2017 IEEE 33rd International Conference on Data Engineering

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

Shuhao Zhang[1,2], Bingsheng He[2], Daniel Dahlmeier[1], Amelie Chi Zhou[3], Thomas Heinze[4]
[1]SAP Innovation Center Singapore, [2]National University of Singapore, [3]INRIA, [4]SAP SE Walldorf

***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 aiming at scaling out using a cluster of commodity machines and built around a number of key design aspects: a) pipelined processing with message passing, b) on-demand data parallelism, and c) JVM based implementation. However, there lacks a study on those key design aspects on modern scale-up architectures, where more CPU cores are being put on the same die, and the on-chip cache hierarchies are getting larger, deeper, and complex. Multiple sockets bring non-uniform memory access (NUMA) effort. In this paper, we revisit the aforementioned design aspects on a modern scale-up server. Specifically, we use a series of applications as micro benchmark to conduct detailed profiling studies on Apache Storm and Flink. From the profiling results, we observe two major performance issues: a) the massively parallel execution 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 benefits 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 design aspects in building those existing DSP systems:

a) *Pipelined processing with message passing:* A streaming application is usually implemented as multiple operators with data dependencies, and each operator performs three basic tasks continuously, i.e., receive, process and output. Such a pipelined processing design enables DSP systems to support very low latency processing, which is one of the key requirements 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 in many DSP systems. Specifically, users can configure the number of threads in each operator (or function) independently in the streaming application. Such an 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 (Java Virtual Machine). Although the use of JVM-based programming language makes the system development more productive (e.g., built-in memory management), many JVM runtime performance issues such as data reference and garbage collection are transparent to programmers.

On the other hand, modern servers are being deployed in the cluster environment. More CPU cores are being put on the same 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 performance factor for data management 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 above common design aspects of DSP systems on modern architectures.

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

Through detailed profiling studies with our benchmark on a four-socket machine, we make the following key observations.

First, the design of supporting both pipelined and data parallel processing leads to a very complex massively parallel execution model in DSP systems, which poorly utilizes

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.119

modern multi-core processors. Based on our profiling results, a significant portion (∼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 large instruction footprint between two consecutive invocations of the same function.

Second, the design of continuous message passing between operators causes a serious performance degradation to DSP systems running on multiple CPU sockets. Furthermore, the current design of data parallelism in DSP systems tends to equally partition input streams regardless of the location of executors (i.e., they may be scheduled on different CPU sockets), which overlooks 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 severely limit the scalability of DSP systems.

Third, the JVM runtime brings two folds of overhead to the execution, and they are moderate in DSP systems. 1) The translation lookaside buffer (TLB) stalls take 5∼10% and 3∼8% of the total execution time for most applications on Storm and Flink, respectively. The major causes include the frequent pointer referencing issues in data accesses and Java execution. 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 data-intensive platforms with large memory footprints (e.g., [13], [14]).

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 instruction footprint for processing a tuple so that the instruction cache performance can be improved; 2) *NUMA-aware executor placement* to make thread placement aware of remote memory accesses across sockets. 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 1.2∼3.1x throughput improvement on Storm and Flink, respectively.

To the best of our knowledge, this is the first detailed study of common design aspects 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 setting, 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 Sections IV and V, respectively. 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.

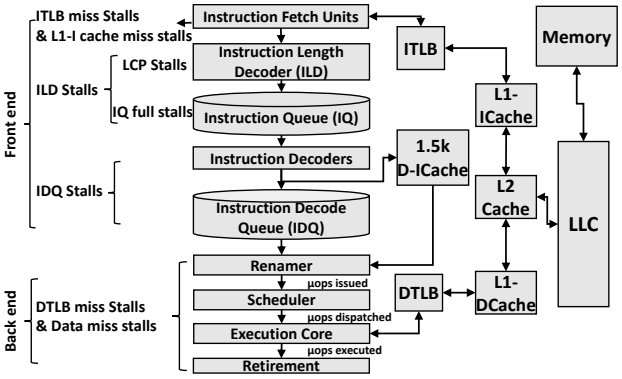


Fig. 1: Pipeline execution components of processor: (left) various stalls caused in the pipeline, (middle) pipelines interactions with cache and memory systems and (right) the interactions among the cache, TLB, and memory systems.

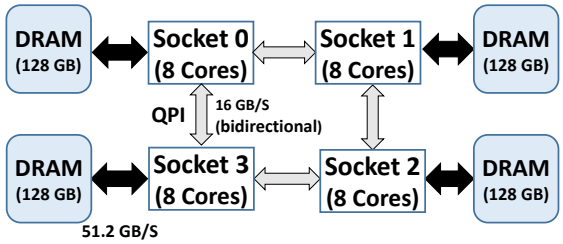


Fig. 2: NUMA topology and peak bandwidth of our sever.

