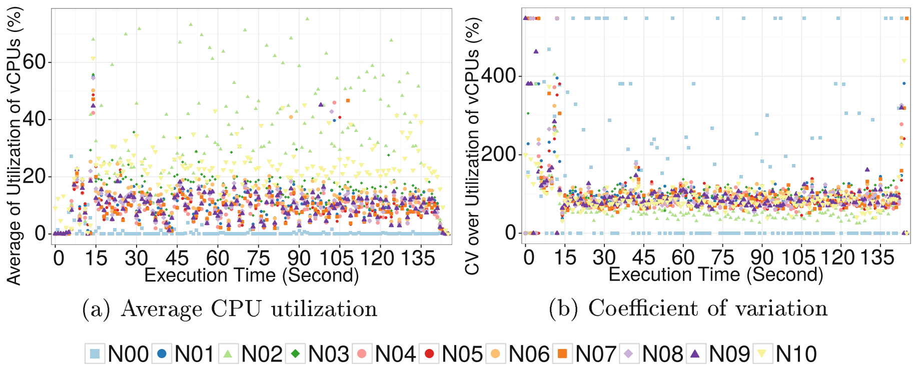


Fig. 5. CPU utilization characteristics of the Java-based trend detector

To benchmark the Java trend detector, we adopted the Cloud setup that we employed with the Yahoo streaming benchmark. Although Flink's throughput was higher than Storm's in our experiment with the Yahoo streaming benchmark, we chose Storm due to its prevalence in industry. Therefore Storm's programming interface was used to implement our Java trend detector. Because our implementation is based on Storm APIs, we ran the benchmark on Storm only.

In the benchmark result of Fig. 5, under-utilization of CPUs is also shown with the Java trend detector's average utilization (see Fig. 5a). Except for a few nodes that reach just under 80%, most nodes stay below 20% average CPU utilization. In Fig. 5b, abundant sparseness of CPU utilization is depicted. Only two nodes are utilized more, and those are at 80%. In the CV diagram, one node, i.e., "▲", performs well in terms of consistently utilizing CPU resources. The Java trend detector's highest throughput is 72 499 tuples per second.

The C++ trend detector was evaluated on a stand-alone Intel Xeon E5-2699 v4 system. The throughput of our shared-memory stateful C++ trend detector implementation is 3 217 432 tuples per second. It can be evaluated in two ways: first, even though we chose to use a global timebucket hashtable, because the hashtable is designed lock-free, it avoids cache coherence overhead from locking. This way, we achieved correct semantics with state information shared by all worker threads. Second, we showed that a single C++ trend detector on a single machine can obtain higher throughput than its Java-based Cloud counterpart.

To determine the maximum possible throughput in terms of tuples emitted by the datagenerators, we removed all worker threads except one consumer per queue which had the sole purpose of emptying its queue. This configuration achieved 309 360 800 tuples per second.

The shared-memory trend detector shows that it is possible to reduce energy consumption and increase the performance over current Cloud streaming engines. However, the C++ programming interface is not as easy as developing streaming applications with Cloud streaming frameworks. Big Data stream programming interfaces are easier for beginners than C++ programming. Developing efficient C++ multi-threaded applications requires careful, manual hand-tuning and optimizations to detect and remove performance bottlenecks.

In conclusion, our experiment shows that the raw performance of sending tuples across processes is two orders of magnitude higher than the performance of a carefully hand-tuned, multi-threaded C++ streaming application. And the performance of this C++ streaming application is 44x times higher than what can be achieved with current state-of-the-art of Cloud streaming frameworks.

In terms of programming effort, the C++ version required the 3-week attention of a multicore programming expert, whereas the Java version of our trend detector was created by a group of software capstone students new to stream programming—in about the same amount of time.

