Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors

Shuhao Zhang[1,2], Bingsheng He[2], Daniel Dahlmeier[1], Chi Zhou[3], Thomas Heinze[1]
[1]SAP, [2]National University of Singapore, [3]INRIA

***Abstract*—Driven by the rapidly increasing demand for handling real-time data streams, many data stream processing (DSP) systems have been proposed. Regardless of the different architectures of those DSP systems, they are mostly aimed at scaling out using a cluster of commodity machines and built around a number of key designs: a) pipelined processing with message passing, b) on-demand data parallelism, and c) JVM based implementation. However, hardware has evolved dramatically. More CPU cores are being put on the same die, and the on-chip cache hierarchies are getting larger, deeper, and more complex. Moreover, we have observed the non-uniform memory access (NUMA) effect. In this paper, we revisit the aforementioned designs on a modern scale-up server. Specifically, we use a series of applications to profile Storm and Flink. From the profiling results, two major performance issues are observed: a) the massively parallel threading model causes serious front-end stalls, which are a major performance bottleneck issue on a single CPU socket, b) the lack of NUMA-aware mechanism causes major drawback on the scalability of DSP systems on multi-socket architectures. Addressing these issues should allow DSP systems to exploit modern scale-up architectures, which also befits scaling out environments. We present our initial efforts on resolving the above-mentioned performance issues, which have shown up to 3.2x and 3.1x improvement on the performance of Storm and Flink, respectively.**

I. Introduction

Many data stream processing (DSP) systems have recently been proposed to meet the increasing demand of processing streaming data, such as Apache Storm [1], Flink [2], Spark Streaming [3], Samza [4] and S4 [5]. Regardless of the different architectures of those DSP systems, they are mainly designed and optimized for scaling out using a cluster of commodity machines (e.g., [6], [7], [8]). We observe the following three common designs in building those existing DSP systems:

a) *pipelined processing with message passing:* A stream application is usually implemented as multiple operators with data dependency, and each operator performs three basic tasks continuously, i.e., receive, process and output. Such a pipelined processing design makes DSP systems able to support very low latency processing, which is one of the key requirement in many real applications that cannot be well supported in batch-processing systems.

b) *on-demand data parallelism:* Fine-grained data parallelism configuration is supported. Specifically, users can configure the number of threads in each operator (or function) independently in the stream application. Such on-demand data parallelism design aims at helping DSP systems scale for high throughput.

c) *JVM based implementation:* DSP systems are mostly built on top of JVM. The use of JVM-based programming language makes the system developing more productive (e.g., built-in memory management) and have many potential contributors from the open-source community.

In computer architecture, there is a trend of increasing the number of cores on a processor die. Subsequently, the on-chip cache hierarchies that support these cores are getting larger, deeper, and more complex. Furthermore, as modern machines scale to multiple sockets, non-uniform memory access (NUMA) becomes an important optimization factor for the performance of multi-socket systems (e.g., [9], [10]). For example, recent NUMA systems have already supported hundreds of CPU cores and multi-terabytes of memory [11]. However, there is a lack of detailed studies on profiling the relationship between those affecting factors and the performance of DSP systems on modern architectures.

In this paper, we experimentally revisit those common designs based on Storm [1], Flink [2] on a modern machine with multiple CPU sockets. We aim to offer a better understanding of how designs on modern DSP systems interact with modern processors when running different types of workloads. We use two DSP systems (i.e., Storm and Flink) as the evaluation targets (Section II-A). As those designs are commonly adopted, we expect similar results on most DSP systems. There has been no standard benchmark for stream applications, especially on the scale-up environment. We design our streaming benchmark with seven stream applications according to the four criteria proposed by Jim Gray [12] (Section III-C). Note that, the major goal of this study is to evaluate the common designs of DSP systems on scale-up architectures using profiled results so that our results can be applicable to many other DSP systems, rather than to compare the absolute performance of individual systems (as in the previous studies [13]).

We make the following key observations.

First, the design of supporting both pipelined and data parallel processing resulting in a very complex massively parallel threading model in DSP systems, which poorly utilizes modern multi-core processors. Based on our profiling results, a significant portion ($\sim$40%) of the total execution time is wasted due to L1-instruction cache (L1-ICache) misses. The significant L1-ICache misses are mainly due to the frequent thread context switching during execution and large instruction footprint of each execution thread.

Second, the design of continuous message passing between

operators, causes a serious performance degradation to the DSP systems running on multiple CPU sockets. Furthermore, the current design of data parallelism tends to equally partition input streams, which causes workload unbalance due to the NUMA effect. The throughput of both Storm and Flink on four CPU sockets is only slightly higher or even lower than that on a single socket for all applications in our benchmark. The costly memory accesses across sockets (up to 24%) severely limit the scalability of DSP systems.

Third, the JVM runtime brings two folds of overhead to the execution, and they are moderate. 1) the frequent pointer referencing during tuple transmission and the frequent accesses to virtual method tables of JVM runtime (especially in Storm) stress Translation Lookaside Buffer (TLB). The resulting TLB stalls take 5∼10% and 3∼8% of the total execution time for most applications on Storm and Flink, respectively. 2) the overhead from garbage collection (GC) accounts for only $1 \sim 3\%$ of the total execution time. The observed minor impact of GC is very different from previous studies on other platforms (e.g., [14], [15]).

Addressing the above-mentioned issues should allow DSP systems to exploit modern scale-up architectures. As initial attempts, we evaluate two optimizations: 1) *non-blocking tuple batching* to reduce the thread context switches so that the instruction cache performance can be improved without explicitly buffering delay; 2) *NUMA-aware executor placement* to make thread placement aware of remote memory access. The evaluation results show that both optimizations are effective in improving the performance of DSP systems on multi-socket multi-core processors. Putting them altogether achieves 1.3∼3.2x and 1.3∼3.1x throughput improvement on Storm and Flink, respectively.

To the best of our knowledge, this is the first detailed study of common designs of DSP systems on scale-up architectures with a wide range of applications. Improving DSP systems on the scale-up architectures is also beneficial for the scale-out environment, by either offering a better performance with the same number of machines or reducing the number of machines to achieve the same performance requirement.

The remainder of this paper is organized as follows. Section II introduces preliminary and background of this study. Section III presents the experimental setup and design, followed by the evaluation and profiling in Section IV and Section V. Section VI describes our preliminary efforts in addressing the performance issues of DSP systems. We review the related work in Section VII and conclude in Section VIII.

II. Preliminaries and Background

In this section, we first introduce three designs of two DSP systems studied in this paper, namely Storm [1] and Flink [2]. Then, we introduce the background of the multi-socket multi-core processors.

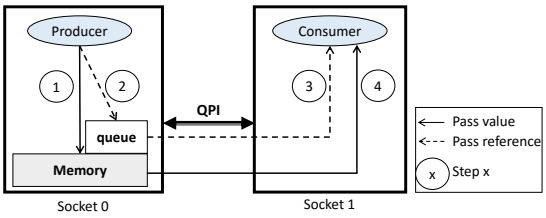


Fig. 1: Message passing mechanism.

A. Data Stream Processing Systems

In the following, we introduce three designs of Storm and Flink with a focus on their similarities and differences: 1) pipelined processing with message passing, 2) on-demand data parallelism, and 3) JVM based implementation.

Pipelined processing with message passing. A commonly adopt computation model for stream computation is graph processing [16]. Specifically, a streaming application is represented by a graph, where nodes in the graph represent either data source operators or data processing operators, and edges represent the data flow between operators. In general, there are two types of operators defined in the topology. (i) Data Source operator generates (or receive from the external environment) events to feed into the topology, and (ii) Data Processor operator encapsulates specific processing logics such as filtering, transforming, correlating or user-defined function.