II. Preliminaries and Background

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

A. Multi-Socket Multi-core Processors

Modern processors consist of multiple different hardware components with deep execution pipelines, as shown in Figure 1. We also illustrate the stalls and the interactions among pipelines and memory systems in the figure. The pipeline can be divided into the front-end component and the back-end component [15].

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 decoded μops. Future references to the same μops can be served by it without performing the fetch and decode stages. D-ICache is continuously enhanced in terms of 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 among 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, non-uniform memory access (NUMA) brings more performance issues. Figure 2 illustrates the NUMA topology of our sever with four sockets. Each CPU socket has its local memory, which is uniformly shared by the cores on the socket. Sockets are connected by a much slower (compared to local memory access) channel called Quick Path Interface (QPI).

B. Data Stream Processing Systems

In the following, we introduce three design aspects of Storm and Flink: 1) pipelined processing with message passing, 2) on-demand data parallelism, and 3) JVM-based implementation.

Pipelined processing with message passing. We describe the execution model with a general definition [16]. 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. 1) a *data source* operator generates (or receives from the external environment) events to feed into the topology, and 2) a *data processor* operator encapsulates specific processing logics such as filtering, transforming or user-defined function.

In a shared-memory environment, an operator (continuously) writes its output data into the local memory. For each output data, the operator also pushes a tuple consisting of a reference (i.e., pointer) of the output data into its output queue. The corresponding consumer (continuously) fetches the tuple from the queue, and then accesses on the output data generated by the producer operator through memory fetches. In other words, the communication between two operators are through the data reference. 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 3 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 remote memory accesses across sockets during message passing in DSP systems, which we will study in details 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

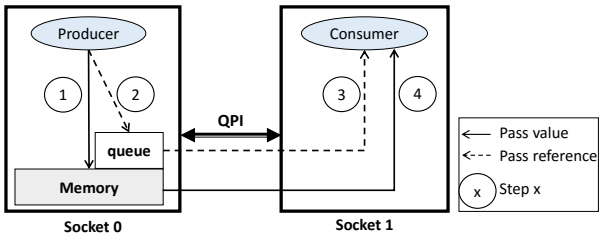


Fig. 3: Message passing mechanism.

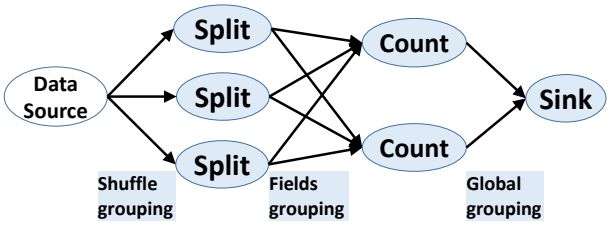


Fig. 4: Word-count execution graph.

operator is carried out by one or more physical threads, which are referred to as executors. Input stream of an operator is (continuously) partitioned among its executors. 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 4. In this example, the split, count, and sink operators have three, two and one executors, respectively. Streams are partitioned and delivered to specific destination executors according to the grouping strategy specified by the user. In the previous example, the shuffle grouping strategy used in the data source operator uniformly distributes the tuples to each split executor. Meanwhile, each spilt executor sends tuples to count executors according to the attribute of the tuple (specified as field grouping) so that the same key (i.e., the same word) 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. Two 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, which may lead to pointer chasing and stress the cache and TLB of the processor heavily.

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 (e.g., [14], [13]). To this end, some DSP systems have even implemented its own memory management besides JVM (e.g., Flink).

TABLE I: JVM profile flags

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

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 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.

First, 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 applications. Second, 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 [17] for gathering detailed instruction-tracing information. We measure the size of the objects created at runtime using the *MemoryUtil* tool from the Classmexer library [18].

Processor profile. We systematically categorize 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 [19] for profiling at the processor level.

Similar to the recent study [14], 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-Cache Stalls	Stall time due to L2-Cache misses that hit in LLC
	LLC stalls (local)	Stall time due to LLC misses that hit in local memory
	LLC stalls (remote)	Stall time due to LLC misses that hit in memory of other socket

TABLE III: Detailed specification on our testing environment

Component		Description
Processor		Intel Xeon E5-4640, Sandy Bridge EP
	Cores (per socket)	8 * 2.4GHz (hyper-threading disabled)
	Sockets	4
	L1 cache	32KB Instruction, 32KB Data per core
	L2-Cache	256KB per core
	Last level cache	20MB per socket
Memory		4 * 128GB, 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. Micro 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 streaming 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 can be easily applied 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 relatively low. The applications also have variety of memory bandwidth demands. *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 topologies of the seven applications.

