Revision Summary of SIGMOD-52

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

October 29, 2018

Dear Reviewers,

We are very thankful to the reviewers for their thorough evaluation reports with insights. We have revised our draft accordingly and are resubmitting our revised draft together with this revision summary for further review. The references are to the revised draft.

Major changes are summarized as follows.

- We have added the discussion of comparing our system to the previous work to better motivate the necessity of a relative-location aware scheduling (RLAS) algorithm.
- We have added more explanation on the instantiation of the performance model as well as the design philosophy of the optimization algorithms.
- We have added the performance comparison to StreamBox, a recently proposed streaming system on shared-memory multi-cores, and demonstrates the superiority of our system addressing NUMA effect.
- We have added the performance comparison to an alternative algorithm (RLAS_fix) that utilizes the same searching process of RLAS but assuming processing capacity of each operator does not vary under different execution plans.
- We have added more explanation on the insights of optimizing our workloads, which may shed light on future research on the concerned optimization problem.

Sincerely,

Authors of SIGMOD-52

Detailed Response to Reviewers

Notation: *C (Reviewer's comments)* and **R (Our response)**.

Meta Reviewer

C: The reviewers felt the papers has solid ideas but needs more work to make it into the conference. Please address the extensive comment and suggestions in the reviews.

R: Thanks for your coordination in the review process. We have carefully addressed the reviews and made major changes as stated above to satisfy all the reviewers.

Reviewer: 1

C1-1: The results are not a surprise. NUMA and multi-core parallelism has been a problem in many systems. Taking a platform like Storm or Flink which have been designed for parallelism at the cluster level and show that they do not do well at the multi-core level is not a groundbreaking discovery. The same can be said of the results. Any strategy that takes into account the interference problems that occur in a multi-core machine will beat a deployment ignoring them.

R1-1: Thanks for your comments. 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]. Witnessing the emergence of modern commodity machines with massively parallel processors, researchers and practitioners find shared-memory multicore architectures an attractive streaming platform [42, 55]. However, fully exploiting the computation power delivered by multicore architectures can be challenging. Prior studies [55] have shown that existing DSPS underutilize the underlying complex hardware micro-architecture and especially show poor scalability due to the unmanaged resource competition and unaware of non-uniform memory access (NUMA) effect. In this paper, we spot that the key challenge of optimizing 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). The extensive experimental results confirm the superiority of our solution.

C1-2: The servers used for the evaluation are very large and not the usual 2 to 4 socket platforms. It would be important to show from which number of cores the results presented in the paper matter or whether the advantages can be demonstrated only on very large machines. experiments showing the results for a varying number of CPU sockets should be added.

R1-2: Thanks for your comments. We have addressed the review by following experiments. We have added scalability evaluation comparing BriskStream to Storm, Flink and StreamBox. The results are included in Figure 8 and 11 of Section 6.3, which demonstrate that BriskStream outperforms those DSPSs significantly regardless of number of CPU cores used in the system. StreamBox [42] 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. The comparison results in Figure 11 show that StreamBox (out-of-order) performs well when the number of working threads are small, but it scales poorly when more than one CPU socket being used. BriskStream shows comparable performance with StreamBox (out-of-order) when two CPU sockets being used, and surpass it significantly afterwards.

C1-3: A common problem in these papers is that the baseline does nothing (see point W1). As a result, even the simplest of strategist to optimize the system will win over an un-optimized system. This makes it very difficult to determine whether the strategy proposed in the paper is the best or one out of many. Is there any way to come up with some ideal performance target and see how close one gets to it? After all, in some experiments, the OS placement seems to do a good enough job.

R1-3: Thanks for your comments. We have addressed the review by following experiments.

First, 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 9 of the revised draft. Theoretically removing RMA cost (i.e., ``W/o rma``) achieves 89~95% of the ideal performance, and it hence confirms that the significant increase of RMA cost is the main reason that BriskStream is not able not able to scale linearly on 8 sockets.

