BriskStream: Scaling Data Stream Processing on Shared-Memory Multicores

ABSTRACT

In this paper, we introduce BriskStream, an in-memory data stream processing (DSP) system specifically designed for efficient stream computation on shared-memory multicores. BriskStream is developed based on the code-base of Storm and supports the same APIs. It inherits the basic architectures including pipelined processing and operator replication designs of modern DSP systems. In particular, each operator runs independently in a dedicated Java thread and its replication and placement (i.e., CPU affinity in our context) can be configured in different execution plans. BriskStream's key contribution is an execution plan optimization paradigm, which takes *relative-location* of each pair of producer-consumer operators in non-uniform memory access (NUMA) into consideration. As far as we are concerned, the resulting nontrivial placement optimization problem has not been previously studied on in-memory DSP. We show a branch and bound based approach with several heuristics to resolve the concerned problem. We also propose a simple iterative-based approach to identify the final execution plan considering both operator replication and placement configurations. Besides our NUMA-aware execution plan optimization paradigm, BriskStream also contains several nontrivial modifications that are specifically optimized for shared-memory multi-core architectures. The experimental evaluations demonstrate that BriskStream significantly outperforms the existing DSP systems including Apache Storm and Flink on shared-memory multicores server with eight CPU sockets, up to an order of magnitude of performance speedup.

ACM Reference Format:

. 2019. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicores. In *Proceedings of ACM Conference (SIGMOD'19).* ACM, Amsterdam, The Netherlands, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SIGMOD'19, June 2019, Amsterdam, The Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern multicore processors have demonstrated superior performance in the latency-sensitive applications [13, 39] 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 [2]. On the other hand, prior studies [45] have shown that existing data stream processing (DSP) systems seriously underutilize the underlying complex hardware micro-architecture and especially show poor scalability due to the unmanaged resource competition and remote memory access overhead on multi-socket servers with non-uniform memory access (NUMA) effect.

This paper introduces BriskStream, an in-memory DSP system specifically designed for efficient stream computation on shared-memory multi-socket multicores. BriskStream inherits the basic architectures including pipelined processing and operator replication designs of modern DSP systems. Specifically, an operator can be replicated into multiple instances running in parallel and each instance is treated as an independent execution unit (e.g., a dedicated Java thread), whose placement (i.e., CPU affinity in our context) can be configured. Subsequently, an execution plan determines the number of replicas that each operator have (i.e., operator replication), as well as the way of allocating each instance to the underlying physical resources (i.e., operator placement). In the following, we refer a replica instance of an operator simply as an "operator".

The key contribution of BriskStream is a NUMA-aware execution plan optimization paradigm, called *Relative-Location Aware Scheduling* (RLAS). The goal of RLAS is to find an execution plan such that the application throughput is maximized. RLAS takes the relative location of each pair of producer-consumer (i.e., NUMA distance) into consideration during optimization. It accurately determines the varying processing capability and resource demand of each operator due to varying remote memory access penalty under different placement plans. In this way, it is able to determine the correlation between a solution and its objective value, e.g., predict the output rate of each operator for a given execution plan. This is very different to some related work [21, 41], which assume a predefined and fixed processing capability of each operator.

While RLAS's NUMA-aware performance modelling provides a more accurate estimation of the application

throughput under the NUMA effect, the resulting placement optimization problem becomes much harder to solve. In particular, stochasticity is introduced into the optimization problem as the objective value (e.g., throughput) or weight (e.g., resource demand) of an operator is variable and depends on all previous decisions. This makes classical approaches like dynamic programming not applicable as it is hard to find common sub-problem. Additionally, the placement decisions may conflict with each other and ordering is introduced into the problem. For instance, scheduling of an operator into a socket at one iteration may prohibit some other operators to be scheduled to the same at later iteration.

In this paper, we illustrate how we use a branch and bound optimization technique to solve the concerned problem. In order to reduce the size of the solution space, we introduce three heuristics. The first heuristic switches the placement consideration from vertex to edge, i.e., only consider placement decision of each pair of directly connected operators. This avoids many placement decisions of a single operator that have little or no impact on the output rate of other operators. The second reduces the size of the problem in special cases by applying best-fit policy and also avoids identical sub-problems through redundancy elimination. The third provides a mechanism to tune the trade-off between optimization granularity and searching space.

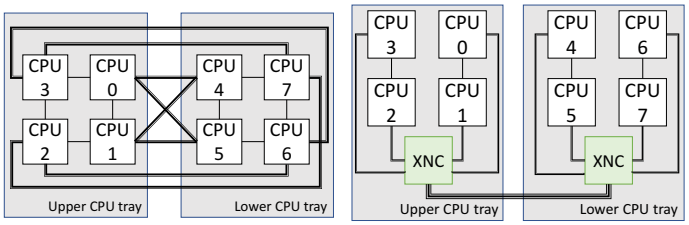
Furthermore, optimizing the replication configuration shall be done together with the placement optimization as an application with a new replication configuration essentially corresponds to a new DAG. We propose a simple iterative approach to identify the final execution plan considering both operator replication and placement configurations, which is mainly based on the idea of iteratively increasing replication level of bottleneck operator that is identified by placement optimization of a given DAG.

BriskStream is developed based on the code-base of Storm and supports the same APIs. More importantly, we discuss two main design aspects that are specifically optimized for shared-memory architectures including improving execution and communication efficiency.

Our experimental results show that our optimization framework scales very well on large NUMA servers. BriskStream significantly outperforms two open-sourced DSP systems including Storm and Flink, up to an order of magnitude. On a single machine, BriskStream [1] achieves roughly 8 million events per second on the real-world application (i.e., linear-road benchmark), which is 2.8 and 12.8 times higher than Storm and Flink, respectively.

Organization. The remainder of this paper is organized as follows. Section 2 covers the necessary background of scale-up servers and overview of BriskStream. Section 3 discusses our NUMA-aware execution plan optimization paradigm, where the performance model and optimization algorithm are presented. Section 4 elaborates several implementation details of BriskStream. We report extensive experimental results in Section 5. Section 6 reviews related work and Section 7 concludes this work.

[1]The source code of BriskStream will be publicly available at https://bitbucket.org/briskStream/briskstream.



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

Figure 1: Interconnect topology for our servers. XNC is the node controller assisting as the glue. Redundancy data path of Server B is omitted and not shown in the figure.

2 BACKGROUND

In this section, we introduce modern scale-up servers and an overview of BriskStream.

2.1 Modern Scale-up Servers

Modern machines 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 ones to carefully align the communication patterns accordingly to get good performance.

Different NUMA configurations exist in nowadays market, which further complicates the software optimization on them. Figure 1 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 the 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 access latency. Table 1 shows the detailed specification of our two NUMA servers. NUMA characteristics, such as local and

Table 1: Characteristics of the two servers we use

Statistic \ Machine	HUAWEI KunLun Servers (Server A)	HP ProLiant DL980 G7 (Server B)
Processor (HT disabled)	8x18 Intel Xeon E7-8890 at 1.2 GHz	8x8 Intel Xeon E7-2860 at 2.27 GHz
Power governors	power save	performance
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

inter-socket idle latencies and peak memory bandwidths, are measured with Intel Memory Latency Checker [7]. These two machines have different NUMA topologies, which lead to different access latencies and throughputs across CPU sockets. The two major takeaways from Table 1 are as follows. First, due to NUMA, both Servers have significantly high remote memory access latency, which is up to 10 times higher than local cache access. Second, Different interconnect and NUMA topologies lead to quite different bandwidth characteristics on these two servers. Server B has a high remote memory access bandwidth, which is, in fact, almost identical to local memory access bandwidth. In contrast, on Server A, remote memory access bandwidth is significantly lower than local memory bandwidth.

2.2 BriskStream Overview

The design goal of BriskStream is to support the same APIs of the popular DSP systems (e.g., Storm) while optimizing the system performance as much as possible targeting at shared-memory multi-socket multicores. An application is expressed as 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. Figure 2 (a) illustrates *word count* (WC) as an example application containing five operators as follows.

(1) *Spout* continuously generates new tuple containing a sentence with random words. Each prepared tuple is then fetched by Parser.
(2) *Parser* drops tuple with a null value. In our testing, the selectivity of the parser of WC is one.
(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.