In the following, we describe each application including its application scenario, implementation details and input setup. In all applications, we use a simple sink operator to measure the throughput.

Stateful Word Count (WC): The stateful word-count counts and remembers the frequency of each received word unless the application is killed. 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 and is updated for each receiving word. 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): 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 a detection algorithm called *missProbability* [20] with sequence window size of 2 events. The topology of FD has only one operator, named as Predict, which is used to maintain and update the state transition of each customer. 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): Log processing represents the streaming application of performing real-time analyzing on system logs. 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 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. We use a subset of the web request data (with 4 million events) from the 1998 World Cup Web site [21]. For data privacy protection, each actual IP address in the requests is mapped to randomly generated but fixed IP address.

Spike Detection (SD): Spike detection tracks measurements from a set of sensor devices and performs moving aggregation calculations. 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. We use the Intel lab data (with 2 million tuples) [22] for this application. The detection threshold of moving average values is set to 0.03.

Spam Detection in VoIP (VS): Similar to fraud detection, spam detection is a use case of outlier detection. 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), and 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 [23]. We use a synthetic data set with 10 million records for this application. Each record contains data on a calling number, called number, calling date, answer time, call duration, and call established.

Traffic Monitoring (TM): Traffic monitoring performs real-time road traffic condition analysis, with real-time mass GPS data collected from taxis and buses[1]. 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. We use a subset (with 75K events) of GeoLife GPS Trajectories [24] for this application.

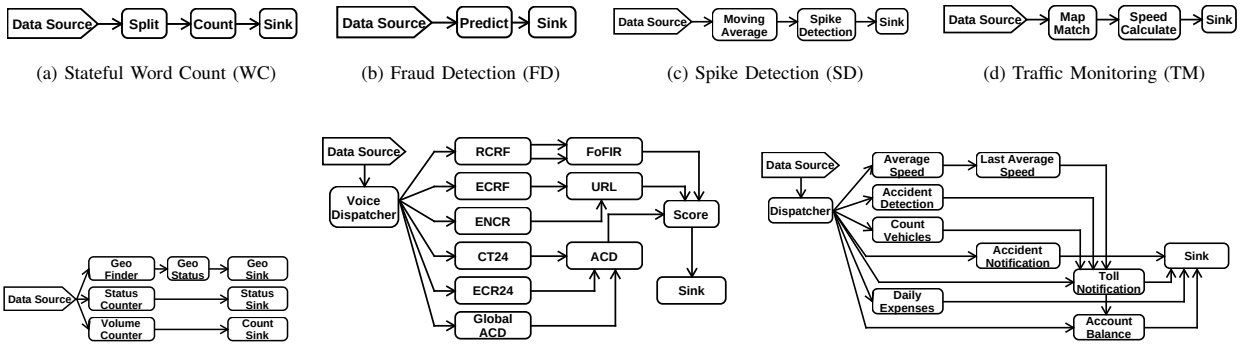
Linear Road (LR): Linear Road (LR) is used for measuring how well a DSP system can meet real-time query response requirements in processing a large volume of streaming and historical data [25]. It models a road toll network, in which tolls depend on the time of the day and level of congestions. Linear Road has been used by many DSP systems, e.g., Aurora [26], Borealis [27], and System S [28]. 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. We have followed the implementation of the previous study [29] for LR. Several queries specified in LR are implemented as operators and integrated into a single topology. The input to LR is a continuous stream of position reports and historical query requests. We merge the two data sets obtained from [30] resulting in 30.2 million input records (including both position reports and query requests) and 28.3 million historical 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 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 bandwidth utilizations of running different applications on Storm and Flink on a single CPU socket. We measure the overall resource utilization during stable

[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: CPU and memory bandwidth utilization on a single CPU socket

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

execution by avoiding the beginning and ending phases of each application. We have 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 utilization. CPU utilization of LG and LR is median, and that of WC and SD is low.

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). Figures 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. *First*, on a single socket, most of the applications scale well with the increasing number of CPU cores for both Storm and Flink. *Second*, most applications perform only slightly better or even worse on multiple sockets than on a single socket. FD and SD become even worse on multiple sockets than on a single socket, due to their relatively low compute resource demand. Enabling multiple sockets only brings additional overhead of remote memory accesses. WC, LG and VS perform similarly for different numbers of sockets. The throughput of LR increases marginally with the increasing number of sockets. *Third*, TM has a significantly higher throughput in both systems on four sockets than on a single socket. This is because TM has high resource demands on both CPU and memory bandwidth.

V. Study the Impact of Common Designs

In the following section, we investigate the underlying reasons for the performance degradation and how the three design aspects (i.e., pipelined processing with message passing, on-demand data parallelism, and JVM-based implementation) 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 execution model, message passing and stream partitioning and JVM runtime environment.