Second, for placement strategies evaluation, we have kept the replication configuration to be the same among different placement strategies in evaluation. We have made this point clear in the revised draft. For simple structure application like FD, OS performs fairly well, but it becomes suboptimal for more complex applications. FF and RR are not aiming at maximizing the application throughput and hence cannot address our problem. Instead, they aim to minimize cross-operator traffic (FF) or achieve better resource balancing among CPU sockets (RR). The same issue exists in other prior studies that have a different optimization goal from our work.

Furthermore, RLAS provides a model-guided approach that automatically determines the optimal operator parallelism and placement addressing the NUMA effect. This allows RLAS to decide how to smartly utilize the underlying hardware resource to achieve optimal application performance instead of blindly trying to utilize all. For example, under a fixed and small external input rate, only utilizing a subset of CPU sockets lead to maximum throughput. This may require tedious tuning process in existing heuristic approaches to determine the optimal configuration. We have added more discussions regarding to this issue in "Comparing different placement strategies" of Section 6.4 of the revised draft.

C1-4: Overall the paper is a strong engineering effort and the integration in Storm a major plus. I am less sure about the intellectual contribution for lack of a proper baseline. These systems have been designed for distributed execution (mainly to address the I/O problem in big data applications). The paper considers them in a single node scenario. That is fine (it has been proposed before) but still leaves the question open whether it would make more sense to run the same computation in two or three smaller computers. In a single box, no matter how many cores there are, the I/O problem will become very serious. The paper should comment on this and explain the scenarios that they have in mind.

R1-4: Thanks for your comments. We target at applications with high throughput and low latency demand, and we assume external input stream ingestion rate (*I*) to the system is sufficiently large without I/O being a performance issue. Subsequently, our optimization problem aims at maximizing application throughput. We have added more experimental results in Section 6.3 of the revised draft.

Reviewer: 2

C2-1: There is no analysis on the complexity of the branch and bound search algorithm.

R2-1: Thanks for your comments. We gave an analysis of the complexity of directly applying the branch and bound search algorithm in Section 4 that motivates three heuristics to further reduce searching space. We have also added an overhead experiment regarding to tuning compression ratio (our third heuristics) in Section 6.4. We leave detailed theoretical analysis of the concerned optimization problem as future work to explore.

C2-2: There is no discussion on the theoretical properties of the output of the b&b algorithm. How far from the optimal solution will the results be?

R2-2: Thanks for your comments. We resolve this concern experimentally. Due to a very large search space, it is almost impossible to examine all execution plans of our test workloads to verify the correctness of our heuristics. Instead, we study the optimality of our algorithm experimentally by utilizing Monte-Carlo simulations. We have also added the experiments of tuning the compression ratio (one of the heuristics we employed). We have made these points clear in Section 6.4 of the revised draft.

C2-3: I am not convinced that the presented performance model required modifications to the RBO framework. It resembles a standard queuing theory where the arrival rate is a function of the output rate of the source operator and the number of hops distance between nodes.

R2-3: Thanks for your comments. Our model definition is related to the execution model. BriskStream employs a pass-by-reference execution model. That is, tuples produced by an operator are simply stored locally (instead of pushing forward by itself), and only the pointer, as reference, to the tuple is inserted into a communication queue. The reference passing delay is negligible. Hence, *r_i* of an operator is simply *r_o* of the corresponding producer in our model. Conversely, upon obtaining the reference, operator needs to read the actual data (referencing by pointers) during its processing, where the actual data accessing delay depends on NUMA distance of it and its producer. The additional effort spent in accessing data remotely increases fetch cost, which consequently slows down the processing rate (and output rate) of the operator. We have made these points clear in Section 3.1 of the revised draft.

C2-4: The paper reads like a follow up on the ICDE study. But even then it is unclear how the optimizations and algorithm explained here relate to the optimizations used there. Hence a better comparison and discussion with respect to state-of-the-art will significantly strengthen the paper.

R2-4: Thanks for your comment. Previous work [55] gave a detailed study on the insufficiency of two popular DSP systems (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 hence requires tedious tuning process and 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. We have made these points clear in Section 7.