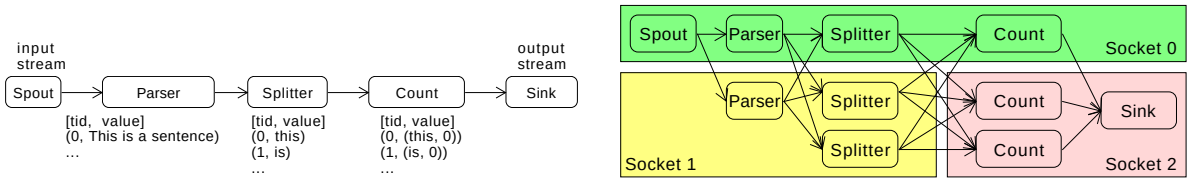
Runtime. There are two important aspects of runtime designs of modern DSP systems [45]. First, the common wisdom of designing the execution runtime of DSP systems is to treat each operator as a single execution unit (e.g., a Java thread) and runs multiple operators in a DAG in a pipelining way. Second, for scalability, each operator may be executed independently in multiple threads. BriskStream inherits the basic architectures including pipelined processing and operator replication designs of modern DSP systems. Such design is well known for its advantage of low processing latency and being adopted by many DSP systems such as Storm [5], Flink [4], SEEP [20], and Heron [27].

3 EXECUTION PLAN OPTIMIZATION

An execution plan concerns how to allocate each operator to underlying physical resources, as well as the number of replicas that each operator should have. In this paper, the goal of execution plan optimization is to find an execution plan such that the application throughput is maximized. Figure 2 (b) illustrates one example execution plan of WC, where parser, splitter and counter are replicated into 2, 3 and 3 instances, and they are placed in three CPU sockets (representing as the coloured rectangles). BriskStream uses a novel NUMA-aware optimization paradigm to optimize its execution plan considering both operator replication and placement. Operator experiences additional remote memory access (RMA) penalty during input data fetch when it is allocated in different CPU sockets to its producers. A bad execution plan may introduce unnecessary RMA communication overhead and/or oversubscribe a few CPU sockets that induces significant resource contention. We propose a novel optimization paradigm called *Relative-Location Aware Scheduling* (RLAS) to optimize replication level and placement (i.e., CPU affinity) of each operator at the same time guided by our performance modelling.

3.1 The Performance Model

Our model is inspired by rate-based optimization (RBO) framework [41]. The major difference is that the original RBO assumes processing capability of an operator is predefined and independent of different execution plans, which is not suitable in our optimization context due to the NUMA effect.



(a) Logical view of word count application.

(b) One example execution plan of word count. Three CPU sockets are used.

Figure 2: Word Count (WC) as an example application.

Table 2: Summary of terminologies

Type	Notation	Definitions
Model inputs	$\mathbf{p}$	Input execution plan
	$\mathbf{I}$	External input rate to source operator
	$\mathbf{C}$	Maximum attainable unit CPU cycles per socket
	$\mathbf{B}$	Maximum attainable local DRAM bandwidth
	$\mathbf{Q_{i,j}}$	Maximum attainable remote channel bandwidth from socket i to socket j
	$\mathbf{L_{i,j}}$	Worst case memory access latency from socket i to socket j
	$\mathbf{S}$	Cache line size
Operator parameters	$\mathbf{M}$	Average memory bandwidth consumption per tuple
	$\boldsymbol{T}$	Average time spent on handling each tuple
	$\boldsymbol{T^f}$	Average fetching time per tuple
	$\boldsymbol{T^e}$	Average execution time per tuple
	$\boldsymbol{N}$	Average size per tuple
Model outputs	$\mathbf{r_o}$	Output rate of an operator
	$\mathbf{\overline{r_o}}$	Expected output rate of an operator
	$\mathbf{r_o(s)}$	Output rate of an operator specifically to producer ``s"
	$\mathbf{r_i}$	Input rate of an operator. r_i of a non-source operator is r_o of its producer and r_i of source operator is external input rate I
	$\boldsymbol{R}$	Application throughput

We summarize the main terminologies used in Table 2. We group them into the following three types, including *model inputs*, *operator parameters* and *model outputs*. Model inputs are the information required by our performance model, and they are given as global constants that are statistics of the machine (e.g., measured by Intel Memory Latency Checker [7]) or given as inputs from the optimization algorithm. Operator parameters are the intermediate information required that is specific to an operator. They need to be directly profiled (e.g., T^e) or indirectly estimated with profiled information and model inputs (e.g., T^f). Model outputs are the final results form the performance model that we are interested in.

Model overview. In this paper, application throughput is the metric to be maximized by optimizations. In the following, we refer to the output rate of an operator by using the symbol r_o, while r_i refers to its input rate. For the sake of simplicity, we refer a replica instance of an operator simply as an "operator".

The throughput (R) of the application is modelled as the summation of r_o of all "sink" operators (i.e., operators with no consumer). That is,

$$R = \sum_{sink} r_o \tag{1}$$

Subsequently, we need to estimate the output rate of each "sink" operator. To simplify the presentation, we omit and assume selectivity is one in the following discussion. In our experiment, the selectivity statistics (e.g., input and output selectivity) of each operator is pre-determined before the optimization applies. In practice, they can be periodically collected during the application running and the optimization needs to be re-performed accordingly.

Estimating r_o. The average processing time per tuple (T) of an operator varies on different execution plans (containing different placements) due to varying input data fetch time caused by the NUMA effect. As a result, the output rate of an operator is not only related to its input rate but also the execution plan, which is quite different from previous studies [41]. We hence estimate r_o of an operator as a function of its input rate r_i and execution plan p, where r_i of spout (i.e., source operator) is given as I (i.e., external input rate configured by user), and r_i of non-source operator is r_o of the corresponding producer.

Consider a time interval t, denote the number of tuples to be processed during t as num and actual time needed to process them as t_p. Further, denote $T(p)$ as the average time spent on handling each tuple for a given execution plan p. Let us first assume input rates are sufficiently large and the operator is always busy during t (i.e., $t_p > t$), and we discuss the case of $t_p \leq t$ at the end of this paragraph. Then, the general formula of r_o can be expressed in Formula 2.

Specifically, *num* is the aggregation of input tuples from all producers arrived during t, and t_p is the total time spent on processing those input tuples.

$$r_o = \frac{num}{t_p},$$
$$where\ num = \sum_{producers} r_i * t$$
$$t_p = \sum_{producers} r_i * t * T(p). \quad (2)$$

We breakdown $T(p)$ into the following non-overlapping components.

- $T^e(p)$: time required in actual function execution and emitting outputs tuples per input tuple under a given execution plan p.
- $T^f(p)$: time required to fetch (local or remotely) the actual data per input tuple under a given execution plan p.

For operators that have a constant workload for each input tuple, we simply measure its average execution time per tuple with one execution plan to obtain its $T^e(p)$. Otherwise, we can use machine learning techniques (e.g., linear regression) to train a prediction model to predict its $T^e(p)$ under varying execution plans. Prediction of an operator (or program) with more complex behaviour has been studied in several previous works [11], and we leave it as future work to enhance our system.

In contrast, $T^f(p)$ is determined by its fetched tuple size and its relative distance to its producer (determined by p), which can be represented as,

$$T^f(p) = \begin{cases} 0 & \text{if collocated with producer} \\ \lceil N/S \rceil * L_(i,j) & \text{otherwise} \end{cases}$$
$$, \text{ where i and j are determined by } p. \quad (3)$$

When the operator is collocated with its producer, the data communication cost is already covered by T^e and hence T^f is 0. On the other hand, it spends additional data transfer cost across CPU sockets per tuple. It is generally difficult to accurately estimate the actual memory communication cost as it is affected by multiple factors such as memory access patterns and hardware prefetcher units. We use a simple formula based on prior work from Surendra and et al. [16] as illustrated in Formula 3. Specifically, we estimate the cross socket communication cost based on the total size of data transfer N bytes per input tuple (measured by classmexer [1]), cache line size S and the worst case memory access latency ($L_(i,j)$) that operator and its producer allocated ($i \neq j$). Despite its simplicity, applications in our testing benchmark roughly follow Formula 3 as we show in our experiments later.

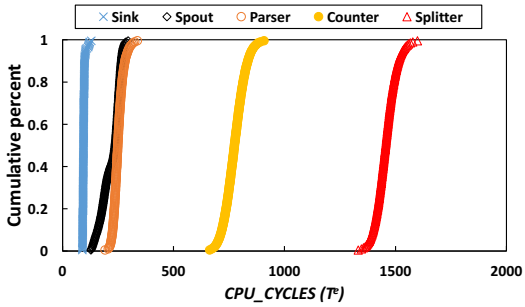


Figure 3: CDF of profiled average execution cycles of different operators of WC.