In a shared-memory environment, an operator (continuously) writes its intermediate results into cache/memory, and push a tuple with reference (i.e., pointer) of the results into its output queue. The corresponding consumer (continuously) fetches the reference of the tuple from the queue, and then process on the actual contents through memory fetch. In other words, the actual contents of output results are not being passed but kept locally, and only the reference (i.e., pointer) to it is being passed. This pass-by-reference message passing approach avoids duplicating data in a shared-memory environment and is the common approach adopted by most modern DSP systems. Figure 1 illustrates an example of message passing between operators in a shared-memory environment, where the producer and consumer are scheduled to CPU socket 0 and socket 1 respectively. The producer first writes its output data to the local memory of socket 0 (step 1) and emits a tuple containing a reference to the output data to its output queue (step 2). The consumer fetches from the corresponding queue to obtain the tuple (step 3) and then accesses the data by the reference (step 4). This example also demonstrates one case of remote memory access during message passing in DSP systems, which we will study in detail in Section V-C.

On-demand data parallelism. Modern DSP systems such as Storm and Flink are designed to support task pipeline and data parallelism at the same time. The actual execution of an operator is carried out by one or more physical threads, which are referred as executors. The inputs of an operator are (continuously) partitioned among its executors.

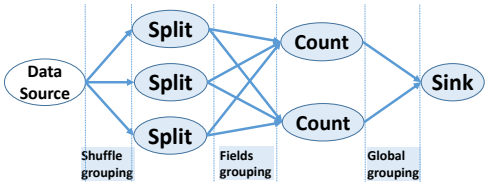


Fig. 2: Word-count execution graph.

The number of executors for a certain operator is referred to as the parallelism level and can be *configured* by users in the topology configuration. A topology at the executor level is called an execution graph. An example execution graph of the word-count application is shown in Figure 2. In this example, split, count, and sink operators have three, two and one executors, respectively. Streams are partitioned and delivered to specific destination executor according to the grouping strategy specified by the user. In the previous example, the Data Source makes sure that an equal number of tuples are delivered to each split executor as shuffle grouping is specified. Meanwhile, each spilt executor sends tuples to count executors according to the attribute of the tuple (specified as field grouping) so that the same word (i.e., with the same key) is always delivered to the same count executor.

JVM based implementation. Both Storm and Flink are implemented with JVM-based programming languages (i.e., Closure, Java, and Scala), and their execution relies on JVM. Three aspects of JVM runtime are discussed as follows.

Data reference: As we have mentioned before, message passing in DSP systems always involves passing the reference. That is, operators access the data through the reference in the tuple. However, accessing data through one or more references may lead to pointer chasing, which stresses the Translation Lookaside Buffer (TLB) of the processor heavily [17].

Method table: To support the runtime polymorphism of programs, when the JVM loads a non-abstract class, it generates a method table (i.e., *vtable*) to store the direct references to all instance methods that could be invoked for an instance of the non-abstract class. Upon a class loading, *invokevirtual* instruction is triggered to search the method table and identify the specific method implementation. Such an operation stresses TLB, where one execution can cause one TLB miss in the worst case.

Garbage collection (GC): Another important aspect of the JVM based system is the built-in memory management. Modern JVM implements generational garbage collection which uses separate memory regions for different ages of objects. The significant overhead of GC has been reported in many existing studies on non-streaming data analytics platforms (e.g., [15], [14]). To this end, some DSP systems have even implemented its own memory management besides JVM (e.g., Flink). However, as we will show later, the overhead of GC is negligible in running all applications in our benchmark on both Storm and Flink.

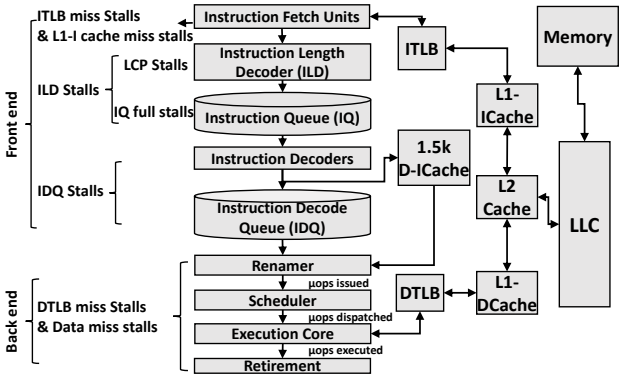


Fig. 3: Pipeline execution components of processor.

B. Multi-Socket Multi-core Processors

Modern processors consist of multiple different hardware components with deep execution pipelines, as shown in Figure 3. Pipelines interact with cache and memory systems and are illustrated in the middle. Varies stalls caused in the pipeline are illustrated on the left, and the interactions with the cache, TLB, and memory systems are illustrated on the right. The pipeline can be divided into the front-end component and the back-end component [18].

The front-end is responsible for fetching instructions and decodes them into micro-operations (μops). It feeds the next pipeline stages with a continuous stream of micro-ops from the path that the program will most likely execute, with the help of the branch prediction unit. Starting from Sandy Bridge micro-architecture, Intel introduces a special component called Decoded ICache (D-ICache), which is essentially an accelerator of the traditional front-end pipeline. D-ICache maintains up to $1.5k$ of μops that are decoded by decoders. Future references to the same μops can be served by it without performing the fetch and decode stages. D-ICache is continuously enhanced regarding size and throughput in the successor generations of Intel processors. Note that, every μops stored in D-ICache is associated with its corresponding instruction in L1-ICache. An L1-ICache miss also causes D-ICache to be invalidated.

The back-end is where the actual instruction execution happens. It detects the dependency chains between the decoded μops (from IDQ or the D-ICache), and executes them in an out-of-order manner while maintaining the correct data flow.

As modern machines scale to multiple sockets, NUMA brings more performance issues. Figure 4 illustrates the NUMA topology of our tested machine with *four* sockets. Each CPU socket has its local memory, which is uniformly shared by *eight* CPU cores on the socket. Sockets are connected by a much slower (compared to local memory access) channel called Quick Path Interface (QPI).

III. Methodology

We conduct an extensive set of experiments to profile the performance of Storm and Flink on a modern scale-up server using different applications. In this section, we first present the

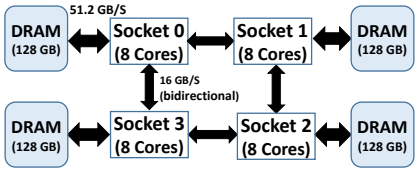


Fig. 4: NUMA topology and theoretical bandwidth.

TABLE I: JVM profile tools

Flags		Description
JIT Logging		Trace just-in-time compilation activities
	UnlockDiagnosticVMOptions	Enable processing of flags relating to field diagnostics
	TraceClassLoading	Trace all classes loaded
	LogCompilation	Enable log compilation activity
	PrintAssembly	Print assembly code
GC Logging		Trace garbage collection activities
	PrintGCTimeStamps	Print timestamps of garbage collection
	PrintGCDetails	Print more details of GC including size of collected objects, time of objects promotion

evaluation goals of this study. Next, we introduce our profiling tools, followed by our benchmark.

A. Evaluation Goals

This study has the following design goals. Firstly, we aim to identify the common designs of modern DSP systems, and to understand how those designs (i.e., pipelined processing with message passing, on-demand data parallelism, and JVM based implementation) interact with modern processors when running different types of workloads. Secondly, with the detailed profiling study, we hope to identify some hardware and software approaches to resolving the bottleneck and point out the directions for the design and implementation of future DSP systems.

B. Profiling Tools

JVM profile. Table I lists the JVM flags that we use to monitor the performance of JVM. We are mainly interested in two kinds of activities, including those in just-in-time (JIT) compilation and GC. We only enable those trace logs when we need to analyze the corresponding activities. Otherwise, the trace logs are disabled. We use Performance Inspector [19] for gathering detailed instruction-tracing information. We measure the size of the objects created at runtime using the *MemoryUtil* tool from the Classmexer library [20].

