BriskStream: NUMA-aware Data Stream Processing

Shuhao Zhang[1,2] Jiong He[3] Amelie Chi Zhou[4] Bingsheng He[2]
[1]SAP Innovation Center Network [2]National University Singapore [3]Advanced Digital Sciences Center
[4]Shenzhen University

shuhao.zhang@sap.com jiong.he@adsc.com.sg chi.zhou@szu.edu.cn
hebs@comp.nus.edu.sg

ABSTRACT

Driven by the rapidly increasing demand for handling real-time data streams, data stream processing (DSP) systems have attracted a lot of attention. Designed with the scalability in mind, most existing DSP systems such as Apache Storm and Flink aim at scale-out architectures using a cluster of commodity machines. On the other hand, *scale-up* servers with multi-socket multi-core processors are being deployed in the cluster environment. Recent studies have shown that existing DSP systems seriously underutilize the underlying complex hardware micro-architecture. There is a must to redesign the DSP system for multi-socket multi-core architectures. In this paper, we present BriskStream, an in-memory data stream processing system that is built specifically for the scale-up architecture with multi-socket multi-core processors. BriskStream is designed and optimized to use system memory and caches more efficiently. Moreover, we introduce a novel performance model that accurately characterizes the performance of stream processing application under different execution plans (considering both operator placement and replication) in the presence of NUMA effect. Based on this model, we further develop a branch and bound based algorithm that exploits the overall structure of both streaming applications and the NUMA architecture to achieve optimal performance. The experimental evaluations demonstrated that BriskStream significantly outperforms the existing DSP systems including Apache Storm and Flink over five common applications on two modern servers with eight CPU sockets, usually at the scale of an order of magnitude.

PVLDB Reference Format:
Shuhao Zhang, Jiong He, Amelie Chi Zhou, Bingsheng He. BriskStream: NUMA-aware Data Stream Processing. *PVLDB*, 11 (4): xxxx-yyyy, 2017.
DOI: https://doi.org/TBD

1. INTRODUCTION

Data stream processing enables the timely reactions to changing conditions and critical events, which is often a must requirement in many domains, such as trading, fraud detection and system monitoring. For instance, online marketplace providers such as eBay run sophisticated fraud detection algorithms on real-time trading activities [32].

Generic stream processing frameworks, often called data stream processing (DSP) systems, have been widely proposed for developers to implement stream processing applications correctly, efficiently, and easily. Over the years, a lot of extensions and optimizations [38, 33, 18] have been proposed to improve the performance of existing DSP systems such as Apache Flink [1], and Apache Storm [2]. Existing DSP systems are mainly designed and optimized for scaling out using a cluster of commodity machines. In particular, existing DSP systems and the optimization techniques focus on providing mechanisms to handle the inherent challenges from the distributed environment settings, such as network communication overhead [38, 34, 9, 26]. On the other hand, modern multi-core processors have demonstrated superior performance in the compute-intensive and latency-sensitive applications [35, 10] with their increasingly great computing capability and larger memory capacity. For example, recent scale-up servers can accommodate even hundreds of CPU cores and multi-terabytes of memory [6].

Nowadays, modern scale-up servers consists of multiple sockets and non-uniform memory access (NUMA) becomes an important performance optimization factor (e.g., [19, 28]). However, recent studies [40] have shown that existing DSP systems seriously underutilize the underlying complex hardware micro-architecture and shows poor scalability on multiple sockets.

In this paper, we introduce BriskStream, a new DSP system that achieves excellent performance on the modern scale-up server with cache-coherent NUMA architecture. BriskStream adopts the commonly used pipelined and operator replication designs inspired by modern DSP systems [40]. Specifically, each operator is treated as a single execution unit (e.g., a Java thread), whose CPU affinity can be configured (i.e., operator placement) and operator can be replicated into multiple copies (i.e., operator replication) running in parallel for higher throughput. We designed BriskStream from the scratch to use system memory and caches efficiently. For instance, BriskStream relies on a shared-memory optimized pipeline execution runtime with the point-to-point communication channels that reduces centralized contention and enables local output caching that significantly improves cache more efficiency.

BriskStream uses a novel NUMA-aware optimizer

Proceedings of the VLDB Endowment, Vol. 11, No. 4

DOI: https://doi.org/TBD

namely *Replication and Location Aware Scheduling* (RLAS) that is able to optimize operator replication setting and placement under the NUMA effect at the same time. Specifically, 1) we introduce a performance model that takes relative location (i.e., relative NUMA distance to each of its producers) of each operator into consideration, which can accurately predict the execution behaviour of the application under different execution plans, and 2) we propose a branch and bound based algorithm to solve the resulting placement problem under the NUMA effect, and we further design a light-weight approach to identify the final execution plan considering both operator replication and placement configurations.

We conduct extensive experimental evaluations over five applications on two modern eight-socket servers with different types of NUMA topology. The results show that BriskStream significantly outperforms two open-sourced DSP systems including Apache Storm and Apache Flink.

Organization. The remainder of this paper is organized as follows. Section 2 covers the necessary background of DSP systems and scale-up servers and reviews related work. Section 3 elaborates how the system designs of BriskStream are optimized for shared-memory environment. Section 4 motivates our RLAS optimization and gives an optimization overview. The performance model and algorithm implementation details are presented in Section 5 and Section 6, respectively. We report extensive experimental results in Section 7, and Section 8 concludes this work.

2. BACKGROUND AND RELATED WORK

In this section, we first briefly introduce data stream processing systems and modern scale-up servers. Then, we discuss related work.

2.1 Data Stream Processing Systems

APIs. In stream processing, events are often processed in multiple stages that are organized into a directed acyclic graph (DAG), where a vertex corresponds to the continuous computation in an operator and an edge represents an event stream flowing downstream from the producer operator to the consumer operator.

Example application. We illustrate the stream processing application with *word-count* (streaming version) containing five operators as depicted in Figure 1.

1. *DataSource* continuously generates new tuple containing a sentence with random words. Each prepared tuple is then fetched by Parser.
2. *Parser* drops tuple with invalid value.
3. *Splitter* processes each input tuple by splitting the sentence into words, and emits each word as a new tuple to Counter.
4. *Counter* maintains and updates a hashmap with the key as the word and value as the number of occurrence of the corresponding word. Each time it receives a word, it updates its hashmap and emits a tuple containing the word and its current occurrence.
5. *Sink* increments a counter each time it receives tuple from Counter. We use this operator to monitor the performance of the application.

Runtime. There are two important aspects of runtime designs of modern DSP systems. First, the common wisdom of designing the execution runtime of DSP systems is

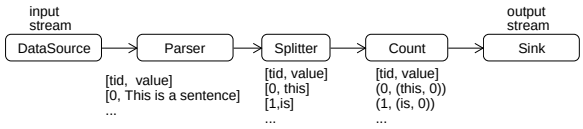


Figure 1: Example application: word-count.

to treat each operator as a single execution unit (e.g., a Java thread) and runs them in a pipelining way [40]. Second, for scalability, each operator may be executed independently in multiple threads (i.e, operator replication). In particular, producer partitions its output stream into multiple substreams to each downstream consumer replicas for data parallel processing to increase throughput. The partition ratio depends on partition policy required by the application such as shuffle grouping and fields grouping [2].

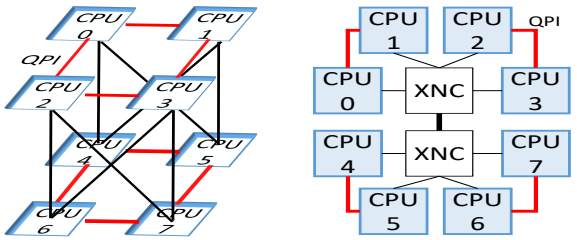
2.2 Modern Scale-up Servers

Modern machines are scale to multiple sockets with non-uniform-memory-access (NUMA) architecture. That is, each socket in a NUMA system has its own "local" memory and is connected to other sockets and, hence to their memory, via one or more links. Therefore, access latency and bandwidth vary depending on whether a core in a socket is accessing "local" or "remote" memory. Such NUMA effect requires one to carefully align the communication patterns accordingly to get good performance.