Finally, let us remove the assumption that input rates are always sufficiently large, and denote the expected output rate as $\overline{r_o}$. There are two cases that we have to consider:

Case 1: We have essentially made an assumption that the operator is in general *over-supplied*, i.e., $t_p > t$. In this case, input tuples are accumulated and $\overline{r_o} = r_o$. As tuples from all producers are processed in a cooperative manner with equal priority, tuples will be processed in a first come first serve manner [2]. Therefore, $r_o(s)$ is proportional to the proportion of the corresponding input ($r_i(s)$), that is, $\overline{r_o(s)} = r_o * \frac{r_i(s)}{r_i}$.

Case 2: On the other hand, operator may need less time to finish processing all tuples arrived during observation time t, i.e., $t_p \leq t$. In this case, we can derived that $r_o \geq \sum_{producers} r_i$. This effectively means the operator is *under-supplied* (or just fulfilled), and its output rate is limited by its input rates, i.e., $\overline{r_o} = r_i$, and $\overline{r_o(s)} = r_i(s)\ \forall$ producer s.

Given an execution plan, we can then identify operators that are over-supplied by comparing its input rate and output rate. Those over-supplied operators are essentially the "bottlenecks" of the current application DAG under the given execution plan. Our scaling algorithm tries to increase the replication level of those operators to remove bottlenecks. After the scaling, we need to again search for the optimal placement plan of the new DAG. This iterative optimization process formed our optimization framework, which will be discussed in detail shortly later in Section 3.2.

Model instantiation. BriskStream requires a profiling stage to collect the necessary running statistics of all operators. Specifically, the application is dry-run launched on a single CPU socket with the replication level of each operator set to one, and we profile each operator to gather

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

parameters including T^e, M (average memory bandwidth consumption per tuple) and N to each of its producers using the overseer library [37]. We sequentially profile each operator, and all the non-relevant operators are paused to prevent interference. Figure 3 shows the profiling results of execution cycles of different operators of WC. Note that, the profiling results had excluded the queue blocking time, and we wait a sufficient warm-up time to stabilize the system before profiling. The major takeaway from Figure 3 is that operators show stable behaviour in general, and the statistics can be used as model input. Selecting a lower (resp. higher) percentile profiled results essentially corresponds to a more (resp. less) optimistic performance estimation. Nevertheless, we use the profiled statistics at the *50th* percentile as the input of the performance model, which successfully guides BriskStream to scale in multicores.

3.2 Algorithms

In this section, we first formally define the execution plan generation problem (Section 3.2.1). We then explain how the branch and bound based technique can be used to solve the concerned optimization problem (Section 3.2.2).

3.2.1 Problem definition. The goal of our optimization is to maximize the throughput of a given application DAG, where we search for the optimal replication level and placement of each operator. Note that, each replica is considered as an operator to be scheduled. For one CPU socket, denote its available CPU cycles as C cycles/sec, the maximum attainable local DRAM bandwidth as B bytes/sec, and the maximum attainable remote channel bandwidth from socket S_i to S_j as $Q_{i,j}$ bytes/sec. Further, denote average tuple size, memory bandwidth consumption and processing time spent per tuple of an operator as N bytes, M bytes/sec and T cycles, respectively, The problem can be mathematically formulated as follows:

$$maximize \sum_{sink} \overline{r_o}$$

under the constraints: $\forall i, j \in 1, .., n$,

$$\sum_{operators\ at\ S_i} \overline{r_o} * T \leq C, \tag{4}$$

$$\sum_{operators\ at\ S_i} \overline{r_o} * M \leq B, \tag{5}$$

$$\sum_{operators\ at\ S_j} \sum_{producers\ at\ S_i} \overline{r_o(s)} * N \leq Q_{i,j}, \tag{6}$$

As the formulas show, we consider three categories of resource constraints that the optimization algorithm needs to make sure the execution plan satisfies. Constraint 4 enforces that the aggregated demand of CPU resource requested to anyone CPU socket must be smaller than the available CPU resource. Constraint 5 enforces that the aggregated amount of bandwidth requested to a CPU socket must be smaller than the maximum attainable local DRAM bandwidth. Constraint 6 enforces that the aggregated data transfer from one socket to another per unit of time must be smaller than the corresponding maximum attainable remote channel bandwidth. In addition, it is also constrained that one operator will be and only be allocated exactly once. This matters because an operator may have multiple producers that are allocated at different places. In this case, the operator may be collocated with only a subset of its producers.

Assuming each operator (in total $|o|$ excluding replicas) can be replicated at most k replicas, we have to consider in total $k^{|o|}$ different replication configurations. In addition, for each replication configuration, there are m^n different placements, where m is the number of CPU sockets and n stands for the total number of replicas ($n \geq |o|$). Such a large solution space makes brute-force unpractical.

3.2.2 RLAS Optimization. The key idea of our optimization process is to iteratively optimize operator placement under a given replication level setting and then try to increase replication level of the "bottleneck operator", which are determined during placement optimization. The bottleneck operator is defined as the operator that has a larger input rate than its processing capability (see Section 3.1 case 1). Figure 4 shows the optimization overview with a simple application consisting of two operators, which is iteratively composed by placement and replication optimization. The initial execution plan with no operator replication is labelled with 0. First, BriskStream optimizes its placement (labelled with 1) with *placement algorithm*, which also identifies bottleneck operators. Subsequently, it tries to increase the replication level of the bottleneck operators (labelled with 2) with *scaling algorithm*. It continues to optimize its placement given the new replication level setting (labelled with 3). Finally, the application with an optimized execution plan (labelled with 4) is submitted to execute.

Determining the optimal replication configuration shall be done together with the placement optimization as an application with a new replication configuration essentially corresponds to a new DAG. Therefore, the scaling optimization relays on placement optimization to provide intermediate optimized execution plan under a given scaling configuration. In the following, we focus on discussing the placement optimization problem. Both the implementations of scaling and placement optimization algorithms are presented in Appendix B.

Branch and Bound based placement optimization. The aforementioned performance modelling provides a

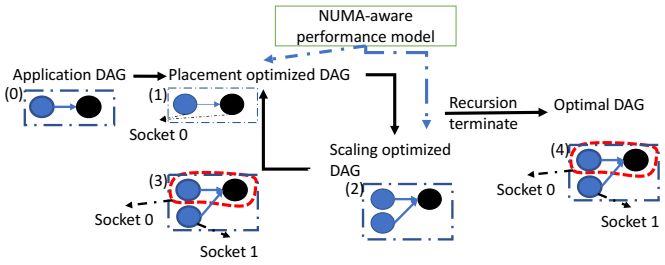


Figure 4: RLAS Optimization overview.

more accurate estimation of the application throughput under the NUMA effect, but the resulting placement optimization problem becomes much harder to solve as we have discussed in Section 1. We now discuss how the B&B based technique [35] is applied to solve our placement optimization problem assuming operator replication is given and fixed. We focus on discussing our bounding function and unique heuristics we used that improves the searching efficiency.

B&B systematically enumerates a tree of candidate solutions, based on a bounding function. There are two types of nodes in the tree: live nodes and solution nodes. In our context, a node represents a placement plan and the value of a node stands for the estimated throughput under the corresponding placement.

Live nodes: A live node contains the placement plan that violates some constraints and they can be expanded into other nodes that violate fewer constraints. The value of a live node is obtained by evaluating the bounding function.

Solution node: A solution node contains a valid placement 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.

In the following, we discuss the bounding function as well as three heuristics to improve searching efficiency.

The bounding function. If the bounding function value of an intermediate node is worse than the solution node 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. Specifically, the bounded value of every live node is obtained by fixing the placement of *valid* operators and let *remaining* operators to be collocated with all of its producers, which may violate resource constraints as discussed before, but gives the upper bound of the output rate that the current node can achieve.

Consider a simple application with operators A, A' (replica of A) and B, where A and A' are producers of B. Assume at one iteration, A and A' are scheduled to socket 0 and 1, respectively (i.e., they become valid). We want to calculate the bounding function value assuming B is the sink operator, which remains to be scheduled. In order to calculate the bounding function value, we simply let B be collocated with both A and A' at the same time, which is certainly invalid. In this way, its output rate is maximized, which is the bounding value of the live node. The calculating of our bounding function has the same cost as evaluating the performance model since we only need to mark T^f (Formula 3) to be 0 for those operators remaining to be scheduled.