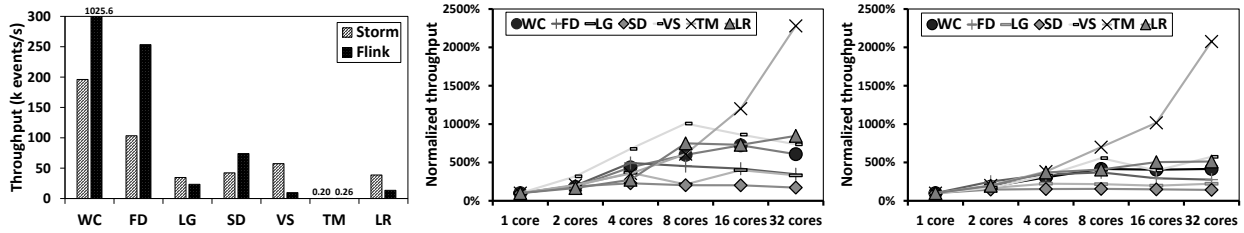
A. Execution time breakdown

Finding (1): During execution of most applications (except TM) on both Storm and Flink, $\sim$ 70% of their execution times are spent on processor stalls.

Figure 7 shows the execution time breakdown of Storm and Flink running on a single socket on different processor components as introduced in Section II-A. We find that 59$\sim$77% and 58$\sim$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.

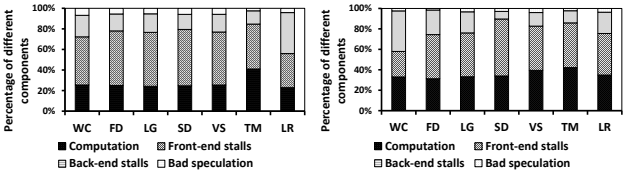
Front-end stalls account for 35$\sim$55% and 25$\sim$56% of the total execution time of Storm and Flink, respectively. This result is significantly different from the batch processing framework (e.g., [13]). Back-end stalls account for approximately 13$\sim$40% and 7$\sim$40% of the total execution time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

In the following, we examine the processor stalls in more details with respect to the three designs of DSP systems (i.e.,



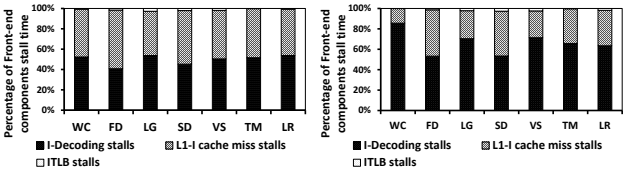
(a) Evaluation of seven applications on a single socket. (b) Storm with varying number of cores/sockets. (c) Flink with varying number of cores/sockets.

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.

pipelined processing with message passing, on-demand data parallelism, and JVM-based implementation).

B. Massively parallel execution model

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

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

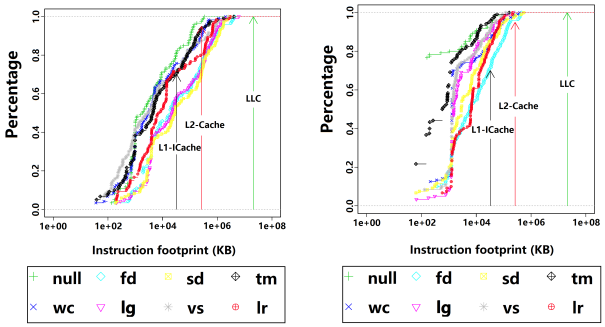
L1-ICache miss stalls: Our investigation reveals that there are two primary sources responsible for the high L1-ICache miss. *First*, due to the lack of a proper thread scheduling mechanism, the massive threading execution runtime of both Storm and Flink produces frequent thread context switching. *Second*, 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 20KB. As the size of current L1-ICache (32KB per core) is still fairly limited, it cannot hold those instructions at runtime, 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"). Figure 9 illustrates the cumulative density function (CDF) of instruction footprints on a log scale, which stands for the percentage that instruction footprint is no larger than a certain number of distinct instructions.

We add three solid vertical arrows 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 footprints during execution. *Second*, the cross-over points of L1-ICache line and different CDF curves are between $0.3 \sim 0.5$ for Storm and $0.6 \sim 0.8$ for Flink. It means, around $50 \sim 70\%$ and $20 \sim 40\%$ of the instruction footprints between two consecutive calls to the same functions are larger than the L1-ICache in Storm and Flink, respectively. This causes severe L1-ICache stalls. 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 smaller 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 broken down 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 pre-fetched instructions in a separate buffer while the processor is executing the current instruction. Due to the large footprint between two consecutive invocation of the



(a) Storm (b) Flink

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

same function, IQ full stalls are frequent and contribute nearly 20% of front-end component stall time for all applications on both Storm and Flink. On the other hand, the LCP stalls are negligible, with less than 0.05% for all applications according to our measurement.

