

BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures

Shuhao Zhang*

National University of Singapore

Amelie Chi Zhou

Shenzhen University

Jiong He

Advanced Digital Sciences Center

Bingsheng He

National University of Singapore

ABSTRACT

We introduce BriskStream, an in-memory data stream processing system (DSPSs) specifically designed for modern shared-memory multicore architectures. BriskStream's key contribution is an execution plan optimization paradigm, namely RLAS, which takes *relative-location* (i.e., NUMA distance) of *each pair* of producer-consumer operators into consideration. We propose a branch and bound based approach with three heuristics to resolve the resulting nontrivial optimization problem. The experimental evaluations demonstrate that BriskStream yields much higher throughput and better scalability than existing DSPSs on multi-core architectures when processing different types of workloads.

ACM Reference Format:

Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures. In *2019 International Conference on Management of Data (SIGMOD '19), June 30–July 5, 2019, Amsterdam, Netherlands*. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3299869.3300067>

1 INTRODUCTION

Modern multicore processors have demonstrated superior performance for real-world applications [14] with their increasing 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].

*Work done while as research trainee at SAP Singapore.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '19, June 30–July 5, 2019, Amsterdam, Netherlands

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-5643-5/19/06...\$15.00

<https://doi.org/10.1145/3299869.3300067>

Witnessing the emergence of modern commodity machines with massively parallel processors, researchers and practitioners find shared-memory multicore architectures an attractive platform for streaming applications [35, 42, 54]. However, prior studies [54] have shown that existing data stream processing system (DSPSs) underutilize the underlying complex hardware micro-architecture and show poor scalability due to the unmanaged resource competition and unawareness of non-uniform memory access (NUMA) effect.

Many DSPSs, such as Storm [5], Heron [36], Flink [4] and Seep [25], share similar architectures including *pipelined* processing and operator *replication* designs. Specifically, an application is expressed as a DAG (directed acyclic graph) where vertexes correspond to continuously running operators, and edges represent data streams flowing between operators. To sustain high input stream ingress rates, each operator can be replicated into multiple *replicas* running in parallel threads. A *streaming execution plan* determines the number of replicas of each operator (i.e., operator replication), as well as the way of allocating each operator to the underlying CPU cores (i.e., operator placement). In this paper, we address the question of how to find a streaming execution plan that maximizes processing throughput of DSPS in shared memory multi-core architectures.

NUMA-aware system optimizations have been previously studied in the context of relational database [27, 39, 48]. However, those works are either 1) focused on different optimization goals (e.g., better load balancing [48] or minimizing resource utilization [27]) or 2) based on different system architectures [39]. They provide highly valuable techniques, mechanisms and execution models but none of them uses the knowledge at hand to solve the problem we address.

The key challenge of finding an optimal streaming execution plan on multicore architectures is that there is a *varying* processing capability and resource demand of each operator due to *varying* remote memory access penalty under *different* execution plans. Witnessing this problem, we present a novel NUMA-aware streaming execution plan optimization paradigm, called *Relative-Location Aware*

Scheduling (RLAS). RLAS takes the relative location (i.e., NUMA distance) of each pair of producer-consumer into consideration during optimization. In this way, it is able to determine the correlation between a solution and its objective value, e.g., predict the throughput of *each* operator for a given execution plan. This is different to some related studies [27, 34, 51], which assume a predefined and fixed processing capability (or cost) of each operator.

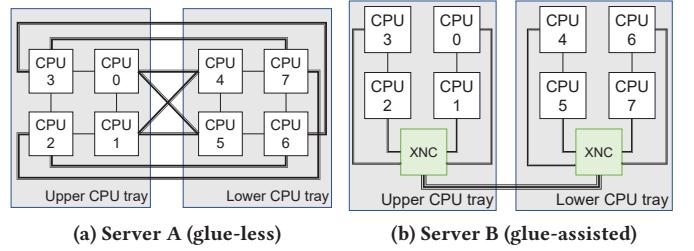
While RLAS provides a more accurate estimation of the application behavior under the NUMA effect, the resulting placement optimization problem is still challenging to solve. In particular, stochasticity is introduced into the problem as the objective value (e.g., throughput) or weight (e.g., resource demand) of each operator is variable and depends on all previous decisions. This leads to a huge solution space. Additionally, the placement decisions may conflict with each other and order constraints are introduced into the problem. For instance, scheduling of an operator at one iteration may prohibit some other operators to be scheduled to the same socket later.

We propose a branch and bound based approach to solve the concerned placement optimization problem. In order to reduce the size of the solution space, we further introduce three heuristics. The first switches the placement consideration from vertex to edge, i.e., only consider placement decision of each pair of directly connected operators. and avoids many placement decisions that have little or no impact on the objective value. 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.

RLAS optimizes both replication and placement at the same time. The key to optimize replication configuration of a streaming application is to remove bottlenecks in its streaming pipeline. As each operator's throughput and resource demand may *vary* in different placement plans due to the NUMA effect, removing bottlenecks has to be done together with placement optimization. To achieve this, RLAS iteratively increases replication level of the bottleneck operator which is identified during placement optimization.

We implemented RLAS in BriskStream with additional optimizations on shared memory (details in Section 5), a new DSPS supporting the same APIs as Storm and Heron. Our extensive experimental study on two eight-socket modern multicore servers show that BriskStream achieves much higher throughput and better scalability than existing DSPSs.

Organization. The remainder of this paper is organized as follows. Section 2 covers the necessary background of scale-up servers and an overview of DSPSs. Section 3 discusses the performance model and problem definition of



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

Figure 1: Interconnect topology for our servers.

RLAS, followed by a detailed algorithm design in Section 4. Section 5 discusses how we optimize BriskStream for shared-memory architectures. We report extensive experimental results in Section 6. Section 7 reviews related work and Section 8 concludes this work.

2 BACKGROUND

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

2.1 Modern Scale-up Servers

Modern machines scale to multiple sockets with non-uniform-memory-access (NUMA) architecture. Each socket 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 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 today’s market. Figure 1 illustrates the NUMA topologies of our servers in the experiments. 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 an eXternal Node Controller (called XNC [6]) that interconnects upper and lower CPU tray (each tray contains 4 CPU sockets). The XNC maintains a directory of the contents of each processors cache and significantly reduces remote memory access latency. The detailed specifications of our two servers are shown in our experimental setup (Section 6).

2.2 DSPS Overview

A streaming application is expressed as a DAG (directed acyclic graph) where vertexes correspond to continuously running operators, and edges represent data streams flowing between operators. Figure 2(a) illustrates *word count* (WC) as an example application containing five operators as

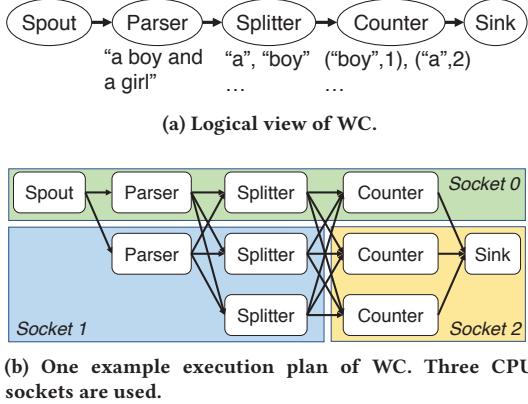


Figure 2: Word Count (WC) as an example application.

follows. *Spout* continuously generates new tuple containing a sentence with ten random words. *Parser* drops invalid tuples (e.g., containing empty value). In our testing workload, the selectivity of the parser is one. *Splitter* processes each tuple by splitting the sentence into words and emits each word as a new tuple to *Counter*. *Counter* maintains and updates a hashmap with the key as the word and the value as the number of occurrences of the corresponding word. Every time it receives a word from *Splitter*, it updates the hashmap and emits a tuple containing the word and its current occurrence. *Sink* increments a counter each time it receives tuple from *Counter*, which we use to monitor the performance of the application.