The branching heuristics. 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 as one operator is allocated in each iteration. In other words, it will still need to examine on *average* $(n * m)^n$ candidate solutions [33]. 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 introduce three heuristics that work together to significantly reduce the solution space as follows.

1) Collocation heuristic: Naively, we need to consider branching to all placement decisions of each operator in each iteration. However, we observe that some placement decisions have *no or little* impact on the output rate of any operators. We propose to consider a list of *collocation* decisions involving a pair of directly connected producer and consumer. During the searching process, we can remove those collocation decisions from the 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.

2) Best-fit & Redundant-elimination heuristic: Consider an operator to be scheduled, if all predecessors (i.e., upstream operators) of it are already scheduled, then the output rate of it can be safely determined without affecting any of its predecessors. In this case, we select only the best way to schedule it to maximize its output rate. In addition, if there are multiple sockets that it can achieve maximum output rate, we select the one with least remaining resource.

3) Compress graph: Under large replication level setting, the execution graph becomes very large and the searching space is huge. We compress the execution graph by grouping multiple (determined by *compress ratio*) replicas of an operator into a single large instance that is scheduled together. Essentially, the compress ratio trade off the

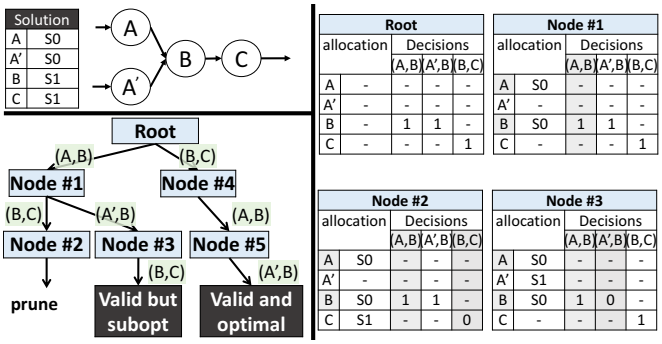


Figure 5: Placement optimization at runtime. Light colored rectangle represents a live node that still violates resource constraints. Dark colored rectangle stands for a solution node contains a valid plan.

optimization granularity and searching space. By setting the ratio to be one, we have the most fine-grained optimization but it takes more time to solve. In our experiment, we set the ratio to be 5, which produces a good trade-off.

We use the scheduling of WC as a concrete example to illustrate the algorithm. For the sake of simplicity, we consider only an intermediate iteration of scheduling of a subset of WC. Specifically, two replicas of the parser (denoted as A and A'), one replica of the splitter (denoted as B), and one replica of count (denoted as C) are remaining unscheduled as shown in the top-left of Figure 5.

In this example, we assume the aggregated resource demands of any combinations of grouping three operators together exceed the resource constraint of a socket, and the only optimal scheduling plan is shown beside the topology. The bottom left of the Figure 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 in this node, although it may violate resource constraints. An operator is interested in the collocation decision involving itself to minimize its remote memory access penalty. 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, and assume they are collocated to S0. Note that, S1 is identical to S0 at this point and does not need to repeatedly consider. The bounding value of this node is essentially collocating all operators into the same socket, and it is larger than solution node hence we need to further explore. At Node #2, we try to collocate A' and B, which however cannot be satisfied (due to the assumed resource constraint). As its bounding value is worse than the solution (if obtained), and it can be pruned safely. On the other hand, Node #3 will eventually lead to a valid yet bad placement plan. One of the searching processes that leads to the solution node is Root→Node #4→Node #5→Solution.

4 IMPLEMENTATION

BriskStream is developed based on the code-base of Storm (version 1.1.1) and inherits the basic architectures. Figure 6 presents an example job overview of BriskStream. Each operator (or the replica) of the application is mapped to one *task*. The task is the fundamental processing unit (i.e., executed by a Java thread), 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 invokes the core logic on the obtained input tuple. After the function execution finishes, 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.

We breakdown the *per-tuple execution time* of an operator into the following components.

- *Execute* phase refers to the average time spent in core function execution. Besides the actual user function execution, it also includes various processor stalls such as instruction cache miss stalls.
- *RMA* phase refers to the time spend due to remote memory access. This is only involved when the operator is scheduled to different sockets to its producers, and it varies depending on the relative location between operators.
- *Others* consist of all other time spent in the critical execution path and considered as overhead. Examples include temporary object creation, exception condition checking and context switching overhead.

All phases shall be minimized in order to improve the performance of the system. Targeting at the shared-memory multicore environment, there are many opportunities to optimize the DSP system. In the following, we discuss two main design aspects of BriskStream.

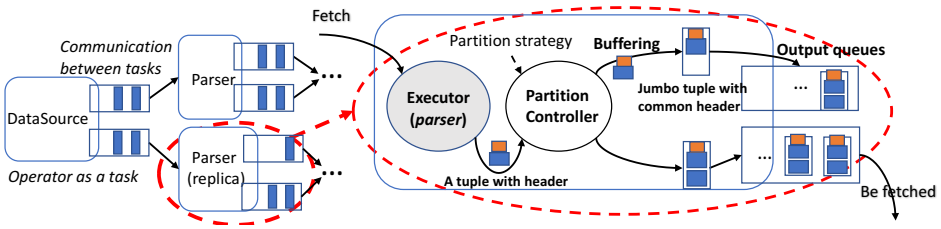


Figure 6: Example job overview in BriskStream.

4.1 Improving Execution Efficiency

Targeting at shared-memory, BriskStream is heavily rewritten to reduce the overhead in its execution. As shown in the previous work [45], instruction footprint between two consecutive invocations of the same function in existing DSP systems is large and resulting in significant instruction cache misses stalls. We eliminate many unnecessary components to reduce the instruction footprint, notably including (de)serialization, cross-process and network-related communication mechanism, and condition checking (e.g., exception handling). Furthermore, we carefully revise the critical execution path to not create unnecessary/duplicate temporary objects. For example, as an output tuple is exclusively accessible by its targeted consumer and all operators share the same memory address, we do not need to create a new instance of the tuple when the consumer obtains it.

4.2 Improving Communication Efficiency

Most modern DSP systems [4, 5, 45] employ buffering strategy to accumulate multiple tuples before sending to improve the application throughput. However, our profiling results reveal that operators spend a large portion of time in emitting tuples in Storm. Storm version 1.1.1 (the baseline of BriskStream) uses LMAX Disruptor[6] as its inter-operator communication backbone. Each output tuple is first added into a buffer. Once the buffer is full (the default size is 100), Storm tries to drain it into the Disruptor queue. This involves another buffering operation of filling up a ringbuffer, one tuple by one tuple. There are mainly two issues of this operation. First, the filling up of ringbuffer (second step) is a duplicate operation, which is unnecessary. Second, the insertion of tuple one by one may induce serious queue contention. It is expected that such odd behaviour will be corrected/improved in the future release of Storm. Nevertheless, in BriskStream, we follow the similar idea of buffering output tuples, but accumulated tuples are combined into one "jumbo tuple" as shown on the right-hand side of Figure 6. This approach has several benefits for scalability. Since we know tuples in the same jumbo tuple are targeting at the same consumer from the same producer in the same process, we can eliminate duplicate tuple header (e.g., metadata, context information) hence reduces communication costs. In addition, the insertion of a jumbo tuple (containing multiple output tuple) requires only one-time access to the communication queue and effectively amortizing the insertion overhead.

It is noteworthy that the aforementioned optimizations are only possible or beneficial for DSP systems designed to run on the shared-memory environment, which is the target of BriskStream. For example, applying our execution plan optimization to the original Storm (without improving the execution and communication efficiency) is less beneficial and not sufficient to make it scale on multicores due to its heavy execution process designed for distributed environment [45]. This can be further validated in our later experiments (Figure 12) where we compare the execution time breakdown between BriskStream and Storm.

5 EVALUATION

In this section, we experimentally evaluate BriskStream in following aspects. First, our proposed performance model accurately predict the throughput of an application under different execution plans (Section 5.2). Second, BriskStream significantly outperforms two popular open-sourced DSP systems on multicores (Section 5.3). Third, our RLAS optimization approach performs significantly better than competing techniques (Section 5.4). We also show in Section 5.5 the relative importance of several of BriskStream's implementation techniques.