C2-5: One of the biggest problems of the submission is the lack of discussion about the complexity and optimality of the deployment algorithm. The first aspect is important to address so that we understand how the algorithm will behave with different properties of the stream query (number of operations, complexity of the operators, complexity of the data-dependency graph, etc.). The second is important because the algorithm gives out an approximate solution, and it is always good to have a theoretical boundaries on how far off the optimal solution we can be.

R2-5: Thanks for your comments. We have addressed this review in R2-1 and R2-2.

C2-6: The second concern I have is related to the performance modelling of the operators. More specifically, when you estimate r_o you mention that the processing rate of an operator depends on the varying input data fetch time. I am not convinced that this is the case. The processing rate of an operator should not depend on the input data stream. The arrival rate (input rate), however, yes. In this case, the arrival rate is a function of (1) the output rate of the upstream operator and (2) the number of hops distance between the nodes where the two operators are placed.

R2-6: Thanks for your comments. We have addressed this review in R2-3.

C2-7: The third major concern is how this paper relates to the ICDE study by Zhang et al. which revisits the design of DSPs on multicore systems. It reads as a follow-up that fixes the inefficiencies that were discovered for the popular *stream engines, but sadly it does not discuss or compare against the optimizations done in that paper. This is clearly the most related work to the submission and deserves a more thorough differentiation and discussion.*

R2-7: Thanks for your comments. We have addressed this review in R2-4.

C2-8: Provide more details for the servers that were used in the experiments. The short description provided in section 2.1. is insufficient. Also make sure to provide references where needed. What is XNC?

R2-8: We are sorry for the confusion. XNC refers to eXternal Node Controller [6] that interconnects upper and lower CPU tray (each tray contains 4 CPU sockets). XNC maintains a directory of the contents of each processors cache and significantly reduces remote memory access latency. We have made these points clear in Section 2.1.

C2-9: On page 4, where you discuss the inputs to the model you use an unlabeled forward reference to the optimization algorithm, which is not introduced to the reader yet. Please fix it.

R2-9: Thanks for your comments. We want to indicate that the algorithm relies on the performance model to guide the optimization process. It's a redundant information so we have removed it from the figure.

C2-10: It is OK to say that for the sake of simplicity of presentation the model assumes that the selectivity of the operators is 1, but you need to discuss the implications of this assumption in one of the later sections of the paper.

R2-10: Thanks for your comments. Selectivity affects output rate of an operator, and subsequently affects the overall application throughput. The execution plan optimization needs to incorporate such information during its optimization process. We have added more discussion regarding to selectivity of testing workloads in Appendix B of the revised draft.

C2-11: Please be more specific when discussing the benefits of the NUMA-awareness that you added to the RBO. It would be nice if in your evaluation you compare against the predictions done by the original framework.

R2-11: Thanks for your comments. The original RBO framework assumes processing capability of an operator is predefined and independent of different execution plans. 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 approach essentially falls back to the original RBO model, and is also similar to some previous works from a high-level point of view [26,33]. The comparison results are included in Section 6.4. The results show that such a static approach either *over-estimates* resource demand of operators that results in resource underutilization or *under-estimates* resource demand that results in severely thread interference. Both leads to suboptimal performance.

C2-12: When you describe the instantiation of the model, you place all the operators on a single socket. Can you fit all of them in a single socket and what happens if you cannot? What happens when you have more complex query plans? What are non-relevant operators? I am also curious on how you avoid to problem of resource interference when co-scheduling the operators on a single socket with many shared resources. Also what is a queue blocking time here? Please be more specific and provide more details and clarification in this section.

R2-12: Thanks for your comments. To prevent 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 the 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 our execution plan optimization (RLAS) 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 have added more discussion regarding this matters in Section 3.1 of the revised draft, where we discuss model instantiation.

C2-13: Please rename section 3.2 to something more descriptive.

R2-13: Thanks for your comments. We have renamed it to Problem Formulation.