There are two important aspects of runtime designs of modern DSPSs [54]. First, the common wisdom of designing the execution runtime of DSPSs 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. Such design is adopted by many DSPSs such as Storm [5], Flink [4], Seep [25], and Heron [36] for its advantage of low processing latency. Figure 2(b) illustrates one example execution plan of WC, where parser, splitter and counter are replicated into 2, 3 and 3 replicas, and they are placed in three CPU sockets (represented as coloured rectangles).

3 EXECUTION PLAN OPTIMIZATION

A streaming execution plan concerns how to allocate each operator to underlying physical resources, as well as the number of replicas that each operator should have. An 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 unnecessarily high RMA communication

overhead and/or oversubscribe a few CPU sockets that induces significant resource contention. In this section, we discuss the performance model that guides optimization process and the formal definition of our problem.

3.1 The Performance Model

Model guided deployment of query plans has been previously studied in relational databases on multi-core architectures, for example [27]. Due to the difference in problem assumptions and optimization goals, we adopt a different approach – the rate-based optimization (RBO) approach [51], where output rate of each operator is estimated. However, the original RBO [51] assumes processing capability of an operator is predefined and independent of execution plans, which is not suitable under the NUMA effect.

We summarize the main terminologies of our performance model in Table 1. We group them into the following four types, including *machine specifications*, *operator specifications*, *plan inputs* and *model outputs*. For the sake of simplicity, we refer a replica of an operator simply as an “operator”. Machine specifications are the information of the underlying hardware. Operator specifications are the information specific to an operator, which need to be directly profiled (e.g., T^e) or indirectly estimated with profiled information and model inputs (e.g., T^f). Plan inputs are the specification of the execution plan including both placement and replication plans as well as external input rate to the source operator. Model outputs are the final results of the performance model. To simplify the presentation, we omit the selectivity estimation and assume selectivity is one in the following discussion. In our experiment, the selectivity statistics of each operator are pre-profiled before the optimization applies. In practice, they can be periodically collected during runtime and the optimization needs to be re-performed accordingly.

Model overview. In the following, we refer to the output rate of an operator using the symbol r_o , while r_i refers to its input rate. 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, $R = \sum_{sink} r_o$. To estimate R , we hence need to estimate r_o of each sink operator. The output rate of an operator is not only related to its input rate but also the execution plan due to NUMA effect, which is quite different from previous studies [51].

As BriskStream adopted the pass-by-reference message passing approach (See Appendix A) to utilize shared-memory environment, the reference passing delay is negligible. Hence, r_i of an operator is simply r_o of the corresponding producer and r_i of spout (i.e., source operator) is given as I (i.e., external input stream ingress rate). Conversely, upon obtaining the reference, an operator then needs to fetch the actual data

Table 1: Summary of terminologies

Type	Notation	Definitions
Machine specific.	C	Maximum attainable unit CPU cycles per socket
	B	Maximum attainable local DRAM bandwidth
	Q _{i,j}	Maximum attainable remote channel bandwidth from socket i to socket j
	L _{i,j}	Worst case memory access latency from socket i to socket j
	S	Cache line size
Operator specific.	M	Average memory bandwidth consumption per tuple
	T	Average time spent on handling each tuple
	T ^f	Average fetching time per tuple
	T ^e	Average execution time per tuple
	N	Average size per tuple
Plan inputs	p	Input execution plan
	I	External input stream ingress rate to source operator
Model outputs	r _o	Output rate of an operator
	\overline{r}_o	Expected output rate of an operator
	r _{o(s)}	Output rate of an operator specifically to producer "s"
	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
	R	Application throughput

during its processing, where the actual data fetch delay depends on NUMA distance between it and its producer. We hence estimate r_o of an operator as a function of its input rate r_i and execution plan p .

Estimating r_o . 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 rate to the operator is 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 1. Specifically, num is the total number 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 $num = \sum_{producers} r_i \times t$

$$t_p = \sum_{producers} r_i \times t \times T(p). \quad (1)$$

We breakdown $T(p)$ into the following two non-overlapping components, T^e and T^f (i.e., $T(p) = T^e + T^f$).

T^e stands for time required in actual function execution and emitting output tuples per input tuple. 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 . Otherwise, we can use machine learning techniques (e.g., linear regression) to train a prediction model to predict its T^e under varying execution plans. Prediction of an operator with more complex behaviour has been studied in previous works [12], and we leave it as future work to enhance our system.

T^f stands for time required to (locally or remotely) fetch the actual data per input tuple. It is determined by its fetched tuple size and its relative distance to its producer (determined by p), which can be represented as follows,

$$T^f = \begin{cases} 0 & \text{if collocated with producer} \\ \lceil N/S \rceil \times L_{(i,j)} & \text{otherwise} \end{cases}$$

where i and j are determined by p . (2)

When the operator is collocated with its producer, the data fetch cost is already covered by T^e and hence T^f is 0. Otherwise, it experiences memory access across CPU sockets per tuple. It is generally difficult to accurately estimate the actual data transfer cost as it is affected by multiple factors such as memory access patterns and hardware prefetcher units. We use a simple formula based on a prior work [17] as illustrated in Formula 2. Specifically, we estimate the cross socket communication cost based on the total size of data transfer N bytes per input tuple, cache line size S and the worst case memory access latency ($L_{(i,j)}$) that operator and its producer allocated ($i \neq j$). Applications in our testing benchmark roughly follow Formula 2 as we show in our experiments later.

Finally, let us remove the assumption that input rate to an operator is larger than its capacity, 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., $t_p \geq t$. 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. It is possible to configure different priorities among different operators here, which is out of the scope of this paper. Therefore, $r_o(s)$ is determined by the proportion of the corresponding input ($r_i(s)$), that is, $\frac{r_o(s)}{r_o} = r_o \times \frac{r_i(s)}{r_i}$.

Case 2: In contrast, an operator may need less time to finish processing all tuples arrived during observation time t , i.e., $t_p < t$. In this case, we can derive that $r_o \geq$

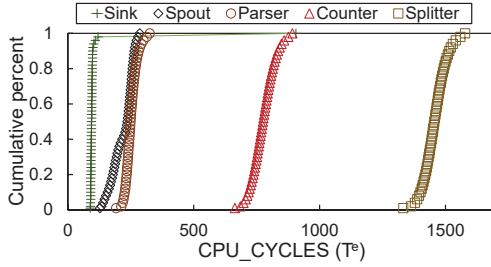


Figure 3: CDF of profiled average execution cycles of different operators of WC.

$\sum_{producers} r_i$. This effectively means the operator is *under-supplied*, and its output rate is limited by its input rates, i.e., $\overline{r_o} = r_i$, and $r_o(s) = r_i(s) \forall$ 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 corresponding execution plan. Our scaling algorithm tries to increase the replication level of those operators to remove bottlenecks. After the scaling, we need to again look for the optimal placement plan of the new DAG. This iterative optimization process formed our optimization framework, which will be discussed shortly later in Section 4.