5.1 Experimental Setup

We pick four common applications with different characteristics to evaluate BriskStream. These tasks are word-count (WC), fraud-detection (FD), spike-detection (SD), and linear-road (LR) with different topology complexity and varying compute and memory bandwidth demand. Furthermore, LR involves relatively more complex data transmission patterns and it is expected that applying different placement plans will have a larger impact on it. We show the topology of the testing applications in Appendix A. We use the same settings as the previous study [45]. For

Table 3: Model evaluation of varying remote memory access cost. The unit is nanoseconds/tuple.

Splitter			Counter		
From-to	Measured	Estimated	From-to	Measured	Estimated
S0-S0 (local)	1612.80	1612.80	S0-S0 (local)	612.30	612.30
S0-S1	1666.53	1991.14	S0-S1	611.40	665.23
S0-S3	1708.20	1994.85	S0-S3	623.07	665.92
S0-S4	2050.63	2923.65	S0-S4	889.92	837.92
S0-S7	2371.31	3196.35	S0-S7	870.23	888.42

more details, readers can refer to the previous paper. When RLAS optimization is applied, the replication level of each operator is automatically decided by the optimizer. Otherwise, we manually tune the total replication level of all operators to its best achievable performance. We also tune the external input rate (I) to a large value to overfeed the system, and report the stable system performance [3].

We use Server A in Section 5.2, 5.3 and 5.5. We study our RLAS optimization algorithms in detail on different NUMA architectures with both two servers in Section 5.4. To minimize interference of operators, we use OpenHFT Thread Affinity Library [8] with core isolation (i.e., configure *isolcpus* to avoid the isolated cores being used by Linux kernel general scheduler) to bind each task thread to cores based on the given execution plan.

5.2 Performance Model Evaluation

In this section, we evaluate the accuracy of our performance model. We first evaluate the estimation of the cost of remote memory access. We take Split and Count operators of WC as an example. Table 3 compares the measured and estimated access time per tuple of each operator. Our estimation generally captures the correlations between remote memory access penalty and NUMA distance. The estimation is larger than measurement, especially for Splitter. This could be mainly due to the help of the hardware prefetcher unit, and it works especially well for large working set due to a more regular address [29]. Another takeaway from Table 3 is that there is a significant increase of RMA cost from between sockets from the same CPU tray (e.g., S0 to S1) to between sockets from different CPU tray (e.g., S0 to S4). Such non-linear increasing of RMA cost has a major impact on the system scalability as we need to pay significantly more communication overhead from using 4 sockets to more sockets.

To validate the overall effectiveness of our performance model, we show the relative error associated with estimating the application throughput by our analytical model. The

[3]Back-pressure mechanism will eventually slow down spout so that the system is stably running at its best achievable performance.

Table 4: Model accuracy evaluation of all applications. The performance unit is K events/sec.

	WC	FD	SD	LR
Measured	96390.8	7172.5	12767.6	8738.3
Estimated	104843.3	8193.9	12530.2	9298.7
Relative error	0.08	0.14	0.02	0.06

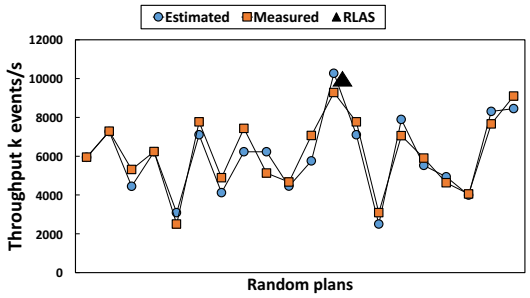


Figure 7: Model accuracy evaluation with 20 randomly generated execution plans of LR.

relative error is defined in Equation 7, where R_{meas} is the measured application throughput and R_{est} is the estimated application throughput by our performance model for the same application. All the experiments are repeated multiple times and the variance is insignificant.

$$relative\_error = \frac{|R_{meas} - R_{est}|}{R_{meas}} \quad (7)$$

The model accuracy evaluation of all applications under the optimal execution plan on eight CPU sockets is shown in Table 4. Overall, our estimation approximates the measurement well for the performance of all four applications. It is able to produce the optimal execution plan and predict the relative performance with the differences less than 2%.

Figure 7 further compares the estimation and measurement of 20 randomly generated execution plans [4] by using LR as an example. Specifically, the replication level of each operator is randomly increased until the total replication level hits the scaling limits. All operators (incl. replicas) are then randomly placed. The major takeaway is that our model successfully estimates (and hence guide to avoid) the potential bad execution plans, which can be as bad as only 27% of the throughput of our optimized plan.

5.3 Evaluation of Execution Efficiency

This section shows that BriskStream significantly outperforms open-sourced distributed DSP systems on

[4]Due to the large solution space, it is difficult to examine all execution plans.

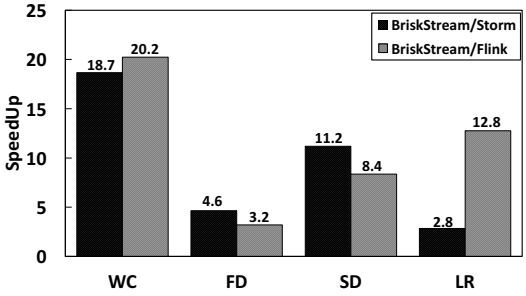


Figure 8: Throughput speedup compared to open-sourced distributed DSP systems.

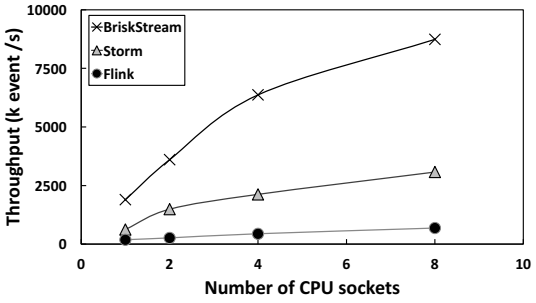


Figure 9: Scalability comparison among different DSP systems using LR as an example.

shared-memory multicores. 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 disable the fault-tolerance mechanism in all comparing systems. 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 even worse performance.

Throughput comparison. Figure 8 shows the significant throughput speedup of BriskStream compared to Storm and Flink. Overall, Storm and Flink show comparable performance for three applications including WC, FD and SD. Flink performs poorly for LR compared to Storm. A potential reason is that Flink requires additional stream merger operator (implemented as the co-flat map) that merges multiple input stream before feeding to an operator with multi-input streams (commonly found in LR). Neither Storm nor BriskStream has such additional overhead.

Evaluation of scalability on varying CPU sockets. Our next experiment shows that BriskStream scales effectively as we increase the numbers of sockets. RLAS re-optimizes the execution plan under a different number of sockets enabled.

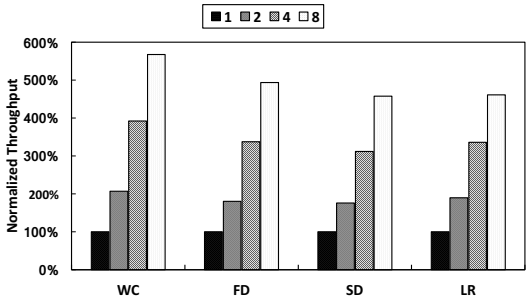


Figure 10: Scalability evaluation of different applications.

Figure 9 shows the better scalability of BriskStream than existing DSP systems on multi-socket servers by taking LR as an example. Both unmanaged thread interference and unnecessary remote memory access penalty prevent existing DSP systems from scaling well on the modern multi-sockets machine.

We show the scalability evaluation of different applications of BriskStream in Figure 10. There is an almost linear scale up from 1 to 4 sockets for all applications. However, the scalability becomes poor when more than 4 sockets are used. This is because of a significant increase of RMA penalty between upper and lower CPU tray. In particular, RMA latency is three times higher between sockets from different tray than the other case.

To better understand the effect of RMA overhead during scaling, we compare the theoretical bounded performance without RMA (denoted as "W/o rma") and ideal performance if the application is linearly scaled up to eight sockets (denoted as "Ideal") in Figure 11. The bounded performance is obtained by evaluating the same execution plan on eight CPU sockets by substituting RMA cost to be zero. There are two major insights taking from Figure 11. First, simply removing RMA cost (i.e., "W/o rma") achieves 89 $\sim$ 95% of the ideal performance, and it hence confirms that the significant increase in RMA cost is the main reason of BriskStream not able to scale linearly on 8 sockets. On the other hand, we still need to re-optimize the execution plan to achieve optimal performance in the presence of changing RMA cost (e.g., in this extreme case, it is reduced to zero).