Processor profile. We systematically catalog where the processor time is spent for executing Storm and Flink to identify common bottlenecks of their system designs when running on multi-socket multi-core processors. We use Intel Vtune [21] for profiling at the processor level.

Similar to the recent study [15], we break down the total execution time to the following components: 1) computation time, which is contributed by the issued μops that subsequently be executed and retired; 2) branch misprediction stall time (T_{Br}), which is mainly due to the executed μops that will however never be retired; 3) front-end stall time (T_{Fe}), which is due to the μops that were not issued because of the stalls in any components in the front-end; 4) back-end stall time (T_{Be}), which is due to the μops that were available in the IDQ but were not issued because of resources being held-up in the back-end.

TABLE II: Processor measurement components

Variable		Description
T_C		Effective computation time
T_{Br}		Branch misprediction stall time
T_{Fe}		Front-end stall time
	ITLB stalls	Stall time due to ITLB misses that causes STLB hit or further cause page walk
	L1-I cache stalls	Stall time due to L1 instruction cache misses
	ILD stalls	Instruction Length Decoder stalls
	IDQ stalls	Instruction Decoder Unit stalls
T_{Be}		Back-end stall time
	DTLB stalls	Stall time due to DTLB misses, which causes STLB hit or further cause page walk
	L1-D Stalls	Stall time due to L1 data cache misses that hit L2 cache
	L2 Stalls	Stall time due to L2 misses that hit in LLC
	LLC miss (local)	Stall time due to LLC misses that hit in local memory
	LLC miss (remote)	Stall time due to LLC misses that hit in memory of other socket

TABLE III: Detailed specification on our testing environment

Component		Detail
Processor		Intel Xeon E5-4640, Sandy Bridge EP
	Cores	8 * 2.4 GHz
	Sockets	4
	L1 Cache	32 KB Instruction, 32 KB Data per core
	L2 Cache	256 KB per core
	L3 Cache	20 MB per socket
Memory		4 * 128 GB, Quard DDR3 channels, 800 MHz
Apache Flink		version 1.0.2 (checkpoint enabled)
Apache Storm		version 1.0.0 (acknowledge enabled)
Java HotSpot VM		java 1.8.0_77, 64-Bit Server VM, (mixed mode) -server -XX:+UseG1GC -XX:+UseNUMA

Table II shows the measurement components for individual stalls. We have conducted an extensive measurement on stalls from front-end, back-end, and branch misprediction.

All our experiments are carried out on a four-sockets server with the Intel Xeon Sandy Bridge EP-8 processors. Table III shows the detailed specification of our server and relevant settings in Storm and Flink.

C. Streaming Benchmark

We design our streaming benchmark according to the four criteria proposed by Jim Gray [12]. As a start, we design the benchmark consisting of seven stream applications including Stateful Word Count (WC), Fraud Detection (FD), Spike Detection (SD), Traffic Monitoring (TM), Log Processing (LG), Spam Detection in VoIP (VS), and Linear Road (LR).

We briefly describe how they achieve the four criteria. 1) Relevance: the applications cover a wide range of memory and computational behaviors, as well as different application complexities so that they can capture the DSP systems on scale-up architectures; 2) Portability: we describe the high-level functionality of each application, and they shall be easily ported to other DSP systems; 3) Scalability: the benchmark includes different data sizes; 4) Simplicity: we choose the

applications with simplicity in mind so that the benchmark is understandable.

Our benchmark covers different aspects of application features. *First*, our applications cover different runtime characteristics. Specifically, TM has highest CPU resource demand, followed by LR, VS, and LG. CPU resource demand of FD and SD is low. TM has the highest memory requirement compared to the other applications. *Second*, topologies of the applications have various structural complexities. Specifically, WC, FD, SD, and TM have single chain topologies, while LG, VS, and LR have complex topologies. Figure 5 shows the topology of the seven applications.

In the following, we describe each application including its (a) meaning, (b) implementation details and (c) input setup.

Stateful Word Count (WC): (a) The stateful word-count counts and remembers the frequency of each received word unless the application is killed. (b) The topology of WC is a single chain composed of a Split operator and a Count operator. The Split operator parses sentences into words and the Count operator reports the number of occurrences for each word by maintaining a hashmap. This hashmap is once created in the initialization phase, updated for each receiving word, and never discarded. (c) The input data of WC is a stream of string texts generated according to a Zipf-Mandelbrot distribution (skew set to 0) with a vocabulary based on the dictionary of Linux kernel (3.13.0-32-generic).

Fraud Detection (FD): (a) Fraud detection is a particular use case for a type of problems known as outliers detection. Given a transaction sequence of a customer, there is a probability associated with each path of state transition, which indicates the chances of fraudulent activities. We use *missProbability* as the detection algorithm [22] with sequence window size of 2. Once the value is lower than the threshold, an alert is triggered. (b) The topology of FD has only one operator, named as Predict, which is used to maintain and update the state transition of each customer. (c) We use a sample transaction with 18.5 million records for testing. Each record includes customer ID, transaction ID, and transaction type.

Log Processing (LG): (a) Log processing represents the stream application of performing real-time analyzing on system logs. (b) The topology of LG consists of four operators. The Geo-Finder operator finds out the country and city where an IP request is from, and the Geo-Status operator maintains and updates all the countries and cities that have been found so far. The Status-Counter operator performs statistics calculations on the status codes of HTTP logs. The Volume-Counter operator counts the number of log events per minute. (c) We use a subset of the web request data (with 4 million events) from the 1998 World Cup Web site [23]. Due to the privacy policy, the actual IPs of the requests are hidden, and we randomly reassign those IPs to each record.

Spike Detection (SD): (a) Spike detection tracks measurements from a set of sensor devices and performs moving aggregation calculations. (b) The topology of SD has two operators. The Moving-Average operator calculates the average of input data within a moving distance. The Spike-Detection operator checks the average values and triggers an alert whenever the value has exceeded a threshold. (c) We use the Intel lab data (with 2 million tuples) [24] for this application. The detection threshold of moving average values is set to 0.03.

Spam Detection in VoIP (VS): (a) Similar to fraud detection, spam detection is a use case of outlier detection. (b) The topology of VS is composed of a set of filters and modules that are used to detect telemarketing spam in Call Detail Records (CDRs). It operates on the fly on incoming call events (CDRs), keeps track of the past activity implicitly through a number of on-demand time-decaying Bloom filters. A detailed description of its implementation can be found at [25]. (c) We use a synthetic data set with 10 million records for this application. Each record contains data on a calling number (String), called number (String), calling date (Date time), answer time (double), call duration (double), and call established (boolean).

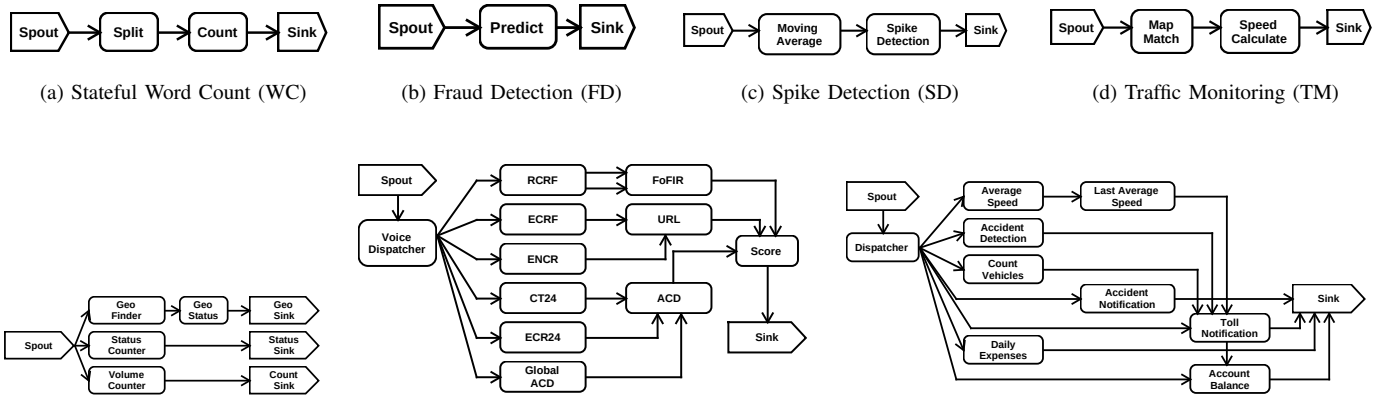
Traffic Monitoring (TM): (a) Traffic monitoring performs real-time road traffic condition analysis, with real-time mass GPS data collected from taxis and buses[1]. (b) TM contains a Map-Match operator which receives traces of an object (e.g., GPS loggers and GPS-phones) including altitude, latitude, and longitude, to determine the location (regarding a road ID) of this object in real-time. The Speed-Calculate operator uses the road ID result generated by Map-Match to update the average speed record of the corresponding road. (c) We use a subset (with 75K events) of GeoLife GPS Trajectories [26] for this application.