Model instantiation. Machine specifications of the model including C , B , $Q_{i,j}$, $L_{i,j}$ and S are given as statistics information of the targeting machine (e.g., measured by Intel Memory Latency Checker [7]). Similar to the previous work [23], we need to profile the application to determine operator specifications. To eliminate the impact of interference, we sequentially profile each operator. Specifically, we first launch a *profiling thread* of the operator to profile on one core. Then, we feed sample input tuples (stored in local memory) to it. Information including T^e (execution time per tuple), M (average memory bandwidth consumption per tuple) and N (size of input tuple) is then gathered during its execution.

The sample input is prepared by pre-executing all upstream operators. As they are not running during profiling, they will not interfere with the profiling thread. To speed up the instantiation process, multiple operators can be profiled at the same time as long as there is no interference among the profiling threads (e.g., launch them on different CPU sockets). The statistics gathered without interference are used in the model as BriskStream avoids interference (see Section 3.2). Task oversubscribing has been studied in some earlier work [31], but it is not the focus of this paper.

We use the overseer library [45] to measure T^e , M , and use classmexer library [1] to measure N . Figure 3 shows the profiling results of T^e of different operators of WC.

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 model, which sufficiently guides the optimization process.

3.2 Problem Formulation

The goal of our optimization is to maximize the application processing throughput under given input stream ingress rate, where we look for the optimal replication level and placement of each operator. 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 Equation 3–5.

As the formulas show, we consider three categories of resource constraints that the optimization algorithm needs to make sure the execution plan satisfies. Constraint in Eq. 3 enforces that the aggregated demand of CPU resource requested to anyone CPU socket must be smaller than the available CPU resource. Constraint in Eq. 4 enforces that the aggregated amount of bandwidth requested to a CPU socket must be smaller than the maximum attainable local DRAM bandwidth. Constraint in Eq. 5 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 is allocated exactly once. This matters because an operator may have multiple producers that are allocated at different places. In this case, the operator can only be collocated with a subset of its producers.

$$\begin{aligned} & \text{maximize} \sum_{\text{sink}} \overline{r_o} \\ & \text{s.t., } \forall i, j \in 1, \dots, n, \\ & \quad \sum_{\text{operators at } S_i} \overline{r_o} * T \leq C, \end{aligned} \tag{3}$$

$$\sum_{\text{operators at } S_i} \overline{r_o} * M \leq B, \tag{4}$$

$$\sum_{\text{operators at } S_j} \sum_{\text{producers at } S_i} \overline{r_o(s)} * N \leq Q_{i,j}, \tag{5}$$

Assuming each operator (suppose in total $|o|$ operators) can be replicated at most k replicas, we have to consider

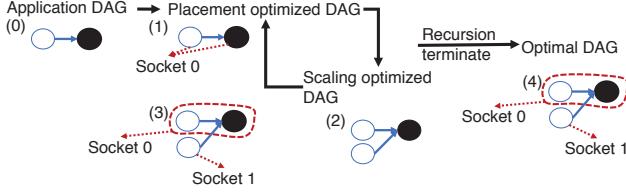


Figure 4: RLAS Optimization example.

in total $k^{|\alpha|}$ 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 ($n \geq |\alpha|$). Such a large solution space makes brute-force unpractical.

4 OPTIMIZATION ALGORITHM DESIGN

We propose a novel optimization paradigm called *Relative-Location Aware Scheduling* (RLAS) to optimize replication level and operator placement at the same time guided by our performance model. The key to optimize replication configuration of a stream application is to remove bottlenecks in its streaming pipeline. As each operator's throughput and resource demand may vary in different placement plans, removing bottlenecks has to be done together with placement 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 is determined during placement optimization. Specifically, the operator that is overfed is defined as bottleneck (see Case 1 in Section 3.1). Figure 4 shows an optimization example of a simple application consisting of two operators. The initial execution plan with no operator replication is labelled with (0). First, RLAS optimizes its placement (labelled with (1)) with *placement algorithm*, which also identifies bottleneck operators. The operators' placement to CPU sockets are indicated by the dotted arrows in the Figure. Subsequently, it tries to increase the replication level of the bottleneck operator, i.e., the hollow circle, with *scaling algorithm* (labelled with (2)). 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.

The details of scaling and placement optimization algorithms are presented in Appendix C. In the following, we discuss how the Branch and Bound (B&B) based technique [43] is applied to solve our placement optimization problem assuming operator replication is given as input. We focus on discussing our bounding function and proposed heuristics that improve the searching efficiency.

Branch and Bound Overview. B&B systematically enumerates a tree with nodes representing 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. 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. 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.

Algorithm complexity: 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 [41]. In order to further reduce the complexity of the problem, heuristics have to be applied.

The bounding function. 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. 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.

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 may violate some constraints. 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 2) to be 0 for those operators remaining to be scheduled.

The branching heuristics. We introduce the following three heuristics that work together to significantly reduce the solution space.

1) *Collocation heuristic:* 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. Specifically, the algorithm considers a list of *collocation* decisions involving a pair of directly connected producer and consumer. During the searching process, collocation decisions are gradually removed from the list once they become 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:* 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. 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. Furthermore, in case that there are multiple sockets that it can achieve maximum output rate, we only consider the socket with the least remaining resource. If there are multiple equal choices, we only branch to one of them to reduce problem size.

3) *Compress graph:* The third provides a mechanism to tune a trade-off between optimization granularity and searching space. Under a large replication level setting, the execution graph becomes very large and the searching space is huge. We compress the execution graph by grouping multiple replicas of an operator (denoted by *compress ratio*) into a single large instance that is scheduled together. Essentially, the compress ratio represents the tradeoff between the 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

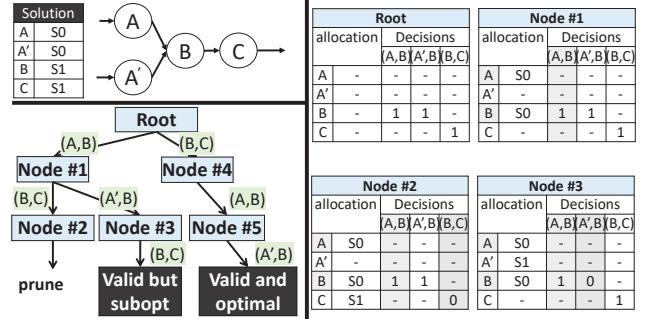


Figure 5: Placement optimization at runtime. Light colored rectangle represents a live node that still violates resource constraints. Dark colored rectangle stands for a solution node contains a valid plan.

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 represents the operator 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 the 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), it can be pruned safely. 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.

5 BRISKSTREAM SYSTEM

Applying RLAS to existing DSPSs (e.g., Storm, Flink, Heron) is insufficient to make them scale on shared-memory multicore architectures. As they are not designed for multicore environment [54], much of the overhead come from the inherent distributed system designs.

We integrate RLAS optimization framework into BriskStream¹, a new DSPS supporting the same APIs as Storm and Heron. More implementation details of BriskStream are given in Appendix A. According to Equation 1, both T^e and T^f shall be reduced in order to improve output rate of an operator and subsequently improve application throughput. In the following, we discuss two design aspects of BriskStream that are specifically optimized for shared-memory architectures that reduce T^e and T^f significantly. We also discuss some limitations in Section 5.3.