C2-14: How do you determine the values of C, B, and Qi,j? Do you perform microbenchmarks using a particular tool? Similarly with M and T? Do you get the from the performance model? I am concerned that this is not a scalable approach given the diversity of operators (exacerbated when supporting user defined functions).

R2-14: Thanks for your comments. 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]). Those parameters only need to be profiled once. For operator specifications including M and T, they need to be profiled as we discussed in R2-12. We have provided a better categorization and discussion of those terminologies in Section 3.1 of the revised draft.

C2-15: On page 8, you say that in the implementation of the algorithm you set the ratio to be 5, but you do not elaborate on why that makes a good trade-off?

R2-15: Thanks for your comments. We use WC as an example to show its impact as shown in Table 7 of the revised draft. Similar trend is observed in other three applications. More discussions are presented in Section 6.4.

C2-16: How do you measure the time breakdown of an operator for execution time, RMA and others?

R2-16: Thanks for your comments. Those are measured by recording corresponding timestamp. To measure "Execute" and "Others", we allocate the operator together with its producer. The time spent 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 "round-trip delay" by "Execute" that represents additional overhead. Note that, the measurement only consists of contiguous successful execution and exclude the time spent in queue blocking (e.g., the queue is empty or full). To measure "RMA cost", we allocate the operator remotely to its producer and measure 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". We have added more explanations regarding to this issue in Section 6.1, where we discuss experimental setup.

C2-17: The result discussion/analysis provided in Section 5.2. is not really clear. What do you mean by a more regular address?

R2-17: We are sorry for the confusions. We mean that when the input tuple size is large (in case of Splitter), the memory accesses have better locality.

Reviewer: 3

C3-1: Lack of comparison with other optimized scale-up systems. The only competitors are Storm and Flink which are not really built for scaling up on multicores. A comparison with StreamBox would have be very useful.

R3-1: Thanks for your comments. We have added the performance comparison to StreamBox, a recently proposed single-node DSPS, and demonstrates the superiority of our system addressing NUMA effect. The results are included in "Comparing with single-node DSPS" part of Section 6.3. In this paper, we aim to demonstrate the importance of relative-location awareness (the key motivation of RLAS) during streaming execution plan optimization under NUMA effect. We hence leave comparison to other related systems as future work to explore.

C3-2: More targeted evaluation of RLAS with more sophisticated alternatives in addition to just FF and round robin.

R3-2: Thanks for your comments. We have addressed this review in the following aspects.

First, 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 approach essentially falls back to the original RBO model, and is also similar to some previous works in a high level point of view [26,33]. It leads to suboptimal performance for different workloads and in varying NUMA architectures. The comparison results are included in Section 6.4. The results show that such a static approach either over-estimates resource demand of operators that results in resource underutilization or under-estimates resource demand that results in severely thread interference. We hope our study on relative-location aware scheduling could shed lights on other NUMA-aware execution plan optimization research.

Second, for placement strategies evaluation, we have kept the replication configuration to be the same among different placement strategies in evaluation. We have made this point clear in the revised draft. For simple structure application like FD, OS performs fairly well, but it becomes suboptimal for more complex applications. FF and RR are not aiming at maximizing the application throughput and hence cannot address our problem. Instead, they aim to minimize cross-operator traffic (FF) or achieve better resource balancing among CPU sockets (RR). The same issue exists in other prior studies that have a different optimization goal from our work.

Furthermore, RLAS provides a model-guided approach that automatically determines the optimal operator parallelism and placement addressing the NUMA effect. This allows RLAS to decide how to smartly utilize the underlying hardware resource to achieve optimal application performance instead of blindly trying to utilize all. For example, under a fixed and small external input rate, only utilizing a subset of CPU sockets lead to maximum throughput. This may require tedious tuning process in existing heuristic approaches to determine the optimal configuration. We have added more discussions regarding to this issue in "Comparing different placement strategies" of Section 6.4 of the revised draft.