There is no standard NUMA system, but different configurations exist in nowadays market, which further complicates the software optimization on them. Figure 2 illustrates the NUMA topologies of our servers. In the following, we use "Server A" to denote the first, and "Server B" to denote the second. Server A can be categorized into glue-less NUMA server, where CPUs are connected directly/indirectly through QPI or vendor custom data interconnects. Server B employs a vendor customized node controller (called XNC) that interconnects upper and lower CPU tray (each tray contains 4 CPU sockets). The node controller maintains a directory of the contents of each processors cache, and significantly reduces remote memory latency. Table 1 shows the detailed specification of our two NUMA servers. NUMA characteristics, such as local and inter-socket idle latencies and peak memory bandwidths, are measured with Intel Memory Latency Checker [3]. These two machines have different NUMA topologies, which lead to different access latencies and throughputs across CPU sockets.

2.3 Related Work

Operator scheduling: The operator scheduling problem in DSP systems concerns determining a good placement of application operators on the computing infrastructure. It has been widely investigated in the literature under different assumptions and optimization goals [26]. In particular, many algorithms and mechanisms are developed to allocate (i.e., schedule) operators of a job into physical resources (e.g., compute node) in order to achieve certain optimization goal, such as maximizing throughput, minimizing latency or minimizing resource consumption, etc. [9]. Aniello et al. [9] propose two schedulers for Storm. The first scheduler is used in an offline manner prior to executing



(a) Server A (glue-less) (b) Server B (glue-assist)

Figure 2: Interconnect topology for our servers. Red lines represent QPI, black lines represent vendor custom data interconnects. Blue rectangles are CPU socket, and XNC is the node controller assisting as the glue.

Table 1: Characteristics of the two NUMA servers we use.

Machine / Statistic	HUAWEI KunLun Mission Critical Servers (Server A)	HP ProLiant DL980 G7 (Server B)
Processor	8x36-core Intel Xeon E7-8890 v3 at 2.50 GHz	8x16-core Intel Xeon E7-2860 at 2.27 GHz
Memory per socket	1 TB	256 GB
Local Latency (LLC)	50 ns	50 ns
1 hop latency	307.7 ns	185.2 ns
Max hops latency	548.0 ns	349.6 ns
Local B/W	54.3 GB/s	24.2 GB/s
1 hop B/W	13.2 GB/s	10.6 GB/s
Max hops B/W	5.8 GB/s	10.8 GB/s
Total local B/W	434.4 GB/s	193.6 GB/s

the topology and the second scheduler is used in an online fashion to reschedule after a topology has been running for a duration. Similarly, T-Storm [38] dynamically assigns/reassigns operators according to run-time statistics in order to minimize inter-node and inter-process traffic while ensuring load balance. R-Storm [33], on the other hand, focuses on resource awareness operator placement, which tries to improve the performance of Storm by assigning operators according to their resource demand and the resource availability of computing nodes. Commonly, their algorithms tend to treat minimizing cross-node communication as the optimization goal. Such heuristic-based approach is no longer suitable to apply in our cache-coherent NUMA environment where communication is not the only performance bottleneck because of the significantly improved performance thanks to shared-memory optimization designs.

Operator scaling: Research efforts have focused on scaling the replication level of application operator in response to workload changes. The scaling action is triggered according to some threshold-based policies based on system utilizations [15, 18]. Recently, Cardellini et al. [14] propose a general formulation of the optimal DSP placement. Later, Cardellini et al. [13] extend their original proposal by considering optimizing replication and placement jointly at the same time (similar to ours). However, applying their solution in NUMA architecture results in suboptimal due to the overlook of the changes in operator execution behaviours (e.g., resource consumption) with different execution plans, which is our main concern.

DSP systems: Data stream processing (DSP) systems have attracted a great amount of research effort. A number of systems have been developed, for example, TelegraphCQ [16], Borealis [7], IBM System S [23] and the more recent ones including Storm [2], Flink [1] and Heron [25]. However, most of them targeted at the distributed environment, and little attention has been paid to the research on DSP systems on the modern multi-core machine. A recent patch on Flink [4] tries to make Flink a NUMA-aware DSP system. The basic idea is to treat each NUMA node a distributed compute node and place application operators as if it is in a distributed environment. However, due to lack a NUMA-aware performance model, the current heuristic based round-robin allocation strategy fails to provide the optimal performance. Closely related to this study, Zhang et al. [40] revisited three common design aspects of modern DSP systems on modern multi-socket multi-core architectures and proposes initial attempts to address the identified performance bottlenecks in existing DSP systems, which however lacks of performance modeling to guide the optimization process. SABER [24] focuses on efficiently realizing computing power from both CPU and GPUs. Streambox [30] provides an efficient mechanism to handle out-of-order arrival event processing in multi-core environment. Those solutions are complementary to ours and our focus is building a NUMA-aware DSP system.

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 [36, 39]. There have been studies on optimizing the instruction cache performance [41, 20], the memory and cache performance [8, 21, 12, 11] and NUMA [28, 29, 19]. The work most closely related to our RLAS paradigm is presented by Giceva et al. [19]. They describe a model called resource activity vectors that characterize the resource footprint of individual database operators. Based on this model, they derive a query execution plan considering both operator collapsing and placement on NUMA system that minimizes the total resource demands of a query plan. In contrast, the pipeline execution model of DSP systems requires different cost models and system designs.

3. ARCHITECTURE

In this section, we show the overall architectural design of BriskStream, which is specifically optimized for the shared-memory environment.

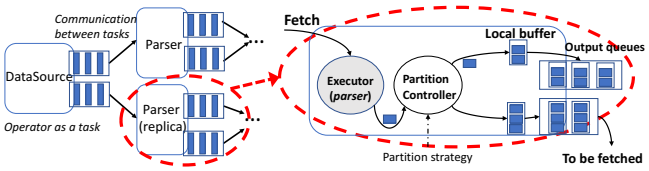


Figure 3: Tasks overview of an application.

BriskStream adopts the commonly used pipelined and operator replication designs inspired by modern DSP systems (Section 2). Figure 3 presents the tasks overview of an application of BriskStream. In particular, each operator (or the replica) is mapped to one *task*, and an edge indicates

communication between tasks. The task is the fundamental processing unit (i.e., executed by a Java thread) within BriskStream, which consists of an executor and a partition controller. The core logic for each *executor* is provided by the corresponding operator of the application. Executor operates by taking a tuple from the output queues of its producers and invoke the core logic on the obtained input tuple. After the function execution finished, it dispatches zero or more tuples by sending them to its *partition controller*. The partition controller decides in which output queue a tuple should be enqueued according to application specified partition strategies such as shuffle partition.

BriskStream applies the following two designs to optimize its execution to better utilize cache and shared memory.

Point-to-point communication channel. To avoid any centralized contention, BriskStream uses point-to-point communication channels structure. Specifically, there is a communication channel between each pair of producer and consumer in BriskStream. In this way, we restrict the potential contention to between only each pair of two tasks with a payoff of potentially higher memory consumption.

We use a size-bounded cache optimized variant of single-producer-single-consumer lock-free queue [27] to implement the communication channel. We use backpressure to handle the buffer overflow so that there is no tuple lost or re-transmission required. Due to the cache coherency among CPU sockets, message passing among operators can be done in a shared-memory way without involving (de)serialization. Especially, only the reference to the output data is passed downstream (i.e., enqueued into the queue).

It is noteworthy that as an operator may have multiple producers, it sequentially scan through output queue of each producer. If it fails to obtain input tuple (e.g., queue is empty) from one producer, it immediately tries to fetch from others. Once it successfully obtains an input tuple, it invokes the core logic, correspondingly.

Local output caching. Task maintains output buffers for each of its consumers. Specifically, the output tuple is first enqueued into a local buffer, which will then be emitted once it is filled up to the threshold size.

The benefits are two-fold. First, although the queue is lock-free, accessing to it may involve cache coherency overhead (e.g., two tasks are allocated into different CPU sockets). Locally buffering multiple tuples before emitting to the queue essentially reduces the frequency of accessing the queue. Second, the consumer can hence execute on multiple tuples (of the same buffer) for each fetch, which significantly improves instruction locality [40]. However, such lazy evaluation behaviour potentially increases process latency. We set the target buffer size to a reasonably large number (10) in our experimental evaluations, and our experimental results show that the end-to-end processing latency of BriskStream is, in fact, significantly smaller (i.e., better) than two existing DSP systems.

4. MOTIVATION AND OPTIMIZATION OVERVIEW

In this section, we first present the motivation for developing the replication and location aware scheduling (RLAS) paradigm. Next, we present the optimization overview of RLAS in high level. The performance model and algorithm implementation details are presented in Section 5 and Section 6, respectively.