5 Related Work

In [3], Chintapalli et al. introduce the Yahoo streaming benchmark and its purpose to measure the latency for a complete processing of a tuple at different Kafka emission rates. Although three streaming engines, namely Apache Spark streaming, Apache Storm and Apache Flink were compared in the paper, because of architecture and language differences, Spark streaming was evaluated differently with regard to micro-batching intervals. Although the micro-batching interval does affect the result, Spark streaming was the slowest with Flink being as the 2nd-slowest of the three. It is important to note that our measurement of throughput is different from what its authors intended to measure with the same framework. Chintapalli et al. measured the up-to-date latency at each stage of tuple process completion. Contrary, in our benchmark, we measured the throughput of tuples at the source-node of a streaming application.

In [4], the principles of trend detection are explained. Three popular models are explained with the point-by-point Poisson model being the simplest but the most effective for a small set of time series. With large-enough sets of time series, the authors recommend the cycle-corrected Poisson model, which will increase the precision of the algorithm. Lastly, a data-driven method is introduced for its stableness and adaptability. In our Java trend detector and the C++ trend detector, we adapted the point-by-point Poisson model.

In [6], McSherry et al. propose a new paradigm in evaluating the performance of distributed data processing systems (aka Big Data processing systems). The authors take examples of parallelized algorithms that scale well compared to other algorithms, while in fact, the performance of the compared algorithm is better. They point out the importance of better (highly-optimized) baselines. If the baseline single-threaded algorithm is of low performance, a parallelized algorithm will inevitably perform better than this baseline, even if it is only parallelizing the overhead contained in the baseline. They suggest to improve the baseline with better algorithms. The paper's idea aligns well with our introduction of the C++ trend detector. Our Java trend detector and other streaming applications that scale well within Big Data platforms should be re-evaluated in terms of energy-efficiency and throughput, because we have shown that our stateful C++ trend detector showed the highest throughput over the other benchmarks, although it runs on a single machine.

6 Conclusions

We have shown that existing Big Data streaming platforms exhibit low throughput and inefficient utilization of the underlying Cloud infrastructure. Measurement data was obtained for the Yahoo streaming benchmark and our real-time trend detectors with the help of the Kieker dynamic profiling framework. Our stateful C++ trend detector uses vertical scaling on a shared-memory multicore server. It outperformed its Cloud-based counterparts by 44 times higher throughput. For reproducibility, we have open-sourced our streaming framework configurations on GitHub [1].

Acknowledgements. Research supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Science, ICT & Future Planning under grant NRF2015M3C4A7065522.

References

1. Yang, S.: Cloud framework configurations for the Yahoo and the real-time trend-detector benchmarks. https://github.com/shinhyungyang/cloud-ready. Created 13 Feb 2017
2. Wikipedia page traffic statistic v3. https://aws.amazon.com/datasets/wikipedia-page-traffic-statistic-v3/. Accessed 13 Feb 2017
3. Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B.J., Poulos, P.: Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, pp. 1789–1792, May 2016
4. Hendrickson, S., Kolb, J., Lehman, B., Montague, J.: Trend detection in social data. https://github.com/jeffakolb/Gnip-Trend-Detection/raw/master/paper/trends.pdf. Accessed 13 Feb 2017
5. van Hoorn, A., Waller, J., Hasselbring, W.: Kieker: a framework for application performance monitoring and dynamic software analysis. In: Proceedings of 3rd ACM/SPEC International Conference on Performance Engineering, ICPE 2012, pp. 247–248. ACM, New York (2012)
6. McSherry, F., Isard, M., Murray, D.G.: Scalability! But at what cost? In: Proceedings of 15th USENIX Conference on Hot Topics in Operating Systems, p. 14, May 2015
7. Preshing, J.: The world's simplest lock-free hash table. http://preshing.com/20130605/the-worlds-simplest-lock-free-hash-table/. Accessed 13 Feb 2017
8. Treibig, J., Hager, G., Wellein, G.: LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of First International Workshop on Parallel Software Tools and Tool Infrastructures, PSTI 2010, San Diego, CA (2010)
9. Wang, J., Zhang, K., Tang, X., Hua, B.: B-queue: efficient and practical queuing for fast core-to-core communication. Int. J. Parallel Prog. **41**(1), 137–159 (2013)
10. Yahoo Inc.: Yahoo streaming benchmarks GitHub page. https://github.com/yahoo/streaming-benchmarks. Accessed 13 Feb 2017