C3-3: In general, the paper is very well written. The fact that the authors chose to optimize Storm and build the whole system on it is much appreciated. However, this reviewer is not entirely convinced about the key argument behind the central contribution of this work (RLAS).

R3-3: Thanks for your comments. We have addressed this review in R3-2.

C3-4: Prior studies have looked at the problem of deploying plans on multicores like Giceva et al pointed to by authors. Given the similarity between relational query plans and stream processing plans, it should be possible to use similar techniques here. The authors of BriskStream argue that such is not possible due to one important reason: processing capability of each operator varies depending on remote memory access. So they argue that it is necessary to accurately measure remote access latency and combine scaling with placement. However, there is no empirical evidence that such a technique is the ideal one in the paper. There is no evaluation of the impact of the three heuristics, the degree of parallelism choice made by the scaling algorithm, a comparison a baseline RLAS that does not use remote access latency, or with the approach presented by Giceva et al as applied to stream processing. The evaluation of RLAS in Section 5.4 cursorily compares RLAS placement with OS, FF, and RR. Even OS, which is completely agnostic to the application, seems to be as good as, or better than FF and RR. While this clearly shows that FF and RR are bad, this doesn't necessarily prove that RLAS is optimal.

R3-4: Thanks for your comments. We have addressed this review in R3-2.

C3-5: In relational plan deployment, query parallelism is decoupled from placement. Parallelism is often determined by the optimizer in a rewrite pass. Once determined, placement algorithm then determines the right location for each parallel replica. In BrickStream's case, as far as this reviewer observed, identification of "bottleneck" operators can be done during preanalysis. Thus, the degree of parallelism for such operators can be done separately before placement similar to database engines. The authors argue that each such parallel DAG would result in operators having a different cost and so scaling and placement should be combined. But no empirical evidence is offered. Can the scaling algorithm is used separately to deteermine degree of parallelism? In such a case, only the placement would have to deal with remote access latency.

R3-5: Thanks for your comment. The scaling algorithm can be used separately *only if* we ignore the NUMA effect. For example, scaling and placement can be separately optimized under the alternative algorithm RLAS_fix (discussed in R3-2). However, it leads to suboptimal performance. 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 due to the NUMA effect (the key motivation for RLAS), removing bottlenecks has to be done together with placement optimization. We have made it clear in Section 4 of the revised draft. Keeping the two phases separate does not lose optimality. As we discuss in Appendix C of the revised draft, the scaling algorithm iterate over *all* the possible scaling ways in parallel. The iteration loop ensures that we have gone through *all* the ways of scaling the topology bottlenecks.

C3-6: Instead of placement dealing with remote access latency via a cost model, an alternative to this is the Morsels approach from Hyper, where a thread pool is used to adaptively decide units of work. Streambox uses intel TBB with buffer annotation to ensure that data from a producer is passed to a consumer in the same node. So they take a different approach to parallelizing the plan and support NUMA awareness without explicitly using a cost model that considers remote memory access. There is no evaluation to show why the suggested approach of combining scaling and placement is better than such an alternative. If what the authors claim is the case, stream processing that are optimized for scale-up multicores like Streambox that do not explicitly address remote access latency should suffer from poor scalability. This is not demonstrated. Instead, they show that Storm and Flink, two systems that are optimized for scale out, do not scale well on multisockets.

R3-6: Thanks for your comment. We have addressed this review in two aspects.

First, we have added the performance comparison to StreamBox [42], a recently proposed single-node DSPS on shared-memory multi-cores. It adopted a morsel-driven like execution model. StreamBox (out-of-order) shows good performance when the number of working threads are small. However, it scales poorly when multi-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 under large core counts. 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 under large core counts, which is sub-optimized for NUMA overhead in its current stage. We compare their NUMA overhead during execution using Intel Vtune Amplifier [8]. Under 8 sockets, BriskStream issues in average 0.09 cache misses served remotely per k events (misses/k events), which is only 1.5% of StreamBox's 6 misses/k events.

Second, we have discussed the necessity of combining scaling and placement in R3-5.