4.1 Motivation Study

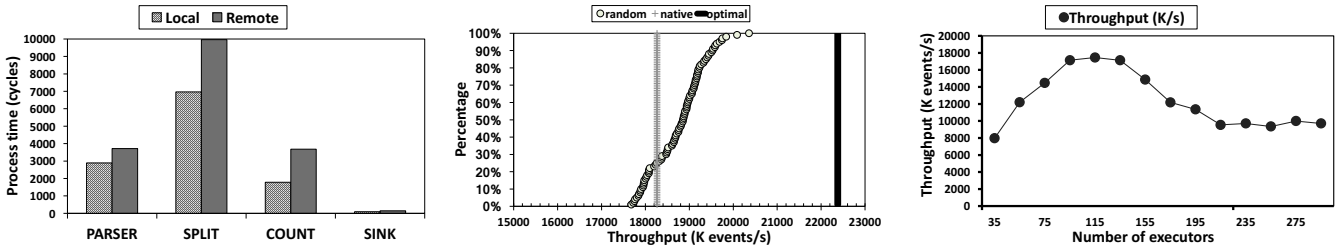
In the following, we run *word count* (the example application) on Server B as an example to motivate our proposal. In each test, we let the system to have a warm-up phase in order to make sure the JVM compilation optimization is completed. All the experiments are repeated three times and the variance is insignificant. Our design of replication and location aware scheduling is motivated by the following observations.

Observation 1: Operator experiences significantly different processing cost with different relative location to its producer. Figure 4a demonstrates the effect of varying different relative location by fixing the replication level of each operator to be one (i.e., no operator replication). We use "local" to denote the case if one operator is allocated to the same socket of its producer, and "remote" to denote if it is 1-hop away (e.g., one is in socket 0 and the other is in socket 1). In comparison with locally allocated case, the average processing time of an operator is up to 42% higher when it is remotely allocated to its producer. The performance of "Sink" does not vary as expected because it does not access the actual content of the tuple (i.e., it simply counts the number of tuples received).

Observation 2: Randomized approaches without performance modelling hardly produce good execution plan. We set the replication level of DataSource and sink to be 5 and 10, respectively, and others to be 30. Figure 4b shows the cumulative density function (CDF) of throughput of WC with 100 random placements (denoted as random). We also add two solid vertical lines, which represent the performance of BriskStream with (denoted as optimal) or without (denoted as native) placement optimization. Obviously, randomized approach hardly results in optimal placement plan, and can produce plans perform even worse than native. Therefore, a performance model is required in optimizing the placement.

Observation 3: Operator replication level can severely affect the performance of the application, and a higher replication level does not necessaryly improve performance, but can worse it. We vary the total replication level and assign same replication level for parser, splitter and counter operator. The replication level of spout and sink are fixed to be 5 and 10, respectively. Figure 4c shows the change of performance with varying replication configurations. Clearly, the performance degrades after a certain point. This is due to the increasing resource contention including cache, memory, remote memory access (RMA) bandwidth, etc., among increasing number of executors. In order to get good performance, one has to set a proper replication configuration. It is noteworthy that, guided by our modelling on performance and resource constraints, BriskStream is able to determine the optimal replication level of each individual operator, and our algorithm tries to make sure the execution plan satisfy resource constraints.

Those observations challenge existing related performance model [37] and optimization approaches [38, 33, 9, 26] as they fail to capture those features. As we will show later in experiments (Section 7) that, commonly applied minimizing traffic heuristic-based approach often results in *not-able-to-allocate* phenomenon, and thus the resource constraints have to be relaxed, resulting in over-subscription of CPU sockets, which performs even worse than OS dynamic scheduling.



(a) Processing time under varying relative location.

(b) Comparing randomized placements to native and optimal placement.

(c) Throughput under varying replication configurations.

Figure 4: Motivation study for replication and location aware scheduling.

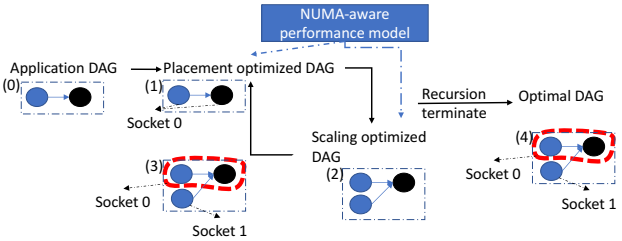


Figure 5: Scheduling overview of BriskStream.

On the other hand, we still have to carefully arrange the placement of operators to CPU sockets in order to get the best performance. As a result, existing heuristic-based approaches (e.g., [38, 9, 26]) commonly used in distributed environment settings falls short in our context.

4.2 Optimization Overview

We study the *Replication and Location Aware Scheduling* (RLAS) problem in stream processing on modern scale-up server with NUMA architecture. Specifically, we are to determine the optimal replication level as well as the optimal placement (i.e., CPU affinity) of each operator at the same time guided by our performance modelling.

Figure 5 shows the scheduling overview of BriskStream with a toy application involving two operators (marked as 0). BriskStream first determines the optimal placement of each operator (marked as 1), which also identifies bottleneck operators. The bottleneck operator is the operator which has a greater input rate than its processing rate (see Section 5). Then, it tries to increase the replication level of the bottleneck operators (marked as 2). BriskStream recursively optimizes its placement and operator replication level (marked as 3). Finally, the application DAG with optimal replication and placement configuration (marked as 4) is submitted to execution engine, where tasks are created, which hosted operators to perform the actual execution as discussed in Section 3.

5. PERFORMANCE MODEL

In the following, we discuss our NUMA-aware performance model that guides the optimization. Our model is inspired by rate-based optimization framework [37], but the original rate-based optimization framework assumes processing capability of an operator is predefined and independent of different execution plans. Moreover, it is neither replication-aware nor NUMA-aware, which are both necessary for our problem.

Table 2: Summary of main terminology

Notation	Definitions	Sources
ε	Cycles available per unit of time in one CPU core	platform input
κ	Number of CPU cores per socket of the system	
$\eta_{i,j}$	Data transfer latency from socket i to socket j, $\eta_{i,i}=0$	
$S_i, i, \in 1...n$	Set of CPU sockets of the system	
χ	External input stream rate of a stream application	application input
O_j, $j \in 1...m$	Set of operators of a stream application	
$\sigma^{in,br,out}$	Input, branch, and output selectivity of an operator	
$\|tuple\|$	average size of each input tuple of an operator	profiling input
$\xi^{pro(fetch)}$	Cycles spend in function execution (data fetch) per input tuple	
LLC_{miss}	number of LLC miss during function execution per input tuple	
$\|O_j\|$	parallelism of each operator	optimizer input
L_j, $j \in 1...m$	location of each operator	
$\xi_{s,c}$	Total cycle required for handling each input tuples from O_s	model output
$\lambda_{s,c}$	Input rate of O_c from O_s	
$\mu_{s,c}$	Processing rate of O_c for handling tuples from O_s.	
r_j	Output rate of O_j	
Ψ	Estimated throughput of the stream application	

Model overview. In this paper, application throughput is the metric to maximized by optimizations. The throughput (Ψ) of the stream application is modelled as the summation of the number of outputs produced in a unit of time (i.e., $r_{operator}$) from all "sink" operators (i.e., operators with no consumer). That is,

$$\Psi = \sum_{sink} r_{sink} \quad (1)$$

Subsequently, in order to predict r_{sink}, i.e., the output rate of each "sink" operator, we need to predict how many tuples it can process in a unit of time (i.e., processing rate) and how many tuples it is expected to receive in a unit of time (i.e., input rate) under an execution plan. We summarize the main terminologies used in Table 2.

5.1 Input Rate Estimation

Definition 1. **Input rate** $\lambda_{s,c}$ is defined to be the number of tuples received at operator O_c in a unit of time, which are produced by operator O_s.

Let input selectivity of O_c be σ^{in} denoting the actual portion of stream it receives from its producer O_s, then

$$\lambda_{s,c} = \begin{cases} \chi \text{ if s} \in \emptyset \\ \sigma^{in} * r_s, \text{otherwise} \end{cases} \quad (2)$$

where r_s denotes output rate of O_s and χ denotes external input stream rate of an application.

Parallel Operators. We consider each operator replica individually as an operator in our model, and we only need

to extend the input estimation accordingly. In particular, r_s is being partitioned.