Linear Road (LR): (a) Linear Road (LR) is used for measuring how well a DSP system can meet real-time query response requirements in processing a large volume of streaming and historical data [27]. It models a road toll network, in which tolls depend on the time of the day and level of congestions. Linear Road has been used by many DSP systems, e.g., Aurora [28], Borealis [29], and System S [30]. LR produces reports of the account balance, assessed tolls on a given expressway on a given day, or estimates cost for a journey on an expressway. (b) We have followed the implementation of the previous study [31] for LR. Several queries specified in LR are implemented as operators and integrated into a single topology. (c) The input to LR is a continuous stream of position reports and historical query requests. We merge the two data sets obtained from [32] resulting in 30.2 million input records (sorted by time attribute) and 28.3 million history records.

IV. Performance Evaluation

In this section, we present the performance evaluation results of different applications on Storm and Flink on multi-core processors. We tune each application on both Storm and

[1] A real deployment at http://210.75.252.140:8080/infoL/sslk_weibo.html.



(a) Stateful Word Count (WC) (b) Fraud Detection (FD) (c) Spike Detection (SD) (d) Traffic Monitoring (TM)

(e) Log Processing (LG) (f) Spam Detection in VoIP (VS) (g) Linear Road (LR)

Fig. 5: Topologies of seven applications in our benchmark.

TABLE IV: Resource utilization on single CPU socket

	WC	FD	LG	SD	VS	TM	LR
CPU Utilization (Storm)	62%	39%	61%	28%	75%	98%	71%
Memory Utilization (Storm)	20%	16%	10%	7%	19%	60%	31%
CPU Utilization (Flink)	75%	27%	31%	13%	92%	97%	78%
Memory Utilization (Flink)	53%	16%	18%	6%	17%	52%	20%

Flink according to their specifications such as the number of threads in each operator.

Throughput and resource utilization on a single socket. Figure 6a shows the throughput and Table IV illustrates the CPU and memory utilizations of running different applications on Storm and Flink on a single CPU socket, respectively. We made two observations. First, the comparison between Storm and Flink is inconclusive. Flink has higher throughput than Storm on WC, FD, and SD, while Storm outperforms Flink on VS and LR. The two systems have similar throughput on TM and LG. Second, our benchmark covers different runtime characteristics. Specifically, VS and TM have high CPU resource demand. CPU resource demand of LG and LR is median, and that of WC and SD is low. TM has the highest memory requirement compared to the other applications.

It is noteworthy that the major goal of this study is to identify the issues in common designs of DSP systems on scale-up architectures, rather than to compare the absolute performance of different DSP systems. We present the normalized performance results in the rest of this paper.

Scalability on varying number of CPU cores. We vary the number of CPU cores from 1 to 8 on the same CPU socket and then vary the number of sockets from 2 to 4 (the number of CPU cores from 16 to 32). Figure 6b and 6c show the normalized throughput of running different applications with varying number of cores/sockets on Storm and Flink, respectively. The performance results are normalized to their throughputs on a single core.

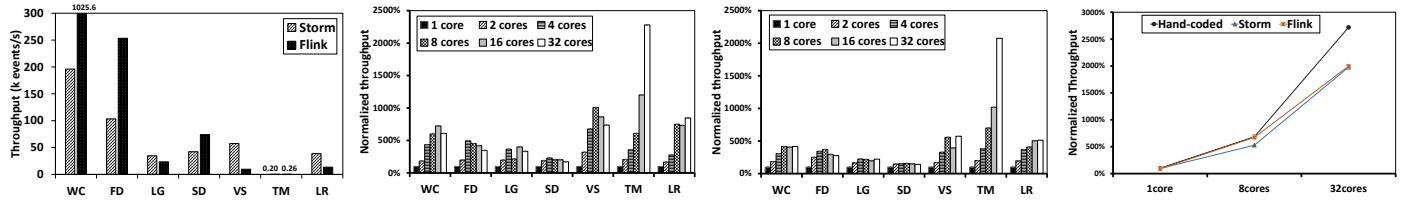
We have the following observations. *F*irstly, on a single socket, most of the applications scale well with the increasing number of CPU cores for both Storm and Flink. In contrast, both Storm and Flink are not able to scale well on multi-sockets. *S*econdly, most applications perform not better or even worse on multiple sockets than on a single socket. FD, SD become even worse on multiple sockets than on a single socket, this is due to their relatively low compute resource demand. Enabling multiple sockets only brings additional overhead of remote memory access (RMA). WC, LG, VS perform similarly in both cases as the trade-off between increased resource and overhead of RMA is about balanced. The throughput of LR increases marginally with the increasing number of sockets. *T*hirdly, TM has a significantly higher throughput in both systems on four sockets than on a single socket. This is because TM has high resource demand on both CPU and memory bandwidth. To better understand the potential improvement on optimizing DSP system for such "heavy" application, we manually write a Java program to simulate the running of TM application without Storm or Flink. Results are normalized by the throughput of hand-coded program on one core as shown in Figure 6d. Compared to hand-coded program, Both Storm and Flink have comparable throughput (i.e., 90% and 98% of hand-coded) of TM on a single socket. However, the throughput achieved on four sockets (i.e., 72% and 73% of hand-coded) of Storm and Flink still leave space for further improvement.

V. Study the impact of common designs

In the following section, we investigate the underlying reasons for the performance degradation and how the three designs interact with multi-socket multi-core processors. Specifically, we first show an execution time breakdown on running different applications on Storm and Flink on a single socket. Then, we study the impact of massively parallel threading model, the impact of message passing and stream partition and the impact of JVM runtime environment.

A. Execution time breakdown

Finding (1): During execution of most applications (except TM) on both Storm and Flink, $\sim$ 70% of their time are spent in processor stalls, which leads low resource utilization.



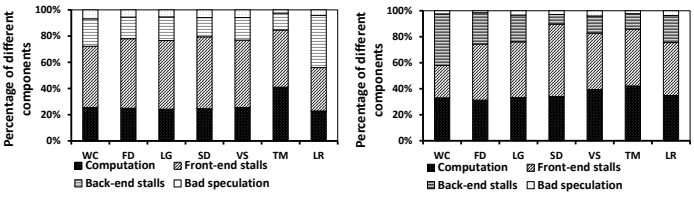
(a) Evaluation of seven applications on a single socket.

(b) Storm with varying number of cores/sockets.

(c) Flink with varying number of cores/sockets.

(d) TM execute on hand-coded program, Storm or Flink.

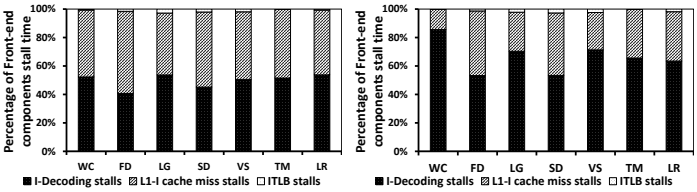
Fig. 6: Performance evaluation results on Storm and Flink.