Per-tuple execution time breakdown. To better understand the source of performance improvement, we show the per-tuple execution time breakdown (as introduced in Section 4) by comparing BriskStream and its baseline, Storm. We use WC as the example application in this study for its simplicity. We measure the total round-trip delay (i.e., the gap between the subsequent call of the core function and time spend in user function per tuple (denoted as *Execute*) of each operator in both systems running on a single CPU

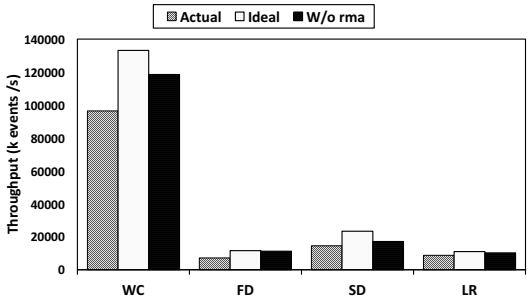


Figure 11: Theoretical performance without RMA.

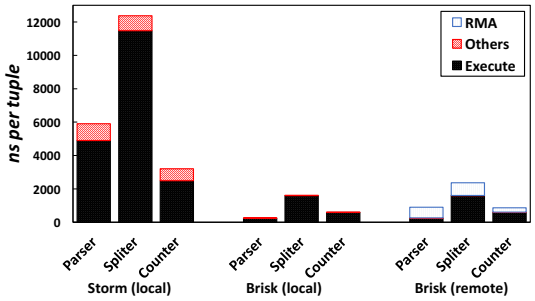


Figure 12: Comparison of per-tuple execution time breakdown.

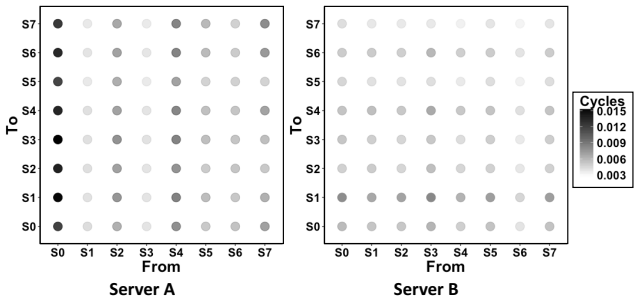


Figure 13: Communication pattern matrices of WC on two Servers. The color scale indicates communication cost in cycles per ns (darker means higher cost).

socket. *Others* is derived as the subtraction from round-trip delay by the average execute time that represents additional overhead, which mainly includes context switching and queue access overhead. The measurement only consists of contiguous successful execution and exclude the time spend in queue blocking (e.g., the queue is empty or full). We also measure *RMA* cost when the operator is remotely allocated to its producer. Figure 12 shows the breakdown of all non-source operators of WC in terms of CPU cycles. We perform analysis in two groups: *local* stands for allocating all operators to the same socket, and *remote* stands for allocating each operator max-hop away from its producer to examine the cost of remote memory access (RMA).

In the local group, we compare execution efficiency between BriskStream and Storm. The "others" overhead of each operator is commonly reduced to about 10% of that of Storm. The function execution time is also significantly reduced to only 5 $\sim$ 24% of that of Storm. There are two main reasons for this improvement. First, the instruction cache locality is significantly improved due to much smaller code footprint. In particular, our further profiling results reveal that BriskStream is no longer front-end stalls dominated (less than 10%), while Storm and Flink are (more than 40%). Second, our "jumbo tuple" design eliminates duplicate metadata creation and successfully amortizing the communication queue access overhead.

In the remote group, we compare the execution of the same operator in BriskStream with or without remote memory access overhead. In comparison with the locally allocated case, the total round trip time of an operator is up to 9.4 times higher when it is remotely allocated to its producer. In particular, Parser has little in computation but has to pay a lot for remote memory access overhead ($T^e << T^f$). The significant different processing capability of the same operator when it is under different placement plan further reaffirms the necessity of our RLAS optimization (Section 3).

Another takeaway is that *Execute* in Storm is much larger than *RMA*, which means $T^e >> T^f$ and NUMA effect may have a minor impact in its plan optimizing. In contrast, thanks to our implementation optimizations (Section 4), BriskStream significantly reduces T^e, and the NUMA effect during communication becomes a critical issue to optimize. In the future, on one hand, T^e may be further reduced with more optimization techniques employed, on the other hand, servers may scale to even more CPU sockets (with potentially larger max-hop remote memory access penalty). We expect that those two trends make the NUMA effect continues to play an important role in optimizing streaming computation on shared-memory multicores.

5.4 Evaluation of RLAS algorithms

In this section, we study the effectiveness of RLAS optimization and compare it with competing techniques.

Communication pattern. In order to understand the impact of different NUMA architectures on RLAS optimization, we show communication pattern matrices of running WC with an optimal execution plan in Figure 13. The same conclusion applies to other applications and hence omitted. Each point in the figure indicates the summation of data fetch cost (i.e., T^f) 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. The main reason is that the remote memory access bandwidth is almost identical to local memory access in Server B thanks to its glue-assisted component as discussed in Section 2, and operators are hence more uniformly placed at different sockets.

Comparing different placement techniques. To gain a better understanding of the importance of relative-location awareness, we evaluate RLAS with different operator placement algorithms.

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

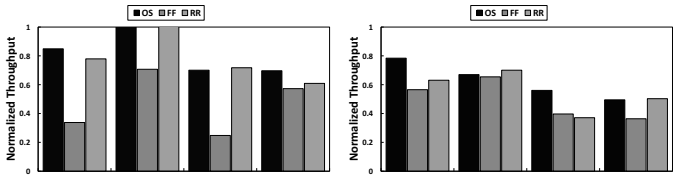
To ensure a fair comparison, the total replication level is fixed to be the same as the total number of CPU cores for different tests. Both FF and RR are enforced to guarantee resource constraints as much as possible. In case they cannot find any plan satisfying resource constraints, they will gradually relax constraints (i.e., increase the available resource per socket while determining if the give execution plan satisfying constraints) until a plan is obtained.

Figure 14 shows that our algorithm generally outperforms other techniques on both two Servers. FF can be view as a minimizing traffic heuristic-based approach as it greedily allocates neighbour operators (i.e., directly connected) together due to its topologically sorting step. Several related works [12, 43] adopt a similar approach of FF in duelling with operator placement problem in the distributed environment. 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. This is due to its greedy nature that leads to a local optimal state. Then, it has to relax the resource constraints and repack the whole topology, which often ends up with oversubscribing of a few CPU sockets. On the other hand, the major drawback of RR is that it does not take remote memory communication overhead into consideration, and the resulting plans often involve unnecessary cross sockets communication.

5.5 Factor analysis

To understand in greater detail the overheads and benefits of various aspects of BriskStream, we show a factor analysis in Figure 15 that highlights the key factors for performance. We use LR as the example workload.

Simple refers to original Storm. *-Instr.footprint* removes unnecessary components resulting in smaller instruction



(a) Server A (b) Server B

Figure 14: Performance comparison among different placement techniques.

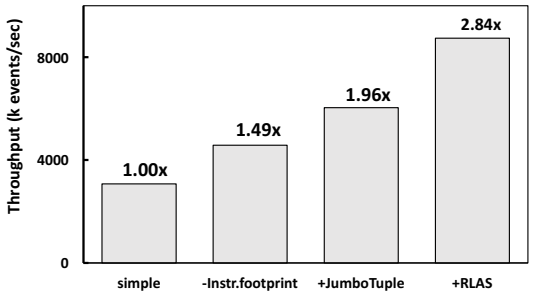


Figure 15: A factor analysis for BriskStream. Changes are added left to right and are cumulative.

footprint and it is hand-optimized to avoid creating unnecessary/duplicate objects described in Section 4.1. *+JumboTuple* allows the operator to reduce the frequency of accessing to the queue described in Section 4.2. *+RLAS* adds the NUMA aware execution plan optimization as described in Section 3.

The major takeaways from Figure 15 are that reducing communication queue access is an important optimization for BriskStream and our RLAS optimization paradigm is critical for DSP system to scale in multicores.