Let $p(s, c_e)$ denote the **partition ratio** from O_s as producer to O_{c_e} as consumer. Then, Equation 2 shall be revised as follows,

$$\lambda_{s,c_e} = \begin{cases} \chi, \text{if s} \in \emptyset \\ \sigma^{in} * r_s * p(s, c_e), \text{otherwise} \end{cases}$$
$$where \sum_{e \in \ replicas \ of \ O_c} p(s, c_e) = 1 \quad (3)$$

The **total input rates** λ_c denotes the total number of tuples to be received from all of the producers of O_c in a unit of time, and can be hence denoted as $\sum_{s \in \vec{Prod}(c)} \lambda_{s,c}$, where $\vec{Prod}(c)$ represents a set producers of operator O_c.

5.2 Processing Rate Estimation

Estimating the actual processing rate of each operator is important as it not only provides necessary information to estimate application throughput, but also the resource demands. However, it is a challenging task due to the following reasons.

1. Due to NUMA effect, the processing rate of each operator varies depending on its placement that results in different data fetch cost.
2. Operator may cooperatively process tuples from different producers that correspond to different processing cost.
3. Input rate may or may not affect the actual processing rate depending on their relative size.

All the above factors need to be considered for building an accurate cost model for DSP systems, which is quite different to previous studies [37]. In order to capture those factors, we first define the basic processing rate that represents the processing capability of an operator assuming it has only one producer. We breakdown the time of processing each tuple into two components (i.e., $\xi^{fetch}_{s,c}$ and $\xi^{pro}_{s,c}$) and take the varying remote memory access cost (C_{RMA}) into consideration so that the basic processing rate of an operator is aware of the relative placements. Then, we show how to estimate total processing rate that represents the processing capability of an operator considering multiple producers. The total processing rate is used to estimate the cooperative processing rate that represents the processing rate of an operator to a producer when it needs to cooperatively process tuples from multiple producers. Finally, we consider both cases of over and under input supply when estimating the actual processing rate.

Definition 2. **Basic processing rate** $\mu_{s,c}$ is the number of tuples (from O_s) that can be processed by O_c in a unit of time. It represents the conceptual processing capability of O_c to handle tuples from O_s without processing tuples from other producers.

Let $\xi_{s,c}$ to denote the **demand cycles per tuple** required by operator O_c on handling one tuple from operator O_s. As an operator is carried by one thread, it can maximally fully occupies one CPU core. That is,

$$\mu_{s,c} = \frac{\epsilon}{\xi_{s,c}} \quad (4)$$

, where ϵ represents the cycles available per unit of time in one CPU core.

We breakdown $\xi_{s,c}$ into the following non-overlapping components.

1. $\xi^{fetch}_{s,c}$: Cycles required to fetch (local or remotely) the actual data of each input tuple from operator O_s.
2. $\xi^{pro}_{s,c}$: Cycles required in actual function execution and emitting outputs tuples per input tuple.

The $\xi^{pro}_{s,c}$ will not vary in different execution plans and can be obtained through profiling. In contrast, due to NUMA, the data fetch cycles $\xi^{fetch}_{s,c}$ is determined by its relative distance to its producer, which can be represented as,

$$\xi^{fetch}_{s,c} = C_{LocalAcc} + C_{RMA} + c \quad (5)$$

, where c represents a constant cost mainly consists of accessing the queue data structure. It can be significantly reduced by applying local output caching (Section 3). $C_{LocalAcc}$ refers to the cost of local data access (i.e., within the same CPU socket), and we find it is mostly severed at the last level cache (LLC). Therefore, $C_{LocalAcc} = \frac{|tuple|}{|cacheLine|} * T_C$, where T_C refers to the access cost of last level cache. On the other hand, C_{RMA} refers to data transfer cost and can be estimated as $C_{RMA} = \frac{|tuple|}{|cacheLine} * \eta_{i,j}$, where $\eta_{i,j}$ represent the data transfer cost from socket i (allocation of producer) to socket j (allocation of consumer), which varies in different execution plans.

Let σ^{br} be the branching ratio that determines the actual portion of input tuples being processed (otherwise being filtered), we have

$$\xi_{s,c} = \xi^{fetch}_{s,c} + \sigma^{br} * \xi^{pro}_{s,c} \quad (6)$$

Definition 3. **Total processing rate** μ_c of an operator O_c is defined to be the total number of tuples (from all of its producers) that can be processed in a unit of time.

Let's assume an arbitrary length of observation time t, and denote number of tuples to be processed during t as *num* and time needed to process them as *time*. Then, we have

$$\mu_c = \frac{num}{time},$$
$$where \ num = \lambda_{s,c} * t,$$
$$time = \sum_{s \in \vec{Prod}(c)} \frac{\lambda_{s,c}}{\mu_{s,c}} * t. \quad (7)$$

Definition 4. **Cooperative processing rate** $\widetilde{\mu}_{s,c}$ of an operator O_c is defined to be the *upper bound* of the number of tuples able to be handled from O_s in a unit of time in the presence of tuples from other producers.

As tuples from all producers are processed in a cooperative manner with equal priority [1], tuples will be get processed in a first come first serve manner. Therefore, given the total processing rate μ_c, the $\widetilde{\mu}_{s,c}$ is proportional to the proportion of the corresponding input rate, that is,

$$\widetilde{\mu}_{s,c} = \mu_c * \frac{\lambda_{s,c}}{\lambda_c} \quad (8)$$

Finally, in order to estimate the **actual processing rate** of an operator ($\overline{\mu}_{s,c}$), there are two cases that we have to consider:

[1]It is possible to configure different priorities among different operators here, but is out of the scope of this paper.

1. $\widetilde{\mu}_{s,c}$ is large enough such that it can finish processing all its given inputs even with the need of cooperatively processing tuples from other producers. In other words, it is under-supplied, and its processing rate is limited by its input rates. In this case, $\overline{\mu}_{s,c} = \lambda_{s,c}$.
2. On the other hand, it may be over-supplied, and the expected rate is simply the cooperative processing rate, and $\overline{\mu}_{s,c} = \widetilde{\mu}_{s,c}$.

Therefore, $\overline{\mu}_{s,c} = min(\widetilde{\mu}_{s,c}, \lambda_{s,c})$.

5.3 Output Rate Estimation

Definition 5. **Output rate** $r_{s,c}$ of an operator O_c is defined to be the expected number of tuples produced in a unit of time in handling tuples from O_s.

Let the selectivity of O_c be σ^{out}, then $r_{s,c} = \overline{\mu}_{s,c} * \sigma^{out}$, and the **total output rate** r_c can be hence expressed as $r_c = \sum_{s \in \vec{Prod}(c)} r_{s,c}$.

Example 5.1. Consider a simple application with three operators A, A' (replica of A) and B as shown in Figure 6 as an example, and we are to estimate the output rate of B. Assume under such an execution plan that we have $\lambda_{a,b} = 15$ and $\lambda_{a',b} = 10$. Further, we are given the processing rate as $\mu_{a,b} = 10$ and $\mu_{a',b} = 15$ (e.g., B is collocated with A' but remotely allocated to A), respectively. The total processing rate μ_b can be estimated as, $\mu_b = \frac{\lambda_{a,b}+\lambda_{a',b}}{\frac{\lambda_{a,b}}{\mu_{a',b}}+\frac{\lambda_{a,b}}{\mu_{a',b}}} = 11.5$. The cooperative processing rate is therefore, $\widetilde{\mu}_{a,b} = \mu_b * \frac{\lambda_{a,b}}{\lambda_{a,b}+\lambda_{a',b}} = 6.9$, and $\widetilde{\mu}_{a',b} = \mu_b * \frac{\lambda_{a',b}}{\lambda_{a,b}+\lambda_{a',b}} = 4.6$.

As $\widetilde{\mu}_{a,b} < \lambda_{a,b}$ and $\widetilde{\mu}_{a',b} < \lambda_{a',b}$, the expected processing rates are both bounded by cooperative processing rates, and the aggregated output is therefore the sum of cooperative processing rate (i.e., 11.5).

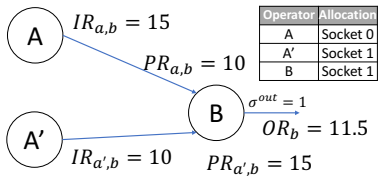


Figure 6: Example of estimating output rate.