5.1 Improving Execution Efficiency

Compared with distributed DSPSs, BrickStream eliminates 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). Those unnecessary components (although not involved during execution) bring many conditional branch instructions and results in large instruction footprint [54]. Furthermore, we carefully revise the critical execution path to avoid unnecessary/duplicate temporary object creations. 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.

5.2 Improving Communication Efficiency

Most modern DSPSs [4, 5, 54] employ buffering strategy to accumulate multiple tuples and send them in batches to improve the application throughput. BriskStream follows the similar idea of buffering output tuples, but accumulated tuples are combined into one “jumbo tuple” (see the example in Appendix A). This approach has several benefits for scalability. First, 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 a single insertion to the communication queue and effectively

¹The source code of BriskStream will be publicly available at <https://github.com/ShuhaoZhangTony/briskstream>.

amortizing the insertion overhead. As a result, both T^e and T^f are significantly reduced.

5.3 Discussions

To examine the maximum system capacity, we assume input stream ingress rate (I) is sufficiently large and keeps the system busy. Hence, the model instantiation and subsequent execution plan optimization are conducted at the same *over-supplied* configuration. In practical scenarios, stream rate as well as its characteristics can vary over time, and application needs to be re-optimized in response to workload changes [20, 26, 49]. To adapt our optimizations to dynamic scenarios, we plan to study simple heuristic algorithms such as round-robin or traffic-minimization allocation [52, 54].

6 EVALUATION

Our experiments are conducted in following aspects. First, our proposed performance model accurately predict the application throughput under different execution plans (Section 6.2). Second, BriskStream significantly outperforms existing open-sourced DSPSs on multicores (Section 6.3). Third, our RLAS optimization approach performs significantly better than competing techniques (Section 6.4). We also show in Section 6.5 the relative importance of BriskStream’s optimization techniques.

6.1 Experimental Setup

We pick four common applications from the previous study [54] 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. More application settings can be found in Appendix B.

To examine the maximum system capacity under given hardware resources, we tune the input stream ingress rate (I) to its maximum attainable value (I_{max}) to keep the system busy and report the stable system performance². To minimize interference of operators, we use OpenHFT Thread Affinity Library [9] with core isolation (i.e., configure *isolcpus* to avoid the isolated cores being used by Linux kernel general scheduler) to bind operators to cores based on the given execution plan.

Table 2 shows the detailed specification of our two eight-socket servers. We use Server A in Section 6.2, 6.3 and 6.5. We study our RLAS optimization algorithms in detail on different NUMA architectures with both two servers in Section 6.4. NUMA characteristics, such as local and inter-socket idle latencies and peak memory bandwidths, are measured with

²Back-pressure mechanism will eventually slow down spout so that the system is stably running at its best achievable throughput.

Table 2: 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

Intel Memory Latency Checker [7]. These two machines have different NUMA topologies, which lead to different access latencies and throughputs across CPU sockets. The three major takeaways from Table 2 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. In particular, remote memory access bandwidth is similar regardless of the NUMA distance in Server B. In contrast, the bandwidth is significantly lower across long NUMA distance than smaller distance on Server A. Third, there is a significant increase in remote memory access latency from within the same CPU tray (e.g., 1 hop latency) to between different CPU trays (max hops latency) on both servers.

In addition to runtime statistics evaluation, we also report how much time each tuple spends in different components of the system. We classify these components as follows: 1) *Execute* 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. 2) *RMA* 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. 3) *Others* consist of all other time spent in the critical execution path and considered as overhead. Examples include temporary object creation, exception condition checking, communication queue accessing and context switching overhead.

To measure *Execute* and *Others*, we allocate the operator to be collocated with its producer. The time spend in user function per tuple is then measured as *Execute*. We measure the gap between the subsequent call of the function as *round-trip delay*. *Others* is then derived as the subtraction from

Table 3: Average processing time per tuple (T) under varying NUMA distance. The unit is nanoseconds/tuple.

Splitter			Counter		
From-to	Measured	Estimated	From-to	Measured	Estimated
S0-S0(local)	1612.8	1612.8	S0-S0(local)	612.3	612.3
S0-S1	1666.5	1991.1	S0-S1	611.4	665.2
S0-S3	1708.2	1994.9	S0-S3	623.1	665.9
S0-S4	2050.6	2923.7	S0-S4	889.9	837.9
S0-S7	2371.3	3196.4	S0-S7	870.2	888.4

round-trip delay by *Execute*. Note that, the measurement only consists of contiguous successful execution and exclude the time spend in queue blocking (e.g., the queue is empty or full). To measure *RMA* cost, we allocate the operator remotely to its producer and measures the new *round-trip delay* under such configuration. The *RMA* cost is then derived as the subtraction from the new *round-trip delay* by the original *round-trip delay*.

6.2 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 process time per tuple (T) 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. When the input tuple size is large (in case of Splitter), the memory accesses have better locality and the hardware prefetcher helps in reducing communication cost [38]. Another observation 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 across different CPU trays.

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 relative error is defined as $\text{relative_error} = \frac{|R_{\text{meas}} - R_{\text{rest}}|}{R_{\text{meas}}}$, where R_{meas} is the measured application throughput and R_{rest} is the estimated application throughput by our performance model for the same application.

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 throughput of all four applications. It is able to produce the optimal execution plan and predict the relative performance quite accurately.

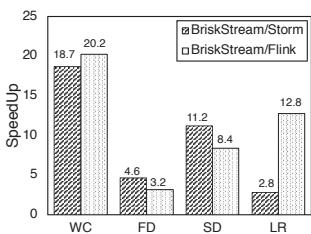


Figure 6: Throughput speedup.

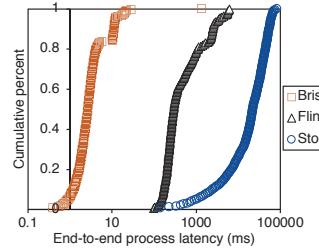


Figure 7: End-to-end latency of WC on different DSPSs.

Table 5: 99-percentile end-to-end latency (ms)

	Brisk Stream	Storm	Flink
WC	21.9	37881.3	5689.2
FD	12.5	14949.8	261.3
SD	13.5	12733.8	350.5
LR	204.8	16747.8	4886.2

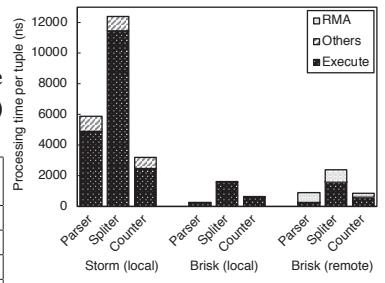
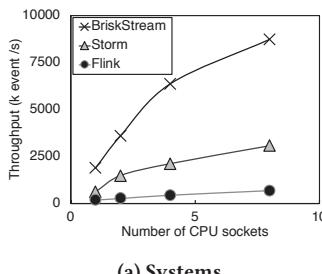
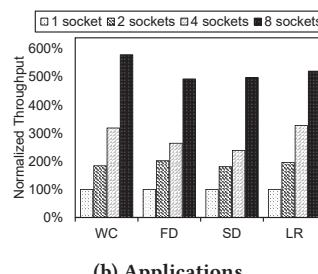


Figure 8: Execution time breakdown.



(a) Systems



(b) Applications

Figure 9: Scalability evaluation.