Another important aspect of I-Decoding stalls is the IDQ stalls, which consist 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 invalidated, which subsequently causes a switch penalty (i.e., the back-end has to re-fetch instructions from the legacy decoder pipeline). *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 decoder pipeline. 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 misses in the D-ICache and eventually causes high IDQ stalls.

C. Message passing and stream partitioning

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 accesses during the fetching of input data. Furthermore, the current design of data parallelism has overlooked the NUMA effect.

Recently, the NUMA-aware allocator has already been implemented in the Java HotSpot Virtual Machine to take advantage of such infrastructures, which provides automatic memory placement optimizations for Java applications. We 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 we have observed poor scalability in both Storm and Flink on multiple sockets. The main problem is the high remote memory access

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%

overhead due to the heavy pipelined message passing design. During execution, each executor needs to fetch data from the corresponding producer continuously. On the multi-socket and multi-core architectures, 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. (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 its local memory. This comes with very high access penalty, and it happens when producer and consumer executors are located in different CPU sockets.

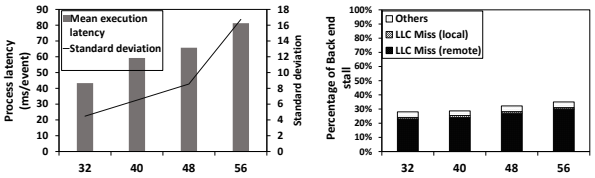
As a result, the data access cost is depending on the location of the producer and consumer, which creates significant performance divergence among parallel executors of even the same operator. However, neither Storm nor Flink is aware of such performance heterogeneity issues and continuously distributes equal amounts (in the case of shuffle grouping) of tuples among executors. Table V shows the LLC miss stalls for 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 partitioning. 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 increases along with the growing number of executors. Figure 10b shows that the back-end stalls gradually become worse with the increase in the number of executors. This indicates the remote memory access penalty prevents DSP systems from scaling well.

D. JVM Runtime Environment

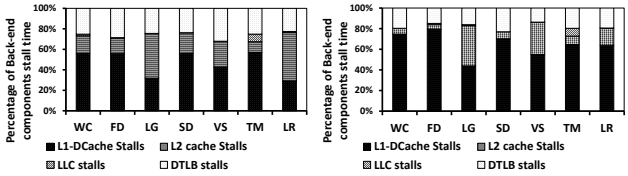
Finding (4): The overhead of JVM runtime contains two major and moderate components. First, TLB stalls account for $5 \sim 10\%$ and $3 \sim 8\%$ on Storm and Flink, respectively. This is caused by frequent pointer referencing in data accesses and Java execution. Second, the overhead of GC in running streaming applications ($1 \sim 3\%$) is insignificant.

Both Storm and Flink are implemented using JVM-based



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

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



(a) Storm (b) Flink

Fig. 11: Back-end stall breakdown.

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 breaks 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-DCache and L2-Cache dominate the back-end stalls in both systems. We measure the size of intermediate results generated during execution of all streaming applications in our benchmark, and we have the following observations. First, the private data structures accessed during execution do not often fit into L1-DCache (32KB) but can fit into L2-Cache (256KB), which causes frequent L1-DCache stalls. Second, the output data for message passing mostly fit into the LLC, and cannot fit into L2 cache. As a result, data passing among executors in a single socket usually get served by the shared LLC (20MB).

TLB stalls: Tuples are passed with reference (instead of the actual data) in both systems. Upon a class loading in java, the *invokevirtual* instruction is triggered to search the method table and identify the specific method implementation, which may cause random accesses on method tables. As a result, the frequent pointer references lead to stress on TLB on both systems. Our further investigation found that enabling *huge page* improves the performance of both Storm and Flink marginally for all seven applications.

Garbage collection overhead: We use G1GC [31] 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. Based on GC logs, we find that no major GC occurs 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 an older version of GC named parallelGC. When the parallelGC mechanism is used, the overhead of GC increases to around $10 \sim 15\%$, which indicates the effectiveness of G1GC. Nevertheless, we plan to evaluate the impact of GC with more extensive applications.

VI. Towards more efficient DSP systems

In this section, we present our initial attempt to address the performance issues found in the previous section. We present two optimization techniques, including non-blocking tuple batching (to reduce instruction cache stalls) 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. We note that, the two optimization techniques address the efficiency issues of the two common designs (i.e., pipelined processing with message passing and on-demand data parallelism), and we conjecture that the optimizations can be applied to other DSP systems with the same designs.