We make from this example two observations. First, operator may experience different processing rate to different producers due to different relative allocations. Second, total processing rate shifts close to the processing rate of the corresponding producer who offers relatively large input rates, i.e., operator needs to process more tuples from that producer.

6. OPTIMIZATION ALGORITHMS

In this section, we first analyze the resource constraints that the algorithm needs to be aware of (Section 6.1). Then, we discuss the complexity of the "naive" searching algorithm for the optimal execution plan (Section 6.2), followed by the improved algorithms including placement algorithm (Section 6.3) to determine the optimal operator placement assuming operator replication level is fixed and scaling algorithm (Section 6.4) that work with the placement algorithm by considering varying operator replication to finally determine the optimal execution plans.

6.1 Resource Constraints

We consider three kinds of resource constraints and the optimization algorithm tries to make sure the execution plans satisfy those constraints. Task oversubscribing has been studied in some earlier work [22], but it is not the focus of this paper.

- *CPU capacity.* Given the available CPU cycles in a CPU socket S_i as $\epsilon * \kappa$ per unit of time, where ϵ is the cycles available of single CPU core, and κ is the number of CPU cores on a socket. The aggregated demand of CPU cycle per unit of time requested from operators, which are scheduled to CPU socket S_i must be smaller than $\epsilon * \kappa$, i.e.,

$$\sum_{c=1}^{m} \sum_{s \in \vec{Prod}(c)} p_{c,i} * \overline{\mu}_{s,c} * \xi_{s,c} \leq \epsilon * \kappa$$
$$where\ p_{c,i} \in 0, 1, \forall c \in 1, .., m, \forall i \in 1, .., n$$
$$p_{c,i} = 1 \text{ if } O_c \text{ is allocated into } S_i \quad (9)$$

- *Local memory bandwidth capacity.* Given the maximum attainable local DRAM bandwidth as B_i per unit of time on socket S_i. The aggregated amount of bandwidth requested to the same CPU socket S_i per unit of time must be smaller than B_i, i.e.,

$$\sum_{c=1}^{m} \sum_{s \in \vec{Prod}(c)} p_{c,i} * \overline{\mu}_{s,c} * |M| \leq B_i$$
$$where\ |M| = |Miss_{LLC}| * |cacheLine|$$
$$p_{c,i} \in 0, 1, \forall c \in 1, .., m, \forall i \in 1, .., n$$
$$p_{c,i} = 1 \text{ if } O_c \text{ is allocated into } S_i \quad (10)$$

- *Remote channel bandwidth capacity.* Given the maximum attainable remote channel bandwidth from socket S_i to S_j as $Q_{i,j}$ bytes per unit of time. The aggregated data transfer from socket S_i to socket S_j per unit of time must be smaller than $Q_{i,j}$, i.e.,

$$\sum_{c=1}^{m} \sum_{s \in \vec{Prod}(c)} p_{c,j} * p_{s,i} * r_{s,c} * |tuple| \leq Q_{i,j}$$
$$where\ p_{k,i} \in 0, 1, \forall k \in 1, .., m, , \forall i, j \in 1, .., n$$
$$p_{k,i} = 1 \text{ if } O_k \text{ is allocated into } S_i \quad (11)$$

6.2 Complexity Analysis

Naively, RLAS concerns the scheduling decisions (i.e., $\binom{m}{1}$, where m is the number of CPU sockets) for all n operators. Furthermore, as we need to consider different replication configurations for each operator, and assume replication of each operator can be set from 1 to K, we have to consider in total K^n different replication configurations. As a result, the solution space of the presented problem has a size of $O(m^n * K^n)$, which makes brute-force unpractical for large DAGs (i.e., n is large).

6.3 Placement Algorithm

Due to resource constraints, the placement decisions are sometimes conflicting each other and ordering constraint is introduced into the problem. For instance, scheduling of an operator into a socket may prohibit some other operators to be scheduled to the same socket. This makes the sub-problem dependent on each other and classical approaches

like dynamic programming not suitable. On the other hand, Branch and Bound (B&B) are successfully applied in many works [31] in dueling with non-convex combinatorial optimization problems. In this section, we discuss our B&B based algorithm to solve the RLAS problem assuming operator replication is given and fixed.

Algorithm overview. Branch and Bound (B&B) systematically enumerates a tree of candidate solutions, based on a bounding function. We first define three types of nodes in the tree: the solution nodes, the live nodes, and the dead nodes.

Live nodes: the live node contains the solution that violates some constraints and they can be expanded into other nodes that violate fewer constraints.

Dead node: once expanded, a live node is turned into a dead node, and we do not have to remember it anymore.

Solution node: solution node contains a scheduling plan without violating any constraint. The value of a solution node comes directly from the performance model. The algorithm may reach multiple solution nodes as it explores the solution space. The solution node with the best value is the output of the algorithm.

Before presenting our algorithm based on B&B, we need to first determine three main components of this algorithm, 1) the bounding function so that the algorithm can efficiently estimate the bounded cost of the live solution, 2) a strategy for selecting the live solution to be investigated in the current iteration, and 3) a branching rule to be applied if the live solution needs to be further explored.

The bounding function. If the bounding function value of an intermediate node is worse than the best feasible solution obtained so far, we can safely prune it and all of its children nodes. This does not affect the optimality of the algorithm because the value of a live node must be better than all its children node after further exploration. In other words, the value of a live node is the theoretical upper bound of the subtree of nodes.

The bounded problem that we used in our optimizer originates from the same optimization problem with relaxed constraints that every non-scheduled operator can be scheduled in order to maximize its process rate. Specifically, the value (objective function) of every intermediate solution is obtained by fixing the allocation of *valid* operators and maximizing processing rates of every non-scheduled ones.

Example 6.1. Consider the same application in previous example with operators A, A' and B as shown in Figure 7. Assume at one iteration, A and A' are scheduled to socket 0 and 1, respectively (i.e., they become valid). We are to calculate the bounding function value assuming B is the sink operator, which awaits to be allocated.

In order to calculate the bounding function value, we can simply try to maximize process rates of B. In this case, we set B to be collocated with A and A' at the same time (which is certainly invalid), and its out rate is maximized, which is the bounding value of the current iteration.

The searching strategy. We adopt a combination strategy of depth first search (DFS) and best first search (BeFS). In particular, the basic searching strategy follows DFS, but we always try first the node with a better bounded value between nodes at the same level of the search tree. This heuristic will not reduce the solution space, but instead, gives a greedy direction on which intermediate node to visit first. The worst case is that all solutions need to be explored. Nevertheless, our experimental results show that this searching heuristic helps in converging towards the best solution faster.

Algorithm 1 Branch and Bound based Placement

```
Data: Stake: stack;                                   ▷ contains all live nodes
Data: Node: solution;
Data: Node: e;                                        ▷ the current visiting node
Results: Scheduling plan: p;
solution.value ← 0                                    ▷ Start with no initial solution
/*Initialize the root node*/
e.decisions ← a list contains all possible collocation decisions;
e.plan ← all operators are collocated into the same CPU socket;
e.validOperators ← 0;
Push(stack,e);
/*Branch and Bound process*/
while ¬IsEmpty(stack) do
    e ← Pop(stack);
    if e.validOperators = totalOperators then
        if e.value ≥ solution.value then
            solution ← e;
        end if
    else if e.value > solution.value then
        Branching(e);
    end if
end while
return solution;
Function – Branching(Node e)
Data: Node[] children ← ChildrenOf(e);
for each Node c ∈ children do
    c.validOperators ← e.validOperators + #newAllocate;
    c.value ← CostFunction(c);
end for
SortByValue(children);                                ▷ for best first search
PushAll(stack,children);
Function – ChildrenOf(Node e)
Data: Node[] children;
for each pair of O_s and O_c in e.decisions do
    if all predecessors are determined except O_s to O_c then
As the process rate can be determined in this case, we apply best fit first strategy for each case.*/
        create nodes of 1): collocating them and 2): separately allocate them;
        add both nodes to children;
    else
        for each way of scheduling O_s and O_c do
            add one node to children;
        end for
    end if
end for
return children;
```

The branching rule. Naively in each iteration, there are $\binom{n}{1} * \binom{m}{1} = n * m$ possible solutions to branch (i.e., schedule which operator to which socket) and an average n depth (i.e., one operator is allocated in each iteration). In other words, it will still need to examine on *average* $(n * m)^n$ candidate solutions [17]. In order to further reduce the complexity of the problem, heuristics have to be applied. The branching rule determines the generation of children nodes of each live node if it needs to be further explored. We

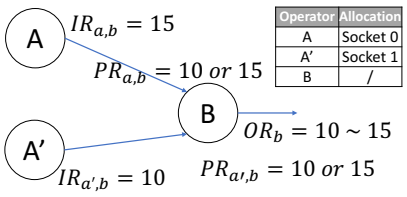


Figure 7: Estimating bounding value.

introduce two heuristics that work together to significantly reduce the solution space as follows.