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

6.3 Evaluation of Execution Efficiency

This section shows that BriskStream significantly outperforms existing DSPSs on shared-memory multicores. We compare BriskStream with two open-sourced DSPSs 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 a single task manager, which shows even worse performance. We also compare BriskStream with StreamBox, a recent single-node DSPS on share-memory multi-core architectures at the end of this section.

Throughput and latency comparison. Figure 6 shows the significant throughput speedup of BriskStream compared to Storm and Flink. Overall, Storm and Flink show comparable throughput 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 operators (implemented as the co-flat map) that merges multiple input streams before feeding to an operator with multi-input streams (commonly found in LR). Neither Storm nor BriskStream has such additional overhead.

Following the previous work [24], we define the end-to-end latency of a streaming workload as the duration between the time when an input event enters the system and the time when the results corresponding to that event is generated. We compare the end-to-end process latency among different DSPSs on Server A. Figure 7 shows the detailed CDF of end-to-end processing latency of WC comparing different DSPSs and Table 5 shows the overall 99-percentile end-to-end processing latency comparison of different applications. The end-to-end latency of BriskStream is significantly smaller than both Flink and Storm.

Per-tuple execution time breakdown. To better understand the source of performance improvement, we show the per-tuple execution time breakdown by comparing BriskStream and Storm. Figure 8 shows the breakdown of all non-source operators of WC, which we use as the example application in this study. 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 RMA.

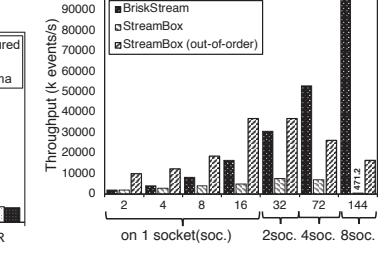


Figure 10: Gaps to ideal. **Figure 11:** Comparing with StreamBox.

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 effectively amortizes 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 ($T^e \ll T^f$). The significant varying processing capability of the same operator when it is under different placement plan reaffirms the necessity of our RLAS optimization.

Another takeaway is that *Execute* in Storm is much larger than *RMAs*, which means $T^e >> T^f$ and NUMA effect may have a minor impact in its plan optimization. In contrast, BriskStream significantly reduces T^e (discussed in Section 5) and the NUMA effect, as a result of improving efficiency of other components, becomes a critical issue to optimize. In the future, on one hand, T^e may be further reduced with more optimization techniques deployed. 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.

Evaluation of scalability on varying CPU sockets. Our next experiment shows that BriskStream scales effectively as we increase the numbers of sockets. RLAS is able to optimize the execution plan under a different number of sockets enabled. Figure 9a shows the better scalability of BriskStream than existing DSPSs on multi-socket servers by taking LR as an example. Unmanaged thread interference and unnecessary remote memory access penalty prevent existing DSPSs from scaling well on the modern multi-sockets machine. We show the scalability evaluation of different applications of BriskStream in Figure 9b. 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 about two 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 10. 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 from Figure 10. First, theoretically removing RMA cost (i.e., “W/o rma”) achieves $89 \sim 95\%$ of the ideal performance, and it hence confirms that the significant increase of RMA cost is the main reason that BriskStream is not able to scale linearly on 8 sockets. Second, we still need to improve the parallelism and scalability of the execution plan to achieve optimal performance even without RMA.

Comparing with single-node DSPS. Streambox [42] is a recently proposed DSPS based on a morsel-driven like execution model – a different processing model to BriskStream. We compare BriskStream with StreamBox using WC as an example. Results in Figure 11 demonstrate that BriskStream outperforms StreamBox significantly regardless of the number of CPU cores used in the system. Note that, StreamBox focuses on solving out-of-order processing problem, which requires more expensive processing mechanisms such as locks and container design. Due to a different system design objective, BriskStream currently does not provide ordered processing guarantee and consequently does not bear such overhead.

For a better comparison, we modify StreamBox to disable its order-guaranteeing feature, denoted as StreamBox (out-of-order), so that tuples are processed out-of-order in both systems. Despite its efficiency at smaller core counts, it scales poorly when multiple sockets are used. There are two main reasons. First, StreamBox relies on a centralized task scheduling/distribution mechanism with locking primitives, which brings significant overhead for more CPU cores. This could be a limitation inherited from adopting morsel-driven execution model in DSPSs – essentially it trades off the reduced pipeline parallelism for lower operator communication overhead, which we defer as a future work to investigate in more detail. Second, WC needs the same word being counted by the same counter, which requires a data shuffling operation in StreamBox. Such data shuffling operation introduces significant remote memory access to StreamBox. We compare their NUMA overhead during execution using Intel Vtune Amplifier [8]. We observe that, under 8 sockets (144 cores), BriskStream issues in average 0.09 cache misses served remotely per k events (misses/k events), which StreamBox’s has 6 misses/k events.

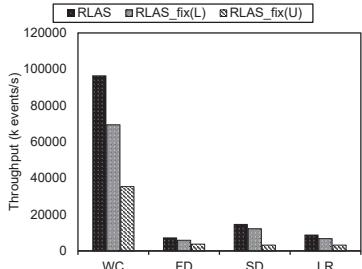


Figure 12: RLAS w/ and w/o considering varying RMA cost.

Table 6: Placement strategies

Name	Placement strategy details
OS	the placement 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 on each CPU socket

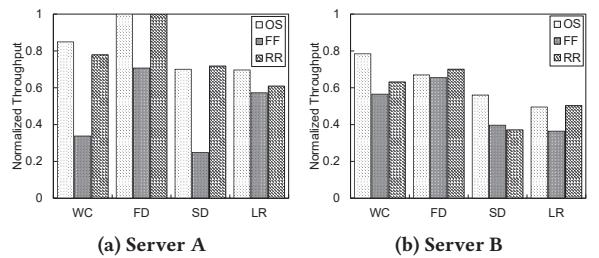


Figure 13: Placement strategy comparison under the same replication configuration.

6.4 Evaluation of RLAS algorithms

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

The necessity of considering varying processing capability. To gain a better understanding of the importance of relative-location awareness, we consider an alternative algorithm that utilizes the same searching process of RLAS but assumes each operator has a fixed processing capability. Such an approach essentially falls back to the original RBO model [51], and is also similar to some previous works [27, 34]. In our context, we need to fix T^f of each operator to a constant value. We consider two extreme cases. First, the lower bound case, namely *RLAS_fix(L)*, assumes each operator pessimistically always includes remote access overhead. That is, T^f is calculated by anti-collocating an operator to all of its producers. Second, the upper bound case, namely *RLAS_fix(U)*, completely ignores RMA, and T^f is set to 0 regardless the relative location of an operator to its producers.

The comparison results are shown in Figure 12. RLAS shows a 19% ~ 39% improvement over *RLAS_fix(L)*. We observe that *RLAS_fix(L)* often results in smaller replication configuration of the same application compared to RLAS and hence underutilizes the underlying resources. This is because it *over-estimates* the resource demand of operators that are collocated with producers. Conversely, *RLAS_fix(U)* *under-estimates* the resource demands of operators that are anti-collocated and misleads the optimization process to involve severely thread interference. Over the four workloads, RLAS shows a 119% ~ 45% improvement over *RLAS_fix(U)*.