(a) Storm

(b) Flink

Fig. 7: Execution time breakdown.



(a) Storm

(b) Flink

Fig. 8: Front-end stall breakdown.

Figure 7 shows the execution time breakdown of Storm and Flink on different processor components as introduced in Section II-B. We find that 59∼77% and 58∼69% of the overall execution time are spent on stalls (Branch misprediction stalls, Front-end stalls, Back-end stalls) for all applications running on Storm and Flink, respectively.

Front-end stalls account for 35∼55% and 25∼56% of the total execution time of Storm and Flink, respectively. This result is significantly different from the batch processing framework (e.g., [14]). Back-end stalls account for approximately 13∼40% and 7∼40% of the total execution time of Storm and Flink, respectively. Branch misprediction stalls are low, ranging from $3 \sim 4\%$ for all applications.

B. Massively parallel threading model

Finding (2): The design of supporting both pipelined and data parallel processing results in a very complex massively parallel threading model in DSP systems. Our investigation reveals that the high front-end stalls are mainly caused by this threading model.

Figure 8 illustrates the breakdown of the front-end stalls in running Storm and Flink on a single socket. The L1 instruction cache (L1-ICache) miss stalls and instruction decoding (I-Decoding) stalls each contributes nearly half of the front-end stalls.

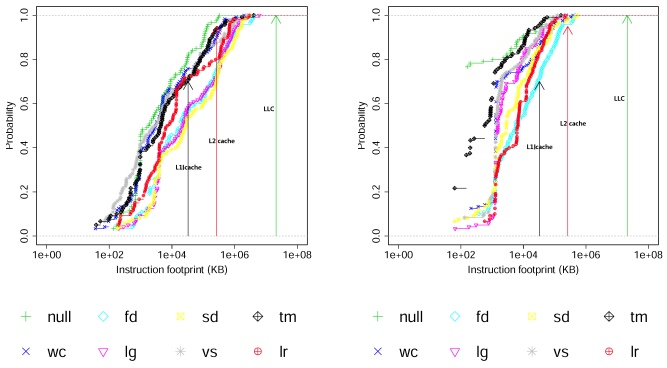
L1-ICache miss stalls: Our investigation reveals that there are two primary sources responsible for the high L1-ICache miss. *First*, due to the lack of a proper thread scheduling mechanism, the massive threading execution runtime of both Storm and Flink produces frequent thread context switching. *Second*, each thread has a large instruction footprint. By logging JIT compilation activities, we found that the average size of the native machine code generated per executor thread goes up to 20 KB. As the size of current L1-ICache (32 KB per core) is still fairly limited, it cannot hold those instructions on the fly, which eventually leads to L1-ICache thrashing.

We now study the details of the instruction footprints between two consecutive invocations of the same function. In order to isolate the impact of user-defined functions, we test a "Null" application, which performs nothing in both Storm and Flink (labeled as "null" in the figure). Figure 9 illustrates the cumulative density function (CDF) of instruction footprints on a log base 10 scale.

We add three solid vertical lines to indicate the size of L1-ICache (32KB), L2 cache (256KB), and LLC (20MB). With the detailed analysis on the instruction footprint, we make three observations. *First*, two turning points on the CDF curves are observed at $x = 1KB$ and $x = 10MB$ for Storm and $x = 1KB$ and $x = 1MB$ for Flink, which reflects the common range of their instruction footprint during execution. *Second*, the cross-over points of L1-ICache line and different CDF curves are low and between 0.3 $\sim$ 0.5 for Storm and 0.6 $\sim$ 0.8 for Flink. Flink has a better instruction locality than Storm on L1-ICache. *Third*, Storm has similar tracing on instruction footprint with or without running user applications. This indicates that many of the instruction cache misses may come from Storm platform itself. This also explains the reason that different applications show similar L1-ICache miss in Storm. In contrast, the platform of Flink has a lower instruction footprint.

I-Decoding stalls: The high instruction decoding (I-Decoding) stalls are related to the high L1-ICache miss issue. The I-Decoding stalls can be further breakdown into Instruction Length Decoder (ILD) stalls and Instruction Decoding Queue (IDQ) stalls.

The ILD stalls further consist of instruction queue (IQ) full stalls and length change prefix (LCP) stalls. IQ is used to store the next instructions in a separate buffer while the processor is executing the current instruction. As mentioned



(a) Storm (b) Flink

Fig. 9: Instruction footprint between two consecutive invocations of the same function.

before, instruction footprint of executors in both Storm and Flink is large. Due to the frequent context switching, many instructions are frequently prefetched and stored in IQ, which may cause a stall if IQ is full. IQ full stalls are frequent and contribute nearly 20% of front-end component stall time for all applications on Storm and Flink respectively. On the other hand, the LCP stalls account for less than 0.05% for all applications according to our measurement.

Another important aspect of I-Decoding stalls is the IDQ stalls, which consists mainly of Decoded Instruction Cache (D-ICache) stalls. D-ICache enables skipping the fetch and decode stages if the same μops are referenced later. However, two aspects of D-ICache may offset its benefits, or even degrade the performance. *First*, when L1-ICache miss occurs, the D-ICache also needs to be updated, which subsequently causes a switch penalty (i.e., the back-end has to re-fetch instructions from the legacy decoder queue). *Second*, if a hot region of code is too large to fit in the D-ICache (up to 1.5k μops), the front-end incurs a penalty when μop issues switch from the D-ICache to the legacy instruction decode queue. As we have shown earlier that L1-ICache misses are high during Storm and Flink execution, this issue propagates to a later stage, which causes frequent missed in the D-ICache and eventually causes high IDQ stalls.

C. Message passing and stream partition

Finding (3): The design of message passing between operators causes a severe performance degradation to the DSP systems running on multiple CPU sockets. During execution, operators may be scheduled into different sockets and experience frequent costly remote memory access during the fetching of input data. Furthermore, the current design of data parallelism creates serious unbalance in stream partition due to the NUMA effect.

Recently, the NUMA-aware allocator has already been implemented in the Java HotSpot Virtual Machine to take advantage of such infrastructures and provides automatic memory placement optimizations for Java applications. We

TABLE V: LLC miss stalls when running Storm with four CPU sockets.

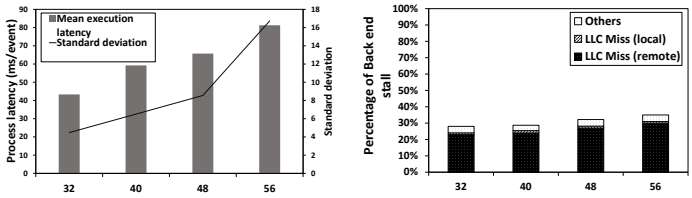
	WC	FD	LG	SD	VS	TM	LR
LLC Miss (local)	0%	5%	3%	4%	4%	1%	7%
LLC miss (remote)	6%	16%	17%	13%	17%	24%	22%

enable this optimization in our JVM by specifying the *useNUMA* flag. However, our experiments have already shown that this flag is insufficient for reducing the NUMA impact and have observed poor scalability in both Storm and Flink on multiple sockets. The main problem is the high remote memory access overhead due to the heavy pipelined message flow design. During execution, each executor needs to fetch data from the corresponding producer continuously. When those actions occur within the same machine (i.e., shared-memory environment), they are essentially cache/memory fetch operations. Due to NUMA, an executor can have three kinds of data accesses.

1) In cache: the input data is accessed in the cache. This comes with minimum access penalty. However, this hardly happens during tuple transmission between different executors.
2) In local memory: access data with a miss in the cache but a hit in its local memory. This happens when producer and consumer executors are located in the same CPU socket, and it comes with a cost of local memory read.
3) In remote memory: access data with a miss in the cache and a further miss in local memory. This comes with the largest access penalty, and it happens when producer and consumer executors are located in different CPU sockets.