A. Non-blocking Tuple Batching

Our profiling results suggest that the large instruction footprints between two consecutive invocations of the same function cause 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 tuples together before passing to the consumer executor for processing. In this way, each function invocation can process multiple tuples so that the instruction footprint between two consecutive invocations of the same function is reduced. Similar ideas of tuple batching are already proposed [29] or in use in some DSP systems [2]. However, those techniques rely on a buffering stage, which introduces wait delay in execution. For example, Sax et al. [29] proposed to create an independent batching buffer for each consumer in order to batch all tuples that will be processed by the same consumer. Tuples are not emitted until the corresponding buffer becomes full. However, the additional explicit buffering delay in every executor may introduce serious negative impact on the system latency. 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 processes a batch of tuples and outputs multiple tuples, we try to put its output tuples together as a batch, or multiple batches each containing tuples belonging to the same key. Once the corresponding consumer receives such a batch, it can then process multiple tuples from the batch with a single function invocation and further batching its output tuples in a similar manner. Our solution requires that 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 tuples (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 operators in an application, this kind of batching has little overhead.

It is rather straightforward to implement the non-blocking tuple batching algorithm for any grouping policy (Section II-B) except the key-grouping policy (i.e., fields grouping), as we can simply group together all the output tuples of an executor. However, if an executor uses fields grouping, simply putting output tuples into one batch may cause errors [29] due to wrongly sending output tuples targeting at different consumers based key in each tuple. Existing batching techniques rely on a buffering stage in order to resolve such issue [29]. In contrast, we develop an algorithm for non-blocking tuple batching of fields grouping, as illustrated in Algorithm 1.