6 RELATED WORK

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 [40, 44]. There have been studies on optimizing the instruction cache performance [22, 46], the memory and cache performance [10, 14, 15, 23] and NUMA [21, 30, 32, 38, 39]. The work most closely related to our RLAS paradigm is presented by Giceva et al. [21]. 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 the 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.

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 [19], Borealis [9], IBM System S [25] and the more recent ones including Storm [5], Flink [4] and Heron [27]. However, most of them targeted at the distributed environment, and little attention has been paid to the research on DSP systems on the modern multicore environment. A recent patch on Flink [3] tries to make Flink a NUMA-aware DSP system. However, its current heuristic based round-robin allocation strategy is not sufficient to make it scales on large multicores as our experiment shows. Closely related to this study, Zhang et al. [45] revisited three common design aspects of modern DSP systems on modern multi-socket multi-core architectures and proposes initial attempts on amending the existing DSP systems to address the identified performance bottlenecks in existing DSP systems. SABER [26] focuses on efficiently realizing computing power from both CPU and GPUs. Streambox [34] provides an efficient mechanism to handle out-of-order arrival event processing in a multi-core environment. Those solutions are complementary to ours and can be potentially integrated together to further improve DSP systems on shared-memory multicores environment.

Execution plan optimization: Both operator scheduling and operator replication are widely investigated in the literature under different assumptions and optimization goals [28]. 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. Aniello et al. [12] propose two schedulers for Storm. The first scheduler is used in an offline manner prior to executing 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 [43] 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 [36], 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. Based on similar idea, we implement algorithms including FF that greedily minimizes communication and RR that tries to ensure resource balancing among CPU sockets. As our experiment demonstrates, both algorithms result in poor performance compared to our RLAS approach in most cases because they are often trapped in local optima. Cardellini et al. [17, 18] propose a general mathematical formulation of the problem of optimizing operator placement for distributed data stream processing (DDSP). However, their approach may lead to a suboptimal performance in the NUMA environment that we are target at. This is because factors including output rate, amount of communication as well as resource consumption of an operator may change in different execution plans due to the NUMA effect and can therefore mislead existing approaches that treat them as predefined constants. Recently, Li et al. [31] present a machine-learning based framework for minimizing end-to-end processing latency on DDSP. It generally remains an open-question whether an optimizer-based (like ours) or model-free approach is more promising [42] in query optimization, and we leave it as a future work to explore the capabilities of the model-free approach in the shared-memory multicores setting.

7 CONCLUSION

This paper introduces BriskStream, a data stream processing (DSP) system specially designed for shared-memory multicores. Particularly, we propose a NUMA-aware plan optimizer of DSP systems, namely Relative-Location Aware Scheduling (RLAS), that is able to generate efficient execution plans considering both operator replication and placements under large multi-socket servers. We compare BriskStream using four streaming applications with Apache Storm and Flink on two modern scale-up servers. The results show that BriskStream significantly outperforms two open-sourced DSP systems up to an order of magnitude even without the tedious tuning process. As a continuation of this work, we plan to evaluate and possibly augment our model to accommodate alternative scenarios to the ones we presented here. We consider a multi-NUMA servers environment, which will additionally introduce the network-communication delays. Furthermore, we also plan to investigate the potential of extension to other optimization techniques such as operator fission and fusion.

REFERENCES

[1] 2008. Classmexer. https://www.javamex.com/classmexer/
[2] 2015. SGI UV 300 UV300EX Data Sheet, http://www.newroute.com/upload/updocumentos/a06ee4637786915bc954e850a6b5580f.pdf.
[3] 2017. NUMA patch for Flink, https://issues.apache.org/jira/browse/FLINK-3163.
[4] 2018. Apache Flink. https://flink.apache.org/
[5] 2018. Apache Storm. http://storm.apache.org/
[6] 2018. Disruptor. https://lmax-exchange.github.io/disruptor/
[7] 2018. Intel Memory Latency Checker, https://software.intel.com/articles/intelr-memory-latency-checker.
[8] 2018. OpenHFT. https://github.com/OpenHFT/Java-Thread-Affinity
[9] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong H. Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing,

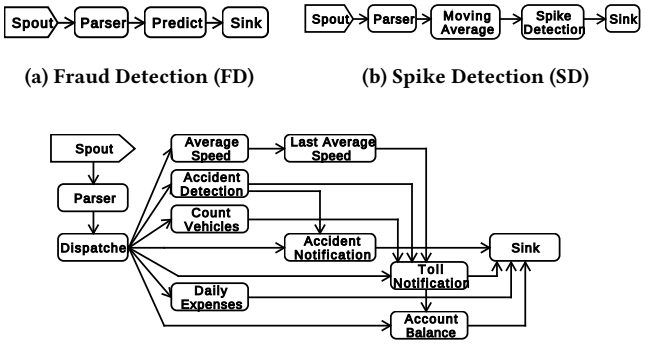
and Stan Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *CIDR*.
[10] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 2009. DBMSs on a Modern Processor: Where Does Time Go?. In *VLDB*.
[11] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *ICDE*.
[12] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. 2013. Adaptive Online Scheduling in Storm. In *DEBS*.
[13] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. 2013. Scale-up vs scale-out for hadoop: Time to rethink?. In *SoCC*.
[14] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. 2013. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *ICDE*.
[15] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the new. Bottleneck: Memory Access. In *VLDB*.
[16] Surendra Byna, Xian He Sun, William Gropp, and Rajeev Thakur. 2004. Predicting memory-access cost based on data-access patterns. In *ICCC*.
[17] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2016. Optimal Operator Placement for Distributed Stream Processing Applications. In *DEBS*.
[18] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2017. Optimal Operator Replication and Placement for Distributed Stream Processing Systems. In *SIGMETRICS Perform. Eval. Rev.*
[19] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*.
[20] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*.
[21] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. 2014. Deployment of Query Plans on Multicores. In *VLDB*.
[22] Stavros Harizopoulos and Anastassia Ailamaki. 2006. Improving Instruction Cache Performance in OLTP. *ACM Trans. Database Syst* (2006).
[23] Bingsheng He, Qiong Luo, and B. Choi. 2005. Cache-conscious automata for XML filtering. In *ICDE*.
[24] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* (2014).
[25] Navendu Jain and et al. 2006. Design, Implementation, and Evaluation of the Linear Road Bnchmark on the Stream Processing Core. In *SIGMOD*.
[26] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *SIGMOD*.
[27] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthikeyan Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *SIGMOD*.
[28] Geetika T. Lakshmanan, Ying Li, and Robert E. Strom. 2008. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Computing* (2008).
[29] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn&Rsquo;T, and Why. *ACM Trans. Archit. Code Optim.* (2012).
[30] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD*.
[31] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. 2018. Model-free Control for Distributed Stream Data Processing Using Deep Reinforcement Learning. *Proc. VLDB Endow.* (2018).
[32] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *CIDR*.
[33] Devroye Luc and Zamora-Cura Carlos. 1999. On the Complexity of Branch-and Bound Search for Random Trees. *Random Struct. Algorithms* (1999).
[34] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2017. StreamBox: Modern Stream Processing on a Multicore Machine. In *ATC*.
[35] David R. Morrison, Sheldon H. Jacobson, Jason J. Sauppe, and Edward C. Sewell. 2016. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization* (2016).
[36] Boyang Peng and et al. 2015. R-Storm: Resource-Aware Scheduling in Storm. In *Middleware*.
[37] Achille Peternier, Daniele Bonetta, Walter Binder, and Cesare Pautasso. 2011. Overseer: low-level hardware monitoring and management for Java. In *PPPJ*.
[38] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. 2012. OLTP on Hardware Islands. *PVLDB* (2012).
[39] Iraklis Psaroudakis and et al. 2015. Scaling up concurrent main-memory column-store scans: towards adaptive NUMA-aware data and task placement. In *VLDB*.
[40] Kian-Lee Tan, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. 2015. In-memory Databases: Challenges and Opportunities From Software and Hardware Perspectives. *SIGMOD Record* (2015).
[41] Stratis D. Viglas and Jeffrey F. Naughton. 2002. Rate-based Query Optimization for Streaming Information Sources. In *SIGMOD*.
[42] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *ICDE*.
[43] J. Xu, Z. Chen, J. Tang, and S. Su. 2014. T-Storm: Traffic-Aware Online Scheduling in Storm. In *ICDCS*.
[44] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* (2015).
[45] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. 2017. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors. In *ICDE*.
[46] Jingren Zhou and Kenneth A. Ross. 2004. Buffering Databse Operations for Enhanced Instruction Cache Performance. In *SIGMOD*.