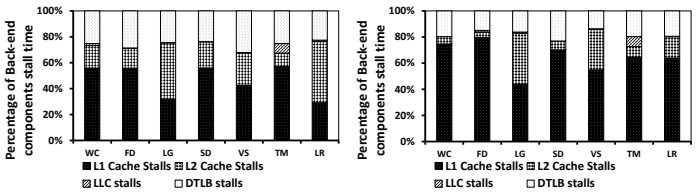
As a result, the data access cost is *producer-location dependent*, which creates significant performance divergence among parallel executors of even the same logical operator. However, neither Storm nor Flink is aware of such performance heterogeneity issues and continuously distributes equal amounts (in the case of shuffle grouping) of tuples among executors, resulting in a poor workload balance. Table V shows the LLC miss stalls for other applications executing on Storm with four CPU sockets. We have similar observations when running the applications on Flink with four CPU sockets enabled

We take TM on Storm as an example to further study the impact of stream partition. We start with the tuned number of threads (i.e., 32) of Map-Matcher operator of TM on four sockets, and further increase the number of threads up to 56. Figure 10a shows a significant increase in the standard derivation of executors' latencies with the increasing of the number of executors when running Storm on four CPU sockets. Those executors experience up to 3 times difference in the average execution latency in the case of 56 Map-Matcher executors, which reaffirms our analysis on performance heterogeneity in NUMA. Further, due to the significant overhead caused by remote memory access, the mean execution latency also increase along with the growing number of executors. Figure 10b shows that the back-end stalls become worse with the increase in the number of executors.



(a) Average execution time per event. (b) Back-end stalls.

Fig. 10: Varing number of executors of Map-Matcher operator when running Storm with four CPU sockets.



(a) Storm (b) Flink

Fig. 11: Back-end stall breakdown.

This indicates the remote memory access penalty prevents DSP systems from scaling well. The results indicate that the LLC miss (remote) stalls are high in all applications. .

D. JVM Runtime Environment

Finding (4): The overhead of JVM runtime contains two folds and are moderate. First, the frequent pointer referencing during tuple transmission and the frequent method table search stress TLB (stalls account for $5 \sim 10\%$ and $3 \sim 8\%$). Second, the overhead of GC in running stream applications ($1 \sim 3\%$) is insignificant.

Both Storm and Flink are implemented using JVM-based programming language. The efficiency of JVM runtime is crucial to the performance of Storm and Flink. As we have mentioned before, the back-end of the processor is where the actual execution happens. Figure 11 break down the back-end stalls into L1-DCache stalls, L2 cache stalls, LLC stalls, and DTLB stalls when running Storm and Flink on a single socket.

Data cache stalls: Stalls in L1-D and L2 cache dominate the back-end stalls in both systems. The intermediate results (content of each output tuple) generated in all stream applications in our benchmark are relatively *small footprint* and *short-lived* (i.e., not persistently stored). As a result, data access misses in L1-DCache are usually get served by L2 cache (L1-DCache stall) and LLC (L2 cache stalls). Further, as we have shown previously that the instruction footprints during Storm and Flink executions are mostly smaller than L2 cache, instructions missed in L1-ICache are also mostly hit in L2 cache.

DTLB stalls: Tuples are passed with reference (instead of the actual data) in both systems. Executors access the data through the reference in the receiving tuple. As a result, the frequent pointer references lead to stress on DTLB on both systems. Storm experiences even higher DTLB stalls than Flink. By investigating the source code of Storm, we find that runtime polymorphism is aggressively used in Storm (especially the Clojure part). As a result, the frequent load and search on method table further stress TLB.

Garbage collection overhead: We use G1GC [33] as the garbage collector in our JVM. The garbage collection (GC) is infrequent in running all applications on both Storm and Flink, and the same observation is made even if we run the benchmark for hours by continuously refeeding the input. Based on GC logs, we found that no major GC occur during the execution and minor GC contributes only $1 \sim 3\%$ to the total execution time across all the applications for both Storm and Flink. As a sanity check, we also study the impact of using parallelGC. When the parallelGC mechanism is used instead, the overhead of GC increases slightly to around $10 \sim 15\%$.

VI. Towards more efficient DSP systems

In this section, we present our initial attempt at addressing the performance issues found in the previous section. We present two optimization techniques, including non-blocking tuple batching (to reduce instruction cache misses) and NUMA-aware executor placement (to reduce remote memory accesses). We evaluate the effectiveness of the techniques by first studying their individual impacts and then combining both techniques together.

A. Non-blocking Tuple Batching

Our profiling results suggest that the frequent context switching causes severely performance issues including L1-ICache miss and I-Decoding stalls, which lead to high front-end stalls. One of the solutions is batching multiple output tuples together before passing down. In this way, each executor can process multiple tuples in receiving each batch, which leads to less frequent context switching. Similar ideas of tuple batching are already proposed [31] or in use in some DSP systems [2]. However, those techniques rely on a buffering stage, which introduces implicit/explicit wait delay in execution. For example, Sax et al. [31] proposed to create an independent batching buffer for each consumer in order to batch all tuples that will be processed by the same. Tuples are not emitted until the corresponding buffer becomes full. However, the buffering stage may cause an unpredictable delay in execution, which subsequently leads to random tuple failure and replay. In order to preserve low latency processing feature of DSP system, we develop a simple yet effective *non-blocking tuple batching* strategy to address this issue.

The basic idea of our solution is as follows. Consider an executor output *multiple* tuples for each input, we try to put its output tuples together as a batch, or multiple batches each contains tuples belonging to the same key. Once the corresponding consumer receives such a batch in one thread invocation, it can then process multiple tuples from the batch and further batching its output tuples similarly. Our solution requires the Data Producer to prepare the initial batches of tuples, where the size of batch S is a parameter that we will tune in later experiments. When the Data Producer of

an application generates multiple data (more than S) each time, it simply groups them into batches with size up to S and feeds to the topology. Otherwise, we can let the Data Producer accumulate S tuples before feeding to the topology. As Data Producer is usually relatively light-weight compared to other Data Processors in an application, this does not bring additional overhead.

It is rather straightforward to implement such non-blocking tuple batching for any grouping policy except key-grouping policy (i.e., fields grouping), as we can simply group together all the output tuples of an executor in those cases. However, if an executor uses fields grouping, simply putting output tuples into one batch may cause errors [31] due to wrongly sending output tuples targeting at *different* consumers based on the key in each tuple. Existing batching techniques rely on a buffering stage in order to resolve such issue [31]. In contrast, we develop an algorithm for non-blocking tuple batching of fields grouping, as illustrated in Algorithm 1.

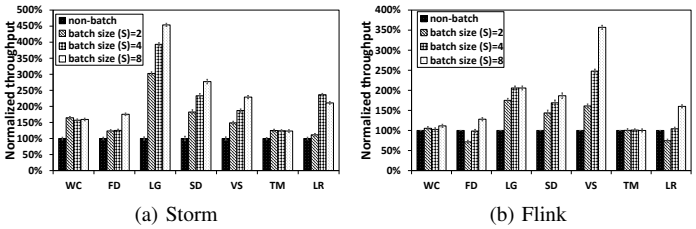
The basic idea is to store multiple output tuples into a multi-value hash map (at lines 10-12), and the fields (i.e., keys) used in choosing consumer are re-computed based on the fields originally declared (at lines 10-11). At line 4, the HashMultimap is the multi-value hash map used to batch multiple values with the same key (implemented based on *org.apache.storm.guava.collect.HashMultimap*). At line 10, we use a concatenate function to combine the original multiple fields. In this way, we guarantee the correctness by always generating the same new key from the same original fields while batching as many tuples as possible (i.e., it may generate the same new key for tuples with different original fields which are can be safely batched together).