Collocation heuristic: Naively, we need to consider branching to all scheduling decisions of each operator in each iteration. However, we observe that some scheduling decisions have *no or little* impact on the process rate of any operators. Therefore, we propose to consider a list of collocation decisions instead of scheduling decisions of each operator. During process, we can remove those collocation decisions from list that are no longer relevant. For instance, it can be safely discarded (i.e., do not need to consider anymore) if both producer and consumer in the collocation decision are already allocated.

Best-fit heuristic: Consider each collocation decision involving a pair of producer and consumer, when all predecessors (i.e., upstream operators) of them are already scheduled, the process rate of them can be determined. In this case, we select the best-possible plan among them with *best fit* strategy. Specifically, if two operators can be collocated into different sockets, we only consider the way of collocating them into one socket that will leave with the smallest idle resource.

Algorithm implementation. Algorithm 1 illustrates the *Branch and Bound based Placement* algorithm. Initially, no valid solution has been found so far and the best objective value is 0 (i.e., zero output rate) [2]. The algorithm proceeds with this and it updates the best solution whenever a feasible solution is found. The searching process dynamically generates a search tree, which initially only contains the root. The root contains the scheduling plan of all operators being collocated and has the upper bound value of the whole solution space. During iteration, scheduling decisions are made which update the list of children node (i.e., branches) by removing those decisions involving producer and consumer have been already allocated. The unexplored subspaces are represented as nodes and each iteration processes one such node.

Example 6.2. As a concrete example to illustrate the algorithm, consider scheduling the same application in previous example with an additional operator C as a consumer of B to a machine with two CPU sockets as shown in the top-left of Figure 8. Both operator B and C are sink operators. In this example, we assume the aggregated resource demands of three operators with either A, B and C or A', B and C exceed resource constraint of a socket, and they cannot be collocated in the same socket due to violation of resource constraints. We also show one feasible solution besides, which is assumed to be the optimal allocation in this example. The bottom left of Figure 8 shows how our algorithm explores the searching space by expanding nodes, where the label on the edge represents the collocation decision considered in the current iteration.

The detailed states of four nodes are illustrated on the right hand side of the figure, where the state of each node is represented by a two-dimensional matrix. The first (horizontal) dimension describes a list of collocation decisions, while the second one the operators that interests in this decision. A value of '-' means that the respective operator is not interested in this collocation decision. A value of '1' means that the collocation decision is made

[2]BriskStream applies a fast greedy approach to obtain a valid solution first to speed up the searching process.

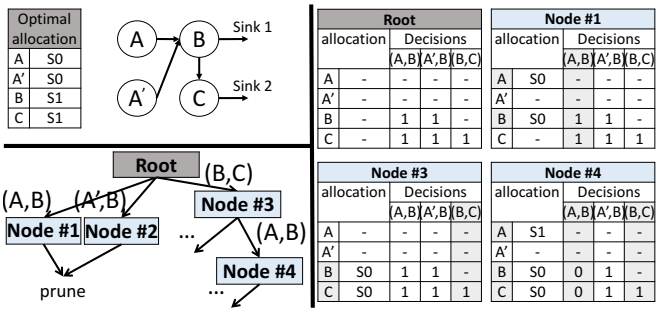


Figure 8: Placement Algorithm at Runtime.

in this node, although it may violate resource constraints. A value of '0' means that the collocation decision is not satisfied and the involved producer and consumer are separately located.

At root node, we consider a list of scheduling decisions involving each pair of producer and consumer. At Node #1, the collocation decision of A and B is going to be satisfied (i.e., all sockets are empty now), and assume they are collocated to socket 0 (S0). However, the solution to the bounding function of this node indicates that while we try to maximize output rates of every pipeline, B and C has to be allocated separately because of the assumed resource constraints in this example. As a result, its bounding value is worse than the known feasible solution, and we can stop further exploring this node.

On the other hand, consider Node #3, we first try to collocate B and C, which is also satisfied, and similarly assume they are collocated to socket 0 (S0). In this simple example, the solution to the bounding function of this node is identical to the feasible solution, and we need to further explore it.

Finally, we should notice the importance of the heuristics. Without the collocation heuristic, root node would generate $2 * 4$ (instead of 3) different child nodes, where each child node contains a plan of allocating one of the four operators to one of the two socket.

6.4 Scaling Algorithm

An application has different optimal placement plans under different replication configurations. Therefore, determining the optimal replication configuration shall be done together with the placement optimization.

Algorithm 2 illustrates our *iterative scaling* algorithm. Initially, we set replication of each operator to be one (Line 1∼2). The algorithm proceeds with this and it determines the optimal placement based on its current replication setting (Line 6). Then, it stores the current setting if it ends up with better performance (Line 7∼9). Subsequently, it increases replication level of the identified bottleneck operator (i.e., this is identified during placement optimization), and determines the optimal placement again (Line 13∼ 23). At Line 13∼14, we iterate over all the sorted list from topologically sorting on the DAG in parallel. This ensures that we have gone through all the way of scaling the topology bottlenecks and hence ensures the optimality of our scaling algorithm. We set an upper limit on the total replication level (e.g., usually set to the total number of CPU cores) to stop the recursion. At Line 10&20, either the algorithm hits the scaling upper bound or fails to find a

Algorithm 2 Topologically sorted iterative scaling

```
Data: Execution Plan: c;                          ▷ the current visiting plan
Data: Execution Plan: opt;                        ▷ the optimal plan
c.parallelism ← set parallelism of all operators to be 1;
c.graph ← creates execution graph according to c.parallelism;
opt.throughput ← 0;
Searching(c);
return opt;
    Function – Searching(c)
c.placement ← optimal placement of c;
if c.throughput > opt.throughput then
|   opt ← c;
else
|   if failed to find feasible placement plan then return ;
|   end if
end if
sorted-lists ← Topological_sort (c.graph)
for each list:sorted-lists do
|   for each Executor e ∈ list do
|   |   if e is bottleneck then
|   |   |   c ← Increase the parallelism of e by 1;
|   |   end if
|   end for
|   if failed to further increase the parallelism then return ;
|   else
|   |   Searching(c);
|   end if
end for
```

feasible plan will cause the searching to terminate.

We set an upper bound of 600 seconds for running RLAS algorithm, which are usually enough to optimize the workload. Otherwise, the best feasible plan obtained is returned as the optimization result. As the streaming application potentially runs forever, the overhead of generating a plan is not included in our measurement.

7. EVALUATION

In this section, we experimentally evaluate BriskStream:

- Our designs successfully optimize BriskStream to better utilize cache and shared memory (§7.2).
- Our model can accurately predict the performance of an application under the NUMA effect (§7.3).
- Our RLAS optimization performs significantly better than competing techniques (§7.4).
- BriskStream significantly outperforms two open-sourced DSP systems (§7.5).

7.1 Experimental Setup

We pick five common applications with different characteristics to evaluate BriskStream. These tasks are word-count (WC), fraud-detection (FD), spike-detection (SD), log-processing (LG), and linear-road (LR) with increasing topology complexity and varying compute and memory bandwidth demand. We use same settings as the previous study [40]. For more details, readers can refer to the previous paper. In all cases, we set the external input rate (χ) based on a reasonable large threshold R of each application so that overflowing do not constantly happen during the testing. We compare BriskStream with two open-sourced DSP systems including Apache Storm (version 1.1.1) and Flink (version 1.3.2). For a better performance, we disabled acknowledgement in Storm and checkpoint in Apache Flink, since there is no failure in our experiments.

7.2 Evaluation on System Designs

In order to understand the effect of our shared-memory optimization designs, we show an execution time analysis of all applications comparing BriskStream and Storm in a single socket in Figure 9. The execution time is breakdown into four components, 1) *Execution* represents effective execution time (i.e., total time excluding other three components), 2) *LLC Miss* represents the time spend due to memory stall, 3) *L1 Miss* represents the time spend due to L1 cache miss, especially including instruction cache miss, and 4) *Bad speculation* refers the time spend due to branch misprediction. The two major observations are that 1) L1 Miss stalls are insignificant, which indicates the efficiency of our designs in utilizing instruction cache, which has shown to be a performance killer in existing DSP systems [40] and 2) effective execution dominates the execution time, which confirms BriskStream's efficient usage of cache and memory.