A APPLICATION TOPOLOGY

We have shown the topology of WC before at Figure 2, and Figure 16 shows the topology of the other three applications.

B ALGORITHM IMPLEMENTATIONS

In this section, we first present the detailed algorithm implementations including operator replication optimization (shown in Algorithm 1) and operator placement (shown in Algorithm 2). After that, we discuss observations made in applying algorithms in optimizing our workload and



(a) Fraud Detection (FD)

(b) Spike Detection (SD)

(c) Linear Road (LR)

Figure 16: Topologies of other three applications in our benchmark.

their runtime (Section B.1). We further elaborate how our optimization paradigm can be extended with other optimization techniques (Section B.2).

Algorithm 1 illustrates our scaling algorithm based on topological sorting. Initially, we set replication of each operator to be one (Lines 1~2). The algorithm proceeds with this and it optimizes its placement with Algorithm 2 (Line 6). Then, it stores the current plan if it ends up with better performance (Lines 7~8). At Lines 11~17, we iterate over all the sorted list from reversely topologically sorting on the execution graph in parallel (scaling from sink towards spout). At Line 15, it tries to increase the replication level of the identified bottleneck operator (i.e., this is identified during placement optimization). The size of increasing step depends on the ratio of over-supply, i.e., $\lceil \frac{r_i}{r_o} \rceil$. It starts new iteration to search for better execution plan at Line 17. The iteration loop ensures that we have gone through all the way of scaling the topology bottlenecks. We can set an upper limit on the total replication level (e.g., set to the total number of CPU cores) to terminate the procedure earlier. At Lines 9&19, either the algorithm fails to find a plan satisfying resource constraint or hits the scaling upper limit will cause the searching to terminate.

Algorithm 2 illustrates our *Branch and Bound based Placement* algorithm. Initially, no solution node has been found so far and we initialize a root node with a plan collocating all operators (Line 1~5). At Line 7~14, the algorithm explores the current node. If it has better bounding value than the current solution, we update the solution node (Line 10~11) if it is valid (i.e., all operators are allocated), or we need to further explore it (Line 13). Otherwise, we prune it at Line 14 (this also effectively prunes all of its children nodes). The branching function (Line 15~32) illustrates how the searching process branches and generates children nodes

Algorithm 1: Topologically sorted iterative scaling

```
   Data: Execution Plan: p // the current visiting plan
   Data: List of operators: sortedLists
   Result: Execution Plan: opt // the solution plan
1  p.parallelism ← set parallelism of all operators to be 1;
2  p.graph ← creates execution graph according to
     p.parallelism;
3  opt.R ← 0;
4  return Searching(p);
5  Function Searching(p):
6      p.placement ← placement optimization of p.graph;
7      if p.R > opt.R then
8          opt ← p // update the solution plan
9      if failed to find valid placement satisfying resource
         constraint then
10         return opt;
11     sortedLists ← reverse TopologicalSort
         (p.graph)// scale start from sink
12     foreach list ∈ sortedLists do
13         foreach Operator o ∈ list do
14             if o is bottleneck then
15                 p.parallelism ← try to increase the
                     replication of o by ⌈r_i / r_o⌉;
16                 if suceessfully increased p.parallelism
                     then
17                     return Searching(p) // start
                         another iteration
18                 else
19                     return opt
20     return opt;
```

to explore. For each collocation decision in the current node (Line 16), we apply the *best fit heuristic* (Line 17~23) and one new node is created. Otherwise, at Line 25~27, we have to create new nodes for each possible way of placing the two operators (i.e., up to $\binom{m}{1} * \binom{2}{1}$). At Line 28~ 31, we update the number of valid operators and bounding value of each newly created nodes in parallel. Finally, the newly created children nodes are pushed back to the stack.

B.1 Discussion

In this section, we discuss some observations made in applying algorithms in optimizing our workload and their optimization runtime.

Observations. We have made some counter-intuitive observations in optimizing our workload. *First*, placement algorithm (Algorithm 2) start with no initial solution (i.e., the *solution.value* is 0 initially at Line 9) by default, and we have tried to use a simple first-fit (FF) placement algorithm

Algorithm 2: B&B based placement optimization

```
Data: Stack stack // stors all live nodes
Data: Node solution // stores the best plan found so far
Data: Node e// the current visiting node
Result: Placement plan of solution node
// Initilization
1  solution.R ← 0 // No solution yet
2  e.decisions ← a list contains all possible collocation decisions;
3  e.plan ← all operators are collocated into the same CPU socket;
4  e.R ← BoundingFunction(e.plan);
5  e.validOperators ← 0;
6  Push(stack,e);
   // Branch and Bound process
7  while ¬IsEmpty(stack) do
8      e ← Pop(stack);
9      if e.R > solution.R then
10         if e.validOperators == totalOperators then
11             solution ← e;
12         else
13             Branching(e);
14     else
           // the current node has worse bounded value than solution, and can be safely pruned.
15 Function Branching(e):
       Data: Node[] children
16     foreach pair of O_s and O_c in e.decisions do
17         if all predecessors of them are already allocated except O_s to O_c then
18             #newAllocate ← 2;
19             if they can be collocated into one socket then
20                 create a Node n with a plan collocating them to one socket;
21             else
22                 create a Node n with a plan separately allocating them to two sockets;
23             add n to children;
24         else
25             #newAllocate ← 1;
26             foreach valid way of placing O_s and O_c do
27                 create a new Node and add it to children;
28     foreach Node c ∈ children// update in parallel
29     do
30         c.validOperators ← e.validOperators + #newAllocate;
31         c.R ← BoundingFunction(c.plan);
32     PushAll(stack, children);
```

to determine an initial solution node to potentially speed up the searching process. In some cases, it accelerates the searching process by earlier pruning and makes the algorithm converges faster, but in other cases, the overhead of running the FF algorithm offsets the gains. *Second*, the placement algorithm may fail to find any valid plan as not able to allocate one or more operators due to resource constraints, which causes scaling algorithm to terminate. It is interesting to note that this *may not* indicates the saturation of the underlying resources but the operator itself is too coarse-grained. The scaling algorithm can, instead of terminate (at Algorithm 1 Line 10), try to further increase the replication level of operator that "failed-to-allocate". After that, workloads are essentially further partitioned among more replicas and the placement algorithm may be able to find a valid plan.

Optimization runtime. The concerned placement optimization problem is difficult to solve as the solution space increase rapidly with large replication configuration. Besides the three proposed heuristics, we also apply a list of optimization tricks to further increase the searching efficiency including 1) memorization in evaluating performance model under a given execution plan (e.g., an operator should behave the same if its relative placement with all of its producers are the same in different plans), 2) heuristically scaling starting point (i.e., instead of starting from scaling with replication set to one for all operators, we can start from a reasonable large DAG configuration to reduce the number of scaling iteration) and 3) parallel branching (i.e., concurrently generate branching children nodes). Overall, the placement algorithm needs less than 5 seconds to optimize placement for a large DAG, and overall scaling takes less than 30 seconds, which is acceptable, given the size of the problem and the fact that the generated plan can be used for the whole lifetime of the application. As the streaming application potentially runs forever, the overhead of generating a plan is not included in our measurement.

B.2 Extension with other optimization techniques

A number of optimization techniques are available in the literature [24]. Many of them can be potentially applied to further improve the performance of BriskStream. Our performance model is general enough such that it can be extended to capture other optimization techniques.

Taking operator fusion as an example, operator fusion trades communication cost against pipeline parallelism and is in particular helpful if operators share little in common computing resource. In our context, let T^e_{fused} and T^f_{fused} to denote the average execution time and fetch time per tuple of the fused operator, respectively. Then, T^e_{fused} can be

estimated as a summation of T^e of all fused operators. T^f_{fused} can be estimated as the T^f of the upstream operator (O_{up}) to be fused. That is,

$$T^e_{fused} = \sum_{\text{all fused operators}} T^e$$
$$T^f_{fused} = T^f_{up} \tag{8}$$

The similar idea has been explored in recent work [21]. In this paper, we focus on operator scheduling and replication optimization, and we leave the evaluation of extension to other optimization techniques as future work.