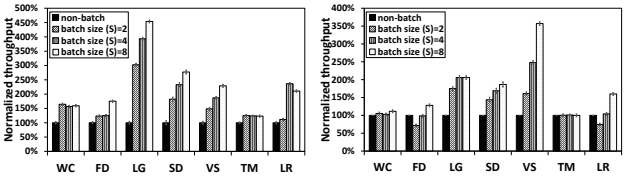
The basic idea is to store multiple output tuples into a multi-valued hash map (at lines 10-12), and the fields (i.e., keys) used in choosing consumer are re-computed based on the fields originally declared (at lines 10-11). At line 4, the HashMultimap is the multi-value hash map used to batch multiple 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 B
 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 B 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, where the values of the same key are
        maintained in a list L;
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 of the same key as a batch.*/
17:     emit(< K_i, L >);
18: end for
```

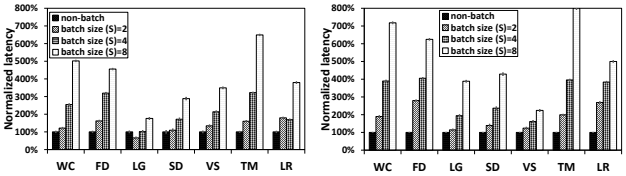
We now study the impact of tuple batching optimization by varying S. Figure 12 illustrates the normalized throughput of Storm and Flink with tuple batching optimization for different applications on a single CPU socket. Results are normalized to the original non-batch setting of Storm and Flink (denoted as non-batch). As expected, 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. Figure 13 shows the normalized average latency per tuple under different batch sizes. Comparing Figures 12 and 13, we observe a clear trade-off between the throughput and latency. Meanwhile, our non-blocking tuple batching scheme preserves a sublinear increase in process latency for most applications, which is due to the much-improved performance and no explicit buffering delay.



(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.

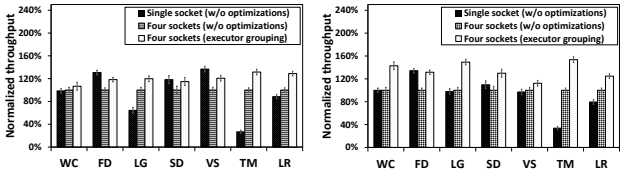
Definition 1: ***Executor placement.*** Given a topology execution graph T and the set of executors W in T, an executor placement $P(T,k)$ represents a plan of placing W onto k CPU sockets. k can be any integer smaller than or equal to the total number of sockets in a NUMA machine.

Definition 2: We denote the remote memory access penalty per unit as R, and the total size of tuples transmitted between any two executors w and w' ($w, w' \in W$) as $Trans(w,w')$. Each placement $P(T,k)$ 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 x as ξ_x, where $x = 1,\ldots,k$.

$$Cost(P(T,k)) = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{w\in\xi_i, w'\in\xi_j} R * Trans(w,w') \quad (1)$$

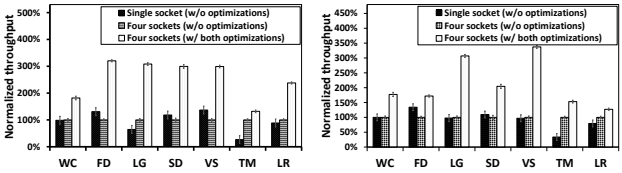
Definition 3: The **optimal executor placement**, denoted by P_{opt}, is defined as $Cost(P_{opt}(T,k)) \leq Cost(P(T,k))$, $\forall P \in Q$, where Q is the set of all feasible executor placement solutions.

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 [32].

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. A mapping from T to G is defined as follows: **(I)** $\forall$ executor $w \in W$, there is a one-to-one mapping from w to a vertex in G. **(II)** For any producer-consumer ($< w, w' >$, $w, w' \in W$) message passing relationships in T, there is a one-to-one mapping to one edge in G. The communication cost ($R * Trans(w, w')$) 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 [32] for solving this problem by fixing k from 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 1.3∼3.2x and 1.2∼3.1x improvement on the performance 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: DSP systems have attracted a great amount of research effort. A number of systems have been developed, for example, TelegraphCQ [33], Borealis [27], Yahoo S4 [5], IBM System S [28] and the more recent ones including Storm [1], Flink [2] and Spark Streaming [3]. However, little attention has been paid to the research on the key and common design aspects of various DSP systems on modern multi-core processors. Through profiling studies on those design aspects in Storm and Flink, we are able to identify the common bottlenecks of those system design aspects when running on multi-socket multi-core processors. There have been a few studies on comparing different DSP systems. A recent study [34] 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 throughput at a relatively high latency. A comparison of S4 and Storm [35] uses a micro-benchmark to understand the performance issues of the systems regarding scalability, execution time and fault tolerance. A similar study [36] has been conducted to compare the performance characteristics of three DSP systems, including System S, S4, and Esper. However, those studies still aim 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 design aspects in DSP systems and conduct detailed profiling on those design aspects of DSP systems on scale-up architectures. Our findings could be generalized to DSP systems other than Storm and Flink, and even future DSP systems.

Optimization for streaming systems: Several recent efforts have been made with the aim of optimizing distributed DSP systems. Sax et al. [29] proposed to create an independent batching buffer for each consumer in order to batch all tuples that will be processed by the same consumer to avoid wrong fields grouping problem. However, the additional explicit buffering delay in every executor may introduce serious negative impact on the system latency. T-Storm [6] assigns/re-assigns executors according to run-time statistics in order to minimize inter-node and inter-process traffic while ensuring load balance. However, based on our study, executor placement inside a single machine also needs to be considered due to the NUMA effect.

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 [37], [38]. There have been studies on optimizing the instruction cache performance [39], [40], the memory and cache performance [41], [42], [43], [44] and NUMA [9], [10], [45]. In the following, we focus on two kinds of studies