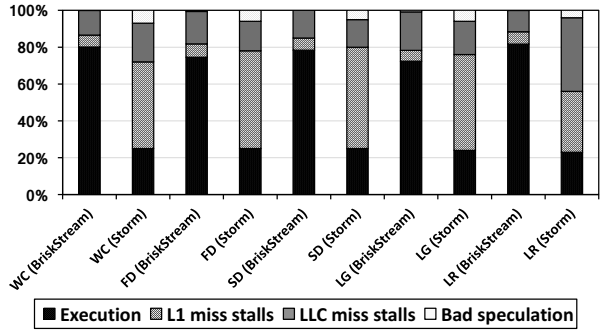


Figure 9: Execution time breakdown of all applications.

7.3 Model Evaluation

BriskStream requires a profiling stage to collect the necessary running statistics of all operators of each application. Specifically, the application is launched with the replication level of all operators set to be one, and we collect information at hardware counter level of each task (containing one operator) using the overseer library [5]. We have made a assumption here that the running statistics will not change significantly with varying replication level.

To validate the effectiveness of our performance model, we show the relative error associated with estimating the application throughput by our analytical model. The relative error is defined in Equation 12, where $\Psi_{measured}$ is the measured application throughput and $\Psi_{estimated}$ is the estimated application throughput by our performance model for the same application.

$$relative\_error = \frac{|\Psi_{measured} - \Psi_{estimated}|}{\Psi_{measured}} \quad (12)$$

The model accuracy evaluation of all applications under optimal execution plan is shown in Table 3. Overall, our model can accurately predict the performance of all five streaming applications, and thus can guide the decision of getting the suitable value. The predication error tends to be larger for low resource demand applications (e.g., FD and SD). This is because the running statistics of light weight operators are more vulnerable to thread interference, i.e., they run slower when more threads interfere each other. Instead of using constant operator statistics obtained through profiling with replication level set to be one, an immediate extension is to use a statistics function which expresses the relationships of statistics change and thread

Table 3: Model accuracy evaluation of all applications.

Server A	WC	FD	SD	LG	LR
Model	45988.2	15713.5	6561.6	2361.7	2321.0
Actual	47492.0	11835.2	5104.3	2071.8	1958.0
Relative error	0.03	0.33	0.29	0.14	0.19
Server B	**WC**	**FD**	**SD**	**LG**	**LR**
Model	23225.1	9290.0	2307.8	1387.4	2097.8
Actual	22378.8	7393.9	2092.5	1376.2	1714.4
Relative error	0.03	0.26	0.1	0.01	0.22

interference that can further improve the accuracy of our performance model.

We further present a detailed model evaluation of WC on Server B in Figure 10, which compares the estimated and measured performance with varying the total replication level under optimal placement. When the optimizer fails to find any placement plans for the given replication configuration that satisfies the resource constraints, it gradually relax the constraints in order to launch the application. We see that our prediction is close to the performance when scaling the application until resource constraints cannot be satisfied. Although the model fails to predict the performance degradation of insufficient resources, our optimization tries to satisfy resource constraints as much as possible, and avoids such performance degradation in practice.

7.4 Comparison on Different Techniques

To gain better understanding of the effects of our RLAS optimizer, we evaluate it with different operator scheduling techniques.

- *OS*: the scheduling is left to the operating system (Both our Servers use Linux-based OS).
- *RR*: operators are placed in a round-robin manner.
- *FF*: operators are first topologically sorted and then placed in a first-fit manner (start placing from DataSource).

Both RR and FF are enforced to guarantee resource constraints as much as possible, and their replication level setting is set to be the same obtained from RLAS. In case they cannot find any feasible plan, they will gradually relax resource constraints until a feasible plan is obtained. The replication level setting of OS is experimentally tuned to its best configuration.

Figure 11 shows that our RLAS optimizer generally outperform other techniques on both two Servers. FF can be view as a minimizing traffic heuristic-based approach as it greedily allocates neighbor operators (i.e., directly connected) together due to its topologically sorting step. However, it performs poorly, and we find that during its searching for optimal placements, it often falls into "not-able-to-progress" situation as it cannot allocate the current item (i.e, operator) into any of the sockets because of the violation of resource constraints. Then, it has to relax the resource constraints and repack the whole topology, which often end up with oversubscribing of a few CPU sockets. On the other hand, RR fails take remote memory communication overhead into consideration, and the resulting plans often involves unnecessary cross sockets communication. BriskStream performs generally better than OS, and the speedup is more significant on complex applications such as LR.

Figure 12 further shows the performance of WC on BriskStream with different scheduling techniques under varying number of sockets to be used from one to eight on Server B. BriskStream performs generally better than FF and RR as expected, but performs poorly compared to OS with only two sockets enabled. We find that this is because BriskStream disallows operator to be rescheduled at runtime in the current design and operators may experience unstable workload due to the changing input characteristics. As a result, resource unbalancing occurs during processing, and is more obvious when less sockets are used. In contrast, the OS dynamic (re)scheduling keeps resource balanced and gracefully reduce such issue. However, with increasing number of sockets, the dynamic scheduling involves more and more remote memory access overhead, which eventually prohibit further scaling.

7.5 Comparison on Different DSP Systems

We compare BriskStream with two open-sourced DSP systems including Apache Storm and Flink. We use Flink with NUMA-aware configuration (i.e., one task manager per CPU socket), and as a sanity check, we have also tested Flink with single task manager, which shows no significant performance changes. We always experimentally tune the replication setting of each application to its best achievable performance for both Storm and Flink.

Figure 13 shows the throughput speedup of the optimized BriskStream compared to Storm and Flink. Overall, Storm and Flink show comparable performance for most applications, and BriskStream has around 2.6 $\sim$ 30.7 and 7.8 $\sim$ 41.6 times higher throughput on two Servers, respectively. The improvement varies for the applications with different characteristics. For example, the speed-up is significant on LR, which has the most complicated topology and performs poorly without a good NUMA-aware execution plans.

Comparing two figures, we can see the performance speed-up is generally higher on Server B than Server A. The reasons are two-fold. First, both Storm and Flink scale relatively well on a single CPU socket [40], and Server A (36 cores per socket with hyper-threading) has a much more powerful single socket than Server B (16 cores per socket with hyper-threading), which makes Storm and Flink runs relatively better. Second, thanks to the node controller, Server B has a much smaller max-hop latency and BriskStream is able to better utilizes all of its resources.

In order to better understand the impact of different NUMA architectures, we show communication pattern matrices of running WC with optimal execution plan in Figure 14. Each point in the figure indicates the summation of data fetch cost (i.e., $\frac{|tuple|}{|cacheLine|} * \eta_{i,j}$) of all operators from the x-coordinate (S_i) to y-coordinate (S_j). The major observation is that the communication requests are mostly sending from one socket (S0) to other sockets in Server A, and they are, in contrast, much more uniformly distributed among different sockets in Server B. This is because the significant differences in their differences of remote memory access penalty among sockets, and confirms that our RLAS optimization is aware of the differences of varying NUMA architectures.

The end-to-end process latency of a tuple is defined as

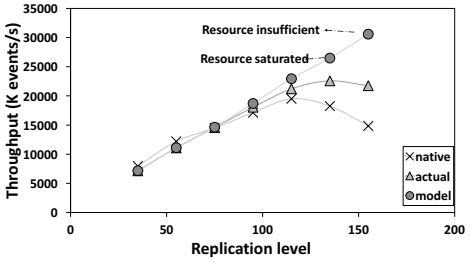
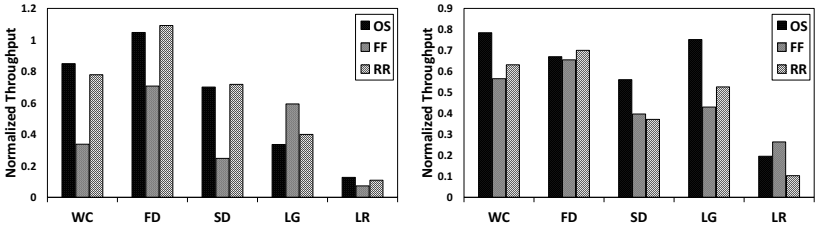


Figure 10: Model evaluation of WC as an example.