Comparing different placement strategies. We now study the effect of different placements under the same replication configuration. In this experiment, the replication configuration is fixed to be the same as the optimized plan generated by RLAS and only the placement is varied under different techniques. Three alternative placement strategies are shown in Table 6. Both FF [52] and RR [44] are enforced to guarantee resource constraints as much as possible. In

case these strategies cannot find any plan satisfying resource constraints, they will gradually relax constraints until a plan is obtained. We also configure external input rate (I) to just overfeed the system on Server A, and using the same I to test on Server B. This allows us to examine the system capacity of different servers. The results are shown in Figure 13. There are two major takeaways.

First, RLAS generally outperforms other placement techniques on both two servers. FF can be viewed as a minimizing traffic heuristic-based approach as it greedily allocates neighbor operators (i.e., directly connected) together due to its topologically sorting step. Several related studies [13, 52] adopt a similar approach of FF in dealing with operator placement problem in the distributed environment. However, it performs poorly, because we find that during its searching for optimal placements, it often falls into “notable-to-progress” situation as it cannot allocate the current 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. 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-socket communication.

Second, RLAS performs generally better on Server B. We observe that Server B is underutilized for all applications under the given testing input loads. This indicates that although the total computing power (aggregated CPU frequency) of Server A is higher, its maximum attainable system capacity is actually smaller. As a result, RLAS chooses to use only a subset of the underlying hardware resource of Server B to achieve the maximum application throughput. In contrast, other heuristic based placement strategies unnecessarily involve more RMA cost by launching operators to all CPU sockets.

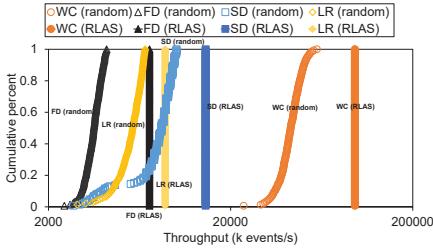


Figure 14: CDF of random plans.

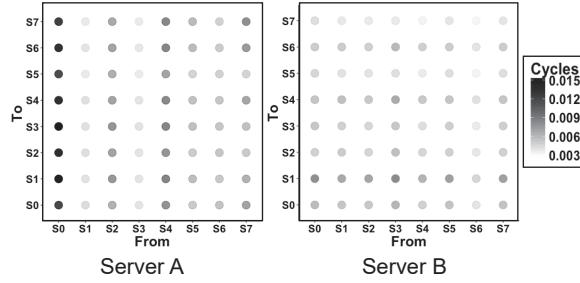


Figure 15: Communication pattern matrices of WC on two Servers.

Table 7: Runtime of the optimization process

r	throughput	runtime (sec)
1	10140.2	93.4
3	10079.5	48.3
5	96390.8	23.0
10	84955.9	46.5
15	77773.6	45.3

Correctness of heuristics. Due to a very large search space, it is almost impossible to examine all execution plans of our test workloads to verify the effectiveness of our heuristics. Instead, we utilize Monte-Carlo simulations by generating 1000 random execution plans, and compare against our optimized execution plan. Specifically, the replication level of each operator is randomly increased until the total replication level hits the scaling limit. All operators (incl. replicas) are then randomly placed. Results of Figure 14 show that none of the random plans is better than RLAS. It demonstrates that random plans hurt the performance in a high probability due to the huge optimization space.

We further observe two properties of optimized plans of RLAS, which are also found in randomly generated plans with relatively good performance. First, operators of FD and LR are completely avoided being remotely allocated across different CPU-tray to their producers. This indicates that the RMA overhead, especially from the costly communications across CPU trays, should be aggressively avoided in these two applications. Second, resources are well utilized for high throughput optimizations in RLAS. Most operators (incl. replicas) end up with being “just fulfilled”, i.e., $\bar{r}_o = r_o = r_i$. This effectively reveals the shortcoming of existing heuristics based approach – maximizing an operator’s performance may be worthless or even harmful to the overall system performance as it may already overfeed its downstream operators. Further increasing its performance (e.g., scaling it up or making it allocated together with its producers) is just a waste of the precious computing resource.

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 15. 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

(S_0) 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.

Varying the compression ratio (r). RLAS allows to compress the execution graph (with a ratio of r) to tune the trade-off between optimization granularity and searching space. We use WC as an example to show its impact as shown in Table 7. Similar trend is observed in other three applications. Note that, a compressed graph contains heavy operators (multiple operators grouped into one), which may fail to be allocated and requires re-optimization. This procedure introduces more complexity to the algorithm, which leads to higher runtime of the optimization process as shown in Table 7. Due to space limitation, a detailed discussion is presented in Appendix D.

6.5 Factor Analysis

To understand the details in the overheads and benefits of various aspects of BriskStream, we show a factor analysis in Figure 16 that highlights the key factors for performance. *Simple* refers to running Storm directly on shared-memory multicore. *-Instr.footprint* refers to BriskStream with much smaller instruction footprint and avoiding unnecessary/duplicate objects as described in Section 5.1. *+JumboTuple* further allows BriskStream to reduce the cross-operator communication overhead as described in Section 5.2. In the first three cases, the system is optimized under *RLAS_fix(L)* scheme without considering *varying RMA cost*. *+RLAS* adds our NUMA aware execution plan optimization as described in Section 3. The major takeaways from Figure 16 are that jumbo tuple design is important to optimize existing DSPSs on shared-memory multicore architecture and our RLAS optimization paradigm

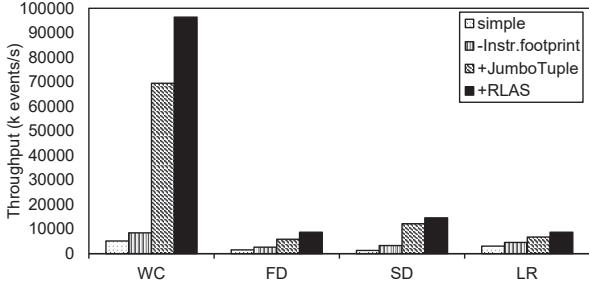


Figure 16: A factor analysis for BriskStream. Changes are added left to right and are cumulative.

is critical for DSPSs to scale different applications on modern multicore environments addressing NUMA effect.

7 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 [50, 53]. There have been studies on optimizing the instruction cache performance [28, 55], the memory and cache performance [11, 15, 16, 29], many-core parallelism in a single chip [22, 33] and NUMA [27, 39, 40, 46–48]. Psaroudakis et al. [47, 48] developed an adaptive data placement algorithm that can track and resolve utilization imbalance across sockets. However, it does not solve the problem we address. In particular, the placement strategy such as RR balances resource utilization among CPU sockets, but shows suboptimal performance in our experiments. Leis et al. [39] proposed a novel morsel-driven query execution model which integrates both NUMA-awareness and fine grained task-based parallelism. A similar execution model is adopted in StreamBox [42], which we compared in our experiments. The results confirm the superiority of BriskStream in addressing NUMA effect.

Data stream processing systems (DSPSs): DSPSs have attracted a great amount of research effort. A number of systems have been developed, for example, TelegraphCQ [21], Borealis [10], IBM System S [32] and the more recent ones including Storm [5], Flink [4] and Heron [36]. However, most of them targeted at the distributed environment, and little attention has been paid to the research on DSPSs on the modern multicore environment. A recent patch on Flink [3] tries to make Flink a NUMA-aware DSPS. However, its current heuristic based round-robin allocation strategy is not sufficient to make it scale on large multicores as our experiment shows. Previous work [54] gave a detailed study on the insufficiency of two popular DSPSs (i.e., Storm and Flink) running on modern multi-core processors. It proposed a heuristic-based

algorithm to deploy stream processing on NUMA-based machines. However, the heuristic does not take relative-location awareness into account. It may not always be efficient for different workloads. In contrast, BriskStream provides a model-guided approach that automatically determines the optimal operator parallelism and placement addressing the NUMA effect. SABER [35] focuses on efficiently realizing computing power from both CPU and GPUs. Streambox [42] 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 DSPSs on shared-memory multicore architectures.