that are closely related to our optimization techniques. Related to the tuple batching optimization, StagedDB [46] reveals that the interference caused by context-switching during the execution results in high performance penalties due to additional conflict and cache misses. They hence introduced a staged design for DBMS, which breaks down DBMS into smaller self-contained modules. The parallel pipeline processing design of DSP systems also has a similar serious problem of frequent instruction cache misses. The design of non-blocking tuple batching is inspired by StagedDB [46]. Query deployment on multi-core machines [45] proposed recently is related to our study of NUMA-aware executor placement. We apply a similar idea to query operator placement in DSP systems in order to minimize the impact of NUMA.

VIII. Conclusions

This paper revisits three common design aspects of modern DSP systems on modern multi-socket multi-core architectures, including a) pipelined processing with message passing, b) on-demand data parallelism, and c) JVM-based implementation. Particularly, we conduct detailed profiling studies with micro benchmark on Apache Strom and Flink. Our results show that those designs have underutilized the scale-up architectures in these two key aspects: a) The design of supporting both pipelined and data parallel processing results in a very complex massively parallel execution model in DSP systems, which causes high front-end stalls on a single CPU socket; b) The design of continuous message passing mechanisms between operators severely limits the scalability of DSP systems on multi-socket multi-core architectures. We further present two optimizations to address those performance issues and demonstrate promising performance improvements.

Acknowledgement

This work is partially funded by a MoE AcRF Tier 1 grant (T1 251RES1610), a startup grant of NUS in Singapore and NSFC Project 61628204 in China. Shuhao Zhang's work is partially funded by the Economic Development Board and the National Research Foundation of Singapore.

References

[1] "Apache Storm," http://storm.apache.org/.
[2] "Apache Flink," https://flink.apache.org/.
[3] M. Zaharia and et al., "Discretized Streams: Fault-tolerant Streaming Computation at Scale," in *SOSP*, 2013.
[4] "Apache Samza," http://samza.apache.org/.
[5] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *ICDMW*, 2010.
[6] J. Xu and et al., "T-Storm: Traffic-Aware Online Scheduling in Storm," in *ICDCS*, 2014.
[7] L. Aniello and et al., "Adaptive online scheduling in storm," in *DEBS*, 2013.
[8] B. Peng and et al., "R-Storm: Resource-Aware Scheduling in Storm," in *Middleware*, 2015.
[9] V. Leis and et al., "Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age," in *SIGMOD*, 2014.
[10] Y. Li and et al., "NUMA-aware algorithms: the case of data shuffling," in *CIDR*, 2013.
[11] "SGI UVTM 300H System Specifications," 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] A. J. Awan and et al., "Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server," in *BDCloud*, 2015.
[14] S. Sridharan and J. M. Patel, "Profiling R on a Contemporary Processor," *Proc. VLDB Endow.*, 2014.
[15] Intel 64 and IA-32 Architectures optimization Reference Manual. http://www.intel.sg/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.
[16] J. Ghaderi and et al., "Scheduling Storms and Streams in the Cloud," in *SIGMETRICS*, 2015.
[17] "Performance Inspector," http://perfinsp.sourceforge.net.
[18] "Classmexer agent," http://www.javamex.com/classmexer/.
[19] "Intel VTune Amplifier," https://software.intel.com/en-us/intel-vtune-amplifier-xe.
[20] "Fraud-detection," https://pkghosh.wordpress.com/2013/10/21/real-time-fraud-detection-with-sequence-mining/.
[21] "Data request to 98 world cup web site," ita.ee.lbl.gov/html/contrib/WorldCup.html.
[22] "Intel lab data," http://db.csail.mit.edu/labdata/labdata.html.
[23] G. Bianchi and et al., "On-demand time-decaying bloom filters for telemarketer detection," *SIGCOMM Comput. Commun. Rev.*, 2011.
[24] Y. Zheng and et al., "Mining Interesting Locations and Travel Sequences from GPS Trajectories," in *WWW*, 2009.
[25] A. Arasu and et al., "Linear Road: A Stream Data Management Benchmark," in *VLDB*, 2004.
[26] D. Abadi and et al., "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, 2003.
[27] D. J. Abadi and et al., "The Design of the Borealis Stream Processing Engine," in *CIDR*, 2005.
[28] N. Jain and et al., "Design, Implementation, and Evaluation of the Linear Road Bnchmark on the Stream Processing Core," in *SIGMOD*, 2006.
[29] M. J. Sax and M. Castellanos, "Building a Transparent Batching Layer for Storm," 2014.
[30] "Uppsala University Linear Road Implementations," http://www.it.uu.se/research/group/udbl/lr.html.
[31] D. Detlefs and et al., "Garbage-first Garbage Collection," in *ISMM*, 2004.
[32] O. Goldschmidt and D. S. Hochbaum, "A Polynomial Algorithm for the k-Cut Problem for Fixed k," *Mathematics of Operations Research*.
[33] S. Chandrasekaran and et al., "Telegraphcq: Continuous dataflow processing for an uncertain world," in *CIDR*, 2003.
[34] "Benchmarking Streaming Computation Engines at Yahoo!" https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at.
[35] 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.
[36] J. Chauhan and et al., "Performance Evaluation of Yahoo! S4: A First Look," in *3PGCIC*, 2012.
[37] K.-L. Tan and et al., "In-memory Databases: Challenges and Opportunities From Software and Hardware Perspectives," *SIGMOD Rec.*, 2015.
[38] H. Zhang and et al., "In-Memory Big Data Management and Processing: A Survey," *TKDE*, 2015.
[39] J. Zhou and K. A. Ross, "Buffering Databse Operations for Enhanced Instruction Cache Performance," in *SIGMOD*, 2004.
[40] S. Harizopoulos and A. Ailamaki, "Improving Instruction Cache Performance in OLTP," *ACM Trans. Database Syst.*, 2006.
[41] A. Ailamaki and et al., "DBMSs on a Modern Processor: Where Does Time Go?" in *VLDB*, 2009.
[42] B. He and et al., "Cache-conscious automata for xml filtering," in *ICDE*, 2005.
[43] P. A. Boncz and et al., "Database Architecture Optimized for the new. Bottleneck: Memory Access," in *VLDB*, 1999.
[44] C. Balkesen and et al., "Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware," in *ICDE*, 2013.
[45] J. Giceva and et al., "Deployment of Query Plans on Multicores," *Proc. VLDB Endow.*, 2014.
[46] S. Harizopoulos and A. Ailamaki, "A Case for Staged Database Systems," in *CIDR*, 2003.