Algorithm 1 Non-blocking tuple batching for fields grouping

```
Input: batch-tuple T_IN
 1: T_o: temporary output tuple;
 2: T_o.attributeList: fields grouping attributes;
 3: N: the number of executors of the consumer;
 4: Initialize cache as an empty HashMultimap;
 5: Initialize newkey as an empty object;
 6: for each tuple T_i of T_IN do
 7:     /*Perform custom function of the operator.*/
 8:     T_o ← function_process (T_i);
 9:     /*Combine the values of fields grouping attributes of T_o into temp.*/
10:     temp ← Combine(T_o.attributeList);
11:     newkey ← (hash value of temp) mod N;
12:     Store the < newkey, T_o > pair in cache;
13: end for
14: for each key K_i of the key sets of cache do
15:     Get the < K_i, L > pair from cache;
16:     /*Emit multiple tuples as a List by the emit API of Storm or Fiink.*/
17:     emit(< K_i, L >);
18: end for
```

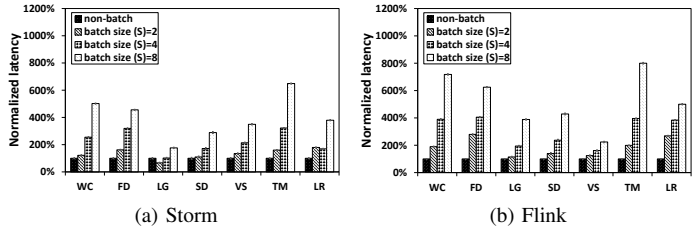
We now study the impact of tuple batching optimization by varying S. Figure 12 illustrates the normalized throughput of Storm and Flink with tuple batching optimization for different applications on a single CPU socket. Results are normalized to the original non-batch setting of Storm and Flink (denoted as non-batch). Tuple batching can significantly reduce instruction cache misses and hence improve the performance of most applications.

With tuple batching, the processing latency of each tuple may be increased as they are not emitted until all tuples in the same batch are processed. In the worst case, the processing latency increases linearly with the size of the batch. Figure 13 shows the normalized average latency per tuple under different batch sizes. Comparing Figure 12 and 13, we observe a clear trade-off between the throughput and latency. Meanwhile, our non-blocking tuple batching scheme preserves a sublinear increasing in process latency for most applications, which is due to the much-improved performance.



(a) Storm (b) Flink

Fig. 12: Normalized throughput of tuple batching optimization.



(a) Storm (b) Flink

Fig. 13: Normalized latency of tuple batching optimization.

B. NUMA-Aware Executor Placement

In order to reduce the remote memory accesses among sockets, the executors in a topology should be placed in an NUMA-aware manner. To this end, we develop a simple yet effective NUMA-aware executor placement approach.

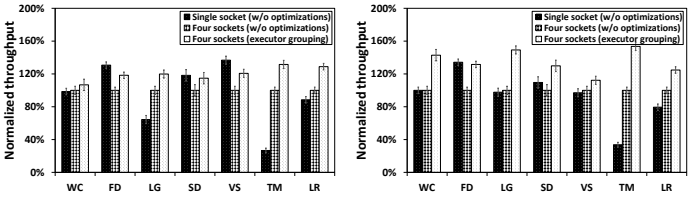
Definition 1: ***Executor placement.*** Given a topology execution graph T and the set of executors W in T, an executor placement $P(T,k)$ represents a plan of placing W onto k CPU sockets. k can be any integer less than or equal to the total number of sockets in a NUMA machine.

Definition 2: We denote the remote memory access penalty per unit as R, and the total size of tuples transmitted between any two executors v and w as $Trans(v,w)$. Here, $v,w \in W$. Each $P(T,k)$ has an associated **cross-socket communication cost**, denoted by Cost($P(T,k)$) as shown in Equation 1. We denote the set of executors placed onto socket i as C_i, where $i = 1,\ldots,k$.

$$Cost(P(T,k)) = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{v \in C_i}^{w \in C_j} R * Trans(v,w) \qquad (1)$$

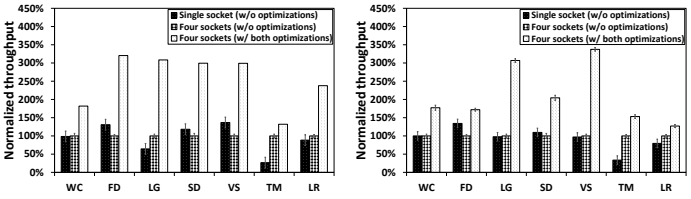
Definition 3: The **optimal executor placement**, denoted by P_{opt}, is defined as $Cost(P_{opt}(T,k)) \leq Cost(P(T,k))$, $\forall P \in Q$, where Q is the set of all executor placement solutions.

Our optimization problem is to find $P_{opt}(T,k)$ for a given topology T and a number of enabled sockets k. In our experiment, we consider k from one to four on the four-socket



(a) Storm (b) Flink

Fig. 14: NUMA-aware executor placement.



(a) Storm (b) Flink

Fig. 15: Normalized throughput with all optimizations enabled.

server. We now illustrate that this problem can be mapped into the minimum k-cut problem [34].

Definition 4: The **minimum k-cut** on weighted graph $G = (V,E)$ produces a vertex placement plan (C_{opt}) such that V is partitioned into k non-empty disjoint sets, and the total weight of edges across disjoint sets is minimized.

Given a topology execution graph T, we can map it to a directed weighted graph $G = (V,E)$. A mapping from T to G is defined as follows: **(I)** $\forall$ executor $w_i \in T$, there is a one-to-one mapping from w_i to a vertex v_i in G. **(II)** For any producer-consumer ($< w_i, w_j >$) message passing relationships in T, there is a one-to-one mapping to one edge $e(v_i, v_j)$ in G. The communication cost ($R * Trans(w_i, w_j)$) is assigned as the edge weight. The cross-socket communication cost corresponds to the total weight of all edges crossing the disjoint sets. Thus, optimizing C_{opt} is equivalent to optimizing P_{opt}.

We use the state-of-the-art polynomial algorithm [34] for solving this problem by fixing k to one to the number of sockets in the machine. Then, from the results optimized for different k values, we test and select the plan with the best performance.

Figure 14 shows the effectiveness of the NUMA-aware executor placement. Results are normalized to four sockets without optimization. The placement strategy improves the throughput of all applications by 7∼32% and 7∼31% for Storm and Flink, respectively.

C. Put It All Together

Finally, we put both optimizations, namely non-blocking tuple batching ($S = 8$) and NUMA-aware executor placement together. Figure 15 illustrates the optimization effectiveness on a single socket and four sockets. Results are normalized to four sockets without optimization. With four sockets, our optimizations can achieve 32∼220% and 27∼212% improvement on throughput for Storm and Flink, respectively. Although our initial attempts have significantly improved the performance, there is still a large room to linear scale-up.

VII. Related Work

Performance evaluation for DSP systems: Stream or event processing has attracted a great deal of research effort. A number of systems have been developed, for example, TelegraphCQ [35], Borealis [29], Yahoo S4 [5], IBM System S [30] and the more recent ones including Storm [1], Flink [2] and Spark Streaming [3]. However, little attention has been paid to the key and common designs of DSP systems on modern multi-core processors. Through profiling studies on those designs in Storm and Flink, we are able to identify the common bottlenecks of those system designs when running on multi-socket multi-core processors. There are a few existing studies on comparing different DSP systems. A recent study [36] comparing Flink, Storm, and Spark Streaming has shown that, Storm and Flink have sub-second latency with relatively low throughputs, while Spark streaming has higher throughputs at a relatively high latency. A comparison of S4 and Storm [37] uses a micro-benchmark to understand the performance issues of the systems regarding scalability, execution time and fault tolerance. A similar study [38] has been conducted to compare the performance characteristics of three DSP systems, including System S, S4, and Esper. However, those studies are still aimed at scaling out setting. In contrast, our study does not focus on the problem of which system runs faster. Rather, we focus on the common system designs in DSP systems and conduct detailed profiling on those design of DSP systems on scale-up architectures. Our findings could be able to generalize to DSP systems other than Storm and Flink, and even future DSP systems.