Execution plan optimization: Both operator placement and operator replication are widely investigated in the literature under different assumptions and optimization goals [37]. In particular, many algorithms and mechanisms [13, 18, 19, 34, 44, 52] are developed to allocate (i.e., schedule) operators of a job into physical resources (e.g., compute node) in order to achieve a certain optimization goal, such as maximizing throughput, minimizing latency or minimizing resource consumption, etc. Due to space limitation, we discuss them in Appendix E. Based on similar ideas from prior works, 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.

8 CONCLUSION

We have introduced BriskStream, a new data stream processing system with a new streaming execution plan optimization paradigm, namely Relative-Location Aware Scheduling (RLAS). BriskStream successfully scales stream computation towards hundred of cores under NUMA effect. The experiments on eight-sockets machines confirm that BriskStream significantly outperforms existing open-sourced DSPSs up to an order of magnitude.

9 ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their valuable comments. This work is supported by a MoE AcRF Tier 2 grant (MOE2017-T2-1-122) and an NUS startup grant in Singapore. Jiong He is supported by the National Research Foundation, Prime Ministers Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme. Chi Zhou's work is partially supported by the National Natural Science Foundation of China under Grant 61802260 and the Guangdong Natural Science Foundation under Grant 2018A030310440.

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. HP ProLiant DL980 G7 server with HP PREMA Architecture Technical Overview. <https://community.hpe.com/hpeb/attachments/hpeb/itrc-264/106801/1/363896.pdf>
- [7] 2018. Intel Memory Latency Checker, <https://software.intel.com/articles/intel-memory-latency-checker>.
- [8] 2018. Intel VTune Amplifier. <https://software.intel.com/en-us/intel-vtune-amplifier-xe>.
- [9] 2018. OpenHFT. <https://github.com/OpenHFT/Java-Thread-Affinity>
- [10] 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*.
- [11] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 2009. DBMSs on a Modern Processor: Where Does Time Go?. In *VLDB*.
- [12] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *ICDE*.
- [13] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. 2013. Adaptive Online Scheduling in Storm. In *DEBS*.
- [14] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. 2013. Scale-up vs scale-out for hadoop: Time to rethink?. In *SoCC*.
- [15] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsü. 2013. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *ICDE*.
- [16] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the new. Bottleneck: Memory Access. In *VLDB*.
- [17] Surendra Byna, Xian He Sun, William Gropp, and Rajeev Thakur. 2004. Predicting memory-access cost based on data-access patterns. In *ICCC*.
- [18] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2016. Optimal Operator Placement for Distributed Stream Processing Applications. In *DEBS*.
- [19] 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.*
- [20] V. Cardellini, M. Nardelli, and D. Luzi. 2016. Elastic Stateful Stream Processing in Storm. In *HPCS*.
- [21] Sirish Chandrasekaran and et al. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*.
- [22] Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. 2017. A Study of Main-Memory Hash Joins on Many-core Processor: A Case with Intel Knights Landing Architecture. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*.
- [23] Alvin Cheung, Owen Arden, Samuel Madden, and Andrew C. Myers. 2013. Speeding up database applications with Pyxis. In *SIGMOD*.
- [24] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive Stream Processing using Dynamic Batch Sizing. *SOCC* (2014).
- [25] 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*.
- [26] B. Gedik, S. Schneider, M. Hirzel, and K. L. Wu. 2014. Elastic Scaling for Data Stream Processing. *TPDS* (2014).
- [27] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. 2014. Deployment of Query Plans on Multicores. In *VLDB*.
- [28] Stavros Harizopoulos and Anastassia Ailamaki. 2006. Improving Instruction Cache Performance in OLTP. *TODS* (2006).
- [29] Bingsheng He, Qiong Luo, and B. Choi. 2005. Cache-conscious automata for XML filtering. In *ICDE*.
- [30] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* (2014).
- [31] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng. 2010. In *IPDPS*.
- [32] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. 2006. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *SIGMOD*.
- [33] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. 2015. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. VLDB Endow.* (2015).
- [34] Khandekar, Rohit and Hildrum, Kirsten and Parekh, Sujay and Rajan, Deepak and Wolf, Joel and Wu, Kun-Lung and Andrade, Henrique and Gedik, Buğra. 2009. COLA: Optimizing Stream Processing Applications via Graph Partitioning. In *Middleware*.
- [35] Alexandros Koliosius, 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*.
- [36] 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*.
- [37] Geetika T. Lakshmanan, Ying Li, and Robert E. Strom. 2008. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Computing* (2008).
- [38] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.* (2012).
- [39] 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*.
- [40] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *CIDR*.
- [41] Devroye Luc and Zamora Cura Carlos. 1999. On the Complexity of Branch-and Bound Search for Random Trees. *Random Struct. Algorithms* (1999).
- [42] 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*.
- [43] 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).
- [44] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. 2015. R-Storm: Resource-Aware Scheduling in Storm. In *Middleware*.
- [45] Achille Peternier, Daniele Bonetta, Walter Binder, and Cesare Pautasso. 2011. Overseer: low-level hardware monitoring and management for

- Java. In *PPSJ*
- [46] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. 2012. OLTP on Hardware Islands. *PVLDB* (2012).
 - [47] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2015. Scaling up concurrent main-memory column-store scans: towards adaptive NUMA-aware data and task placement. In *VLDB*.
 - [48] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc of the VLDB Endow.* (2016).
 - [49] Scott Schneider, Joel Wolf, Kirsten Hildrum, Rohit Khandekar, and Kun-Lung Wu. 2016. Dynamic Load Balancing for Ordered Data-Parallel Regions in Distributed Streaming Systems. (2016).
 - [50] 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).
 - [51] Stratis D. Viglas and Jeffrey F. Naughton. 2002. Rate-based Query Optimization for Streaming Information Sources. In *SIGMOD*.
 - [52] J. Xu, Z. Chen, J. Tang, and S. Su. 2014. T-Storm: Traffic-Aware Online Scheduling in Storm. In *ICDCS*.
 - [53] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *TKDE* (2015).
 - [54] 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*.
 - [55] Jingren Zhou and Kenneth A. Ross. 2004. Buffering Database Operations for Enhanced Instruction Cache Performance. In *SIGMOD*.

A IMPLEMENTATION DETAILS

BriskStream shares many similarities to existing DSPSs including pipelined processing and operator replication designs. To avoid reinventing the wheel, we reuse many components found in existing DSPSs such as Storm, Heron and Flink, notably including *API design, application topology compiler, pipelined execution engine with communication queue and back-pressure mechanism*. In contrast, BrickStream embrace various designs that are suitable for shared-memory multicore architectures. For example, Heron has an operator-per-process execution environment, where each operator in an application is launched as a dedicated JVM process. In contrast, an application in BriskStream is launched in a JVM process, and operators are launched as Java threads inside the same JVM process, which avoids cross-process communication and allows the pass-by-reference message passing mechanism. Specifically, tuples produced by an operator are stored locally, and pointers as reference to tuple are inserted into a communication queue. Together with the jumbo tuple design, reference passing delay is minimized and becomes negligible.