(a) Server A (b) Server B

Figure 11: Performance comparison among different optimization approaches.

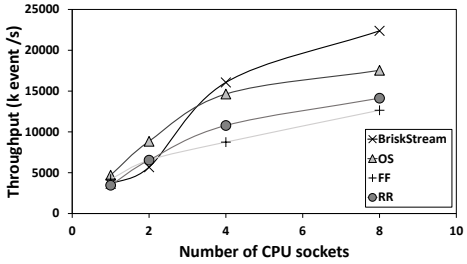
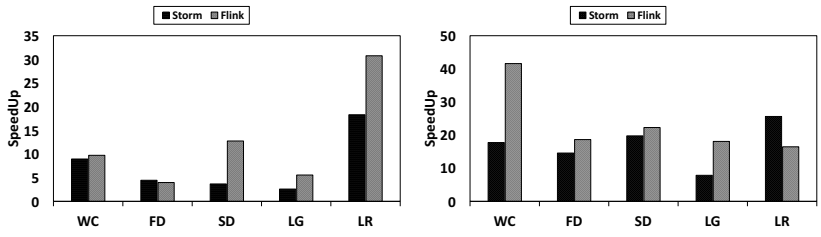


Figure 12: Scalability comparison among different optimization approaches (WC).



(a) Server A (b) Server B

Figure 13: Throughput speedup (the higher the better) compared to different DSP systems.

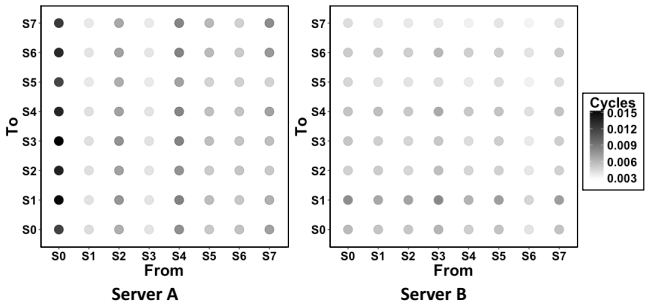


Figure 14: Communication pattern matrices of WC on two Servers. The color scale indicates communication cost in cycles per ns (darker means higher cost).

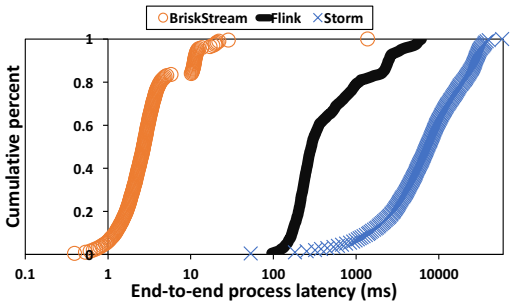


Figure 15: CDF of end-to-end process latency of WC on different DSP systems.

Table 4: Comparing 99-percentile end-to-end processing latency (ms) of different applications among different DSP systems.

	WC	FD	SD	LG	LR
BriskStream	21.9	12.5	13.5	115.1	204.8
Storm	37881.3	14949.8	12733.8	9857.8	16747.8
Flink	5689.2	261.3	350.5	1042.0	4886.2

the time duration between it's emitting from DataSource and the last descendant tuple of it being received in sink operator. This is one of the key metrics in DSP system that significantly differentiate itself to traditional batch based system such as MapReduce. We compare the end-to-end process latency among different DSP systems by using Server A as an example evaluation target. The evaluation on Server B shows similar results and hence omitted. Figure 15 shows the detailed CDF of end-to-end processing latency of WC comparing different DSP systems and Table 4 shows the overall 99-percentile end-to-end processing latency comparison of different applications. Clearly, BriskStream demonstrates significantly lower process latency, and confirms the superiority of our designs.

8. CONCLUSION

This paper proposes BriskStream, a DSP system specially designed for modern server with cache-coherent NUMA architecture. Particularly, we propose a novel NUMA-aware optimizer of DSP system, namely Replication and Location Aware Scheduling (RLAS), that is able to generate efficient execution plans considering both operator replication and placements with the NUMA effect. We compare BriskStream with five common streaming applications on Apache Strom and Flink on two modern multi-core machines. The results show that BriskStream significantly outperforms two open-sourced DSP systems usually by an order of magnitude even without tedious tuning process.

9. REFERENCES

[1] Apache flink, `https://flink.apache.org/`.

[2] Apache storm, `http://storm.apache.org/`.

[3] Intel memory latency checker, `https://software.intel.com/articles/intelr-memory-latency-checker`.

[4] Numa patch for flink, `https://issues.apache.org/jira/browse/FLINK-3163`.

[5] Overseer: Low-level hardware monitoring and management for java, `http://sosoa.inf.unisi.ch/Instruments/Overseer`.

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

[7] D. J. Abadi and et al. The Design of the Borealis Stream Processing Engine. CIDR'05.

[8] A. Ailamaki and et al. DBMSs on a Modern Processor: Where Does Time Go? VLDB'09.

[9] L. Aniello and et al. Adaptive online scheduling in storm. DEBS '13.

[10] R. Appuswamy and et al. Scale-up vs scale-out for hadoop: Time to rethink? SoCC'13.

[11] C. Balkesen and et al. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. ICDE'13.

[12] P. A. Boncz and et al. Database Architecture Optimized for the new. Bottleneck: Memory Access. VLDB'99.

[13] V. Cardellini and et al. Optimal operator replication and placement for distributed stream processing systems. *SIGMETRICS Perform. Eval. Rev. March 2017.*

[14] V. Cardellini and et al. Optimal operator placement for distributed stream processing applications. DEBS '16.

[15] V. Cardellini and et al. Elastic stateful stream processing in storm. HPCS'16.

[16] S. Chandrasekaran and et al. Telegraphcq: Continuous dataflow processing for an uncertain world. CIDR'03.

[17] L. Devroye and et al. On the complexity of branch-and bound search for random trees. *Random Struct. Algorithms'99.*

[18] B. Gedik and et al. Elastic scaling for data stream processing. *TPDS'14.*

[19] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of query plans on multicores. VLDB'14.

[20] S. Harizopoulos and A. Ailamaki. Improving Instruction Cache Performance in OLTP. *ACM Trans. Database Syst'06.*

[21] B. He and et al. Cache-conscious automata for xml filtering. ICDE'05.

[22] C. Iancu and et al. Oversubscription on multicore processors. IPDPS'10.

[23] N. Jain and et al. Design, Implementation, and Evaluation of the Linear Road Bnchmark on the Stream Processing Core. SIGMOD'06.

[24] A. Koliousis and et al. Saber: Window-based hybrid stream processing for heterogeneous architectures. SIGMOD '16.

[25] S. Kulkarni and et al. Twitter heron: Stream processing at scale. SIGMOD '15.

[26] G. T. Lakshmanan and et al. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing'08.*

[27] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst. 1983.*

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

[29] Y. Li and et al. Numa-aware algorithms: the case of data shuffling. CIDR '13.

[30] H. Miao and et al. Streambox: Modern stream processing on a multicore machine. ATC'17.

[31] D. R. Morrison and et al. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization'16.*

[32] N. Parikh and N. Sundaresan. Scalable and near real-time burst detection from ecommerce queries. SIGKDD'08.

[33] B. Peng and et al. R-storm: Resource-aware scheduling in storm. Middleware '15.

[34] P. Pietzuch and et al. Network-aware operator placement for stream-processing systems. ICDE'06.

[35] I. Psaroudakis and et al. Scaling up concurrent main-memory column-store scans: towards adaptive numa-aware data and task placement. VLDB'15.

[36] K.-L. Tan and et al. In-memory Databases: Challenges and Opportunities From Software and Hardware Perspectives. *SIGMOD Rec'15.*

[37] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. SIGMOD'02.

[38] J. Xu and et al. T-storm: Traffic-aware online scheduling in storm. ICDCS'14.

[39] H. Zhang and et al. In-Memory Big Data Management and Processing: A Survey. *TKDE'15.*

[40] S. Zhang and et al. Revisiting the design of data stream processing systems on multi-core processors. ICDE'17.

[41] J. Zhou and K. A. Ross. Buffering Databse Operations for Enhanced Instruction Cache Performance. SIGMOD'04.