Optimization for streaming systems: Several recent efforts have been efforts have been made with the aim of optimizing DSP systems. Implementing a batch layer in Storm without affecting the semantic of user applications is difficult due to the possibility of causing wrong fields grouping. T-Storm [6] assigns/re-assigns tasks according to run-time statistics in order to minimize inter-node and inter-process traffic while ensuring no worker nodes is overloaded. However, based on our study, task/executor placement inside a single machine also need to be considered, in order to minimize the intra-process traffic.

Database optimizations on scale-up architectures: Scale-up architectures have brought many research challenges and opportunities for in-memory data management, as outlined in recent surveys [39], [40]. There have been studies on optimizing the instruction cache performance [41], [42], the memory and cache performance [43], [44], [45], [46] and NUMA [9], [10], [47]. In the following, we concentrate on two kinds of studies that are closely related to our optimization techniques. Related to the tuple batching optimization, StagedDB [48] reveals that the interference caused by context-switching during the execution results in high penalties due

to additional conflict and compulsory cache misses. They hence introduced a staged design for DBMS, which breaks down DBMS into smaller self-contained modules. Pipeline processing design of DSP systems also has a similar serious problem due to its much more flexible execution model, i.e., each query contains multiple independent running operators. Based on our profiling, thread management should also be considered in order to reduce the impact of frequent instruction cache misses. Query deployment on multi-core machine [47] proposed recently is related to our study of NUMA-aware executor placement. A similar idea to query operator placement is applied in order to minimize the impact of NUMA.

VIII. Conclusions

This paper revisit three common designs of modern DSP systems including a) pipelined processing with message passing, b) on-demand data parallelism, and c) JVM based implementation on modern multi-socket multi-core processors. Our results show that those designs underutilized the scale-up architectures in these two key aspects: a) The design of supporting both pipelined and data parallel processing resulting in a very complex massively parallel threading model in DSP systems, which causes high front-end stalls on a single CPU socket; b) The design of continuous message passing between operators and the current data parallelism approach severely limit the scalability of DSP systems on multi-socket multi-core processors. We further present two optimizations to address those performance issues and demonstrate promising performance improvements. Through this profiling study, we have identified many research opportunities. One example is that the schedulers of Storm and Flink are both designed for scale-out environments. There is a lack of the good mechanism to schedule components of topologies inside scale-up architectures.

References

[1] Apache storm, url: http://storm.apache.org/.

[2] Apache flink, url: https://flink.apache.org/.

[3] M. Zaharia and et al, "Discretized streams: Fault-tolerant streaming computation at scale," in *SOSP '13*.

[4] Apache samza, url: http://samza.apache.org/.

[5] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *ICDMW'10*.

[6] J. Xu and et al, "T-storm: Traffic-aware online scheduling in storm," in *ICDCS '14 IEEE*, June.

[7] L. Aniello and et al, "Adaptive online scheduling in storm," in *DEBS '13*.

[8] B. Peng and et al, "R-storm: Resource-aware scheduling in storm," in *Middleware '15*, 2015.

[9] V. Leis and et al, "Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age," in *SIGMOD '14*.

[10] Y. Li and et al, "Numa-aware algorithms: the case of data shuffling," in *CIDR '13*.

[11] Sgi uvtm 300h system specifications, url:https://www.sgi.com/pdfs/4559.pdf.

[12] J. Gray, *Benchmark Handbook: For Database and Transaction Processing Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[13] G. Hesse and M. Lorenz, "Conceptual survey on data stream processing systems," in *ICPADS'15*.

[14] A. J. Awan and et al, "Performance characterization of in-memory data analytics on a modern cloud server," in *BDCloud '15*.

[15] S. Sridharan and J. M. Patel, "Profiling r on a contemporary processor," *Proc. VLDB Endow.*, 2014.

[16] J. Ghaderi and et al, "Scheduling storms and streams in the cloud," in *SIGMETRICS '15*.

[17] Y. Shuf and et al, "Characterizing the memory behavior of java workloads: A structured view and opportunities for optimizations," *SIGMETRICS Perform. Eval. Rev.*, 2001.

[18] Intel 64 and ia-32 architectures optimization reference manual, url: http://www.intel.sg/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[19] Performance inspector, url: http://perfinsp.sourceforge.net.

[20] Classmexer agent. http://www.javamex.com/classmexer/.

[21] Intel vtune amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[22] Fraud-detection, url: https://pkghosh.wordpress.com/2013/10/21/real-time-fraud-detection-with-sequence-mining/.

[23] Data request to 98 world cup web site, url: ita.ee.lbl.gov/html/contrib/worldcup.html.

[24] Intel lab data, url: http://db.csail.mit.edu/labdata/labdata.html.

[25] G. Bianchi and et al, "On-demand time-decaying bloom filters for telemarketer detection," *SIGCOMM Comput. Commun. Rev.*, 2011.

[26] Y. Zheng and et al, "Mining interesting locations and travel sequences from gps trajectories," in *WWW '09*.

[27] A. Arasu and et al, "Linear road: A stream data management benchmark," in *VLDB '04*.

[28] D. Abadi, D. Carney, U. etintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, 2003.

[29] D. J. Abadi and et al, "The Design of the Borealis Stream Processing Engine," in *CIDR'05*.

[30] N. Jain and et al, "Design, implementation, and evaluation of the linear road bnchmark on the stream processing core," in *SIGMOD '06*.

[31] M. J. Sax and M. Castellanos, "Building a transparent batching layer for storm," 2014.

[32] T. risch, uppsala university linear road implementations, april 2014, url: http://www.it.uu.se/research/group/udbl/lr.html.

[33] D. Detlefs and et al, "Garbage-first garbage collection," in *ISMM'04*.

[34] D. S. H. Olivier Goldschmidt, "A polynomial algorithm for the k-cut problem for fixed k," *Mathematics of Operations Research*.

[35] S. Ch, , and et al, "Telegraphcq: Continuous dataflow processing for an uncertain world," in *CIDR'03*.

[36] Benchmarking streaming computation engines at yahoo!, url: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at.

[37] M. R. Mendes and et al, "Performance evaluation and benchmarking," R. Nambiar and M. Poess, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. A Performance Study of Event Processing Systems.

[38] J. Chauhan and et al, "Performance evaluation of yahoo! s4: A first look," in *3PGCIC '12*.

[39] K.-L. Tan and et al, "In-memory databases: Challenges and opportunities from software and hardware perspectives," *SIGMOD Rec.*, 2015.

[40] H. Zhang and et al, "In-memory big data management and processing: A survey," *TKDE '15*.

[41] J. Zhou and K. A. Ross, "Buffering databse operations for enhanced instruction cache performance," in *SIGMOD '04*.

[42] S. Harizopoulos and A. Ailamaki, "Improving instruction cache performance in oltp," *ACM Trans. Database Syst.*, 2006.

[43] A. Ailamaki and et al, "Dbmss on a modern processor: Where does time go?" in *VLDB '09*.

[44] A. Shatdal and et al, "Cache conscious algorithms for relational query processing," in *VLDB '94*.

[45] P. A. Boncz and et al, "Database architecture optimized for the new bottleneck: memory access," in *VLDB '99*.

[46] C. Balkesen and et al, "Main-memory hash joins on multi-core CPUs: tunning to the underlying hardware," in *ICDE '13*.

[47] J. Giceva and et al, "Deployment of query plans on multicores," *Proc. VLDB Endow.*, 2014.

[48] S. Harizopoulos and A. Ailamaki, "A case for staged database systems," in *CIDR'03*.