Figure 17 presents an example job (WC) of BriskStream. Each operator (or the replica) of the application is mapped to one *task*. The task is the basic processing unit in BrickStream (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 partitioning. Furthermore, each task maintains output buffers for each of its consumers, where jumbo tuples are formed accordingly.

B APPLICATION SETTINGS

In this section, we discuss more settings of the testing applications in our experiment. We have shown the topology of WC in Figure 2. Figure 18 shows the topology of the other three applications. More details about the specification about them can be found in the previous paper [54].

The selectivity is affected by both input workloads and application logic. Parser and Sink have a selectivity of one in all applications. Splitter has a output selectivity of ten in WC. That is, each input sentence contains 10 words. Counter has an output selectivity of one, thus it emits the counting results of each input word to Sink. Operators have an output selectivity of one in both FD and SD. That is, we configure that a signal is passed to Sink in both predictor operator of FD and Spike detection operator of SD regardless of whether detection is triggered for an input tuple. Operators may contain multiple output streams in LR. If an operator has only one output stream, we denote its stream as the *default* stream. We show the selectivity of each output stream of them of LR in Table 8.

C ALGORITHM DETAILS

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 their runtime (Appendix D). We further elaborate how our optimization paradigm can be extended with other optimization techniques (Appendix D).

Algorithm 1 illustrates our scaling algorithm based on topological sorting. Initially, we set replication level of each operator to be one (Lines 1~2). The algorithm proceeds with this and it optimizes operator placement with Algorithm 2 (Line 6). Then, it stores the current plan if it ends up with better performance (Lines 7~8). At Lines 11~19, 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

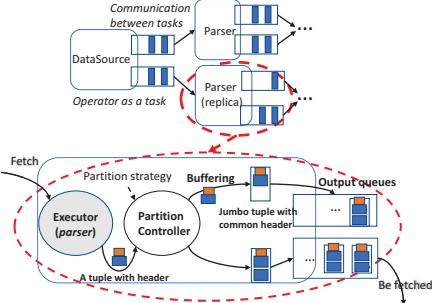


Figure 17: An example job in BriskStream.

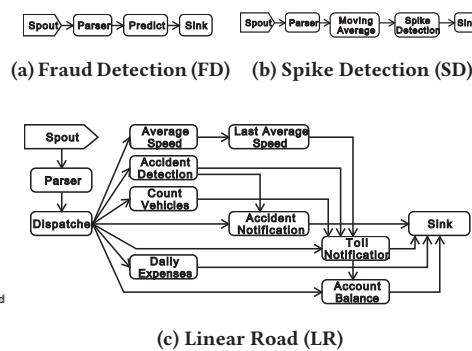


Figure 18: Topologies of other three applications in our benchmark.

Table 8: Operator selectivity of LR

Operator Name	Input streams	Output streams	Selectivity
Dispatcher	default	position_report	≈0.99
		balance_stream	≈0.0
		daily_exp_request	≈0.0
Avg_speed	position_report	avg_stream	1.0
Las_avg_speed	avg_stream	las_stream	1.0
Accident_detect	position_report	detect_stream	0.0
Count_vehicle	position_report	counts_stream	1.0
Accident_notify	detect_stream, position_report	notify_stream	0.0
Toll_notify	detect_stream	toll_notify_stream	0.0
	position_report	toll_notify_stream	1.0
	counts_stream	toll_notify_stream	1.0
	las_stream	toll_notify_stream	1.0
Daily_expen	daily_exp_request	default	0.0
Account_balance	balance_stream	default	0.0

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 a new iteration to look for a 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 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
12  start from sink
13  foreach list ∈ sortedLists do
14    foreach Operator o ∈ list do
15      if o is bottleneck then
16        p.parallelism ← try to increase the replication
17        level of o by  $\lceil \frac{r_i}{r_o} \rceil$ ;
18        if successfully increased p.parallelism then
19          return Searching(p) // start another
                           iteration
20      else
21        return opt
22
23  return opt;

```

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 (Lines 1~5). At Lines 7~14, the algorithm explores the current node. If it has better bounding value than the current solution, we update the solution node (Lines 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 (Lines 15~32) illustrates how the searching process branches and generates children nodes to explore. For each collocation decision in the current node (Line 16), we apply the *best fit heuristic* (Lines 17~23) and one new node is created. Otherwise, at Lines 25~27, we have to create new nodes for each possible way of placing the two operators (i.e., up to $(m)_1 * (n)_1$). 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.

D DISCUSSIONS ON OPTIMIZATION PROCESS

We have made some interesting 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 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 gain. *Second*, the placement algorithm may fail to find any valid plan as it is not able to allocate one or more operators due to resource constraints, which causes scaling algorithm to terminate. It is interesting to note

Algorithm 2: B&B based placement optimization

```

Data: Stack stack // stores 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
// Initialization
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);
7 while !isEmpty(stack)// Branch and Bound process
8 do
9     e ← Pop(stack);
10    if e.R > solution.R then
11        if e.validOperators == totalOperators then
12            | solution ← e;
13        else
14            | Branching(e);
15    else
16        // the current node has worse bounded value
17        // than solution, and can be safely pruned.
18
19 Function Branching(e):
20     Data: Node[] children
21     foreach pair of  $O_s$  and  $O_c$  in e.decisions do
22         if all predecessors of them are already allocated except  $O_s$  to  $O_c$ 
23             then
24                 #newAllocate ← 2;
25                 if they can be collocated into one socket then
26                     create a Node n with a plan collocating them to one
27                     socket;
28                 else
29                     create a Node n with a plan separately allocating
30                     them to two sockets;
31                     add n to children;
32             else
33                 #newAllocate ← 1;
34                 foreach valid way of placing  $O_s$  and  $O_c$  do
35                     | create a new Node and add it to children;
36
37     foreach Node c ∈ children// update in parallel
38     do
39         | c.validOperators ← e.validOperators + #newAllocate;
40         | c.R ← BoundingFunction(c.plan);
41     PushAll(stack, children);

```

that this *may not* indicate the saturation of the underlying resources but the operator itself is too coarse-grained. The scaling algorithm can, instead of termination (in Line 10 Algorithm 1), 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. This procedure, however, introduces more complexity to the algorithm.

Optimization runtime. The placement optimization problem is difficult to solve as the solution space increases rapidly with increased replication level configurations. Besides the three proposed heuristics, we also apply a

list of optimization techniques 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) 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) the algorithm is highly optimized for higher concurrency (e.g., concurrently generate branching children nodes). Overall, the placement algorithm needs less than 5 seconds to optimize placement for a large DAG, and the entire optimization usually 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 usually runs for a long time, the overhead of generating a plan is not included in our measurement.

Extension with other optimization techniques. A number of optimization techniques are available in the literature [27, 30]. Many of them can be potentially applied to further improve the performance of BriskStream. Taking operator fusion as an example, which trades communication cost against pipeline parallelism and is in particular helpful if operators share little in common computing resource. Our performance model is general enough such that it can be extended to capture other optimization techniques.

E MORE RELATED WORK

Aniello et al. [13] 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 [52] dynamically assigns/reassigns operators according to runtime statistics in order to minimize inter-node and inter-process traffic while ensuring load balance. R-Storm [44] 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. Cardellini et al. [18, 19] propose a general mathematical formulation of the problem of optimizing operator placement for distributed data stream processing. Those approaches 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.