C3-7: Expand section 5.4 by adding a better alternative to RLAS in addition to OS, FF, and RR. At the very least, one could consider removing the remote memory access latency component from the cost model. If so, what performance does RLAS provide without it? What impact do the heuristics have?

R3-7: Thanks for your comment. We have addressed this review in R3-2.

C3-8 Present a comparison with StreamBox or other alternatives that are optimized for shared memory multicores. Demonstrate that these systems suffer from NUMA overhead due to remote memory access.

R3-8: Thanks for your comment. We have addressed this review in R3-6.

Reviewer: 4

C4-1: The operator allocation problem is not really defined as to maximize the system throughput as claimed in the paper.

R4-1: We appreciate the reviewer's comments. The concerned optimization problem takes external input stream ingestion rate (*I*) as one of its input parameters (see Section 3.1 of the revised draft). To examine the maximum system capacity with given hardware resources, we have assumed input stream ingestion rate (*I*) is sufficiently large and keeps the system busy. That is, 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. Application needs to be re-optimized in response to workload changes [18,25,49]. In our context, the model instantiation may not have to be re-performed as operator specification, such as average time spent on handling each tuple, does not vary if only input rates change. It needs to be re-performed, however, if characteristics, such as tuple size, change in the input data stream. In both cases, the execution plan needs to be re-optimized. As a result, both operator replication and placement may vary over time -- system elasticity is needed. We have added the discussion of adding elasticity feature into BriskStream in Section 5.3 of the revised draft.

C4-2: The paper completely ignore latency, a usual metric for DSP systems.

R4-2: Thanks for your comment. We have added latency comparison experiments for both system and application aspects in Figure 7 and Table 5 of Section 6.3 of the revised draft. The results show that the end-to-end latency of BriskStream is significantly smaller than both Flink and Storm. Despite that our optimization focuses on maximizing system throughput, the much-improved throughput reduces queueing delay [23] and consequently reduces latency.

C4-3: The presentation is a bit imprecise in various places. The experimental configuration is vaguely presented

R4-3: Thanks for your comments. We have added more discussions of experimental settings in Section 6.1 of the revised draft.

C4-3: Only compared with naive alternatives.

R4-3: Thanks for your comments. We have added better baseline algorithm (RLAS_fix) to confirm the necessity of considering varying processing capacity of each operator. We also added the comparison to a more related competitor system (StreamBox).

C4-4: The operator allocation problem is defined to maximize the expected output rate. The expected output rate of an operator is equal to the sum of the input rate if it is under supplied. This means that if the system is under-supplied (when statistics are being collected), the optimizer would only look for a feasible solution that fulfills the constraints without actually optimizing the system throughput. This is problematic. Because in a DSP, stream rates can vary over time. How BrinkStream can use the right parameter to optimize operator allocation in practical scenarios is unclear. BrinkStream does not perform load balancing. Hence if the operator allocation is based on statistics with under-supplied situations, the system could suffer from severe throughput problems in comparing to a balanced allocation if the input rates increase significantly during runtime. In the experiments, only the cases with over-supplied situations are used. This further strengthens my concern.

R4-4: We have addressed this review in R4-1.

C4-5: The paper completely ignore latency, a usual metric for DSP systems. One may argue that latency may not be a significant issue in some DSP application, which I completely agree. However, as a scientific study, it would be nice to understand the latency performance of BrinkStream. What the trade-offs are and what can be improved for designing a system for applications where latency is critical.

R4-5: Thanks for your comments. We have addressed this review in R4-2.

C4-6 The authors only compare LRAS with some naive allocation algorithms. There are plenty algorithms for operator allocation in distributed DSP systems that attempts to collocate operators, e.g. "Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel L. Wolf, Kun-Lung Wu, Henrique Andrade, Bugra Gedik:. COLA: Optimizing Stream Processing Applications via Graph Partitioning. Middleware 2009: 308-327". Why not compare with them? You can argue that the objectives in those methods are different. But this is also true for the naive algorithms, which do not specifically optimize the expected output rate.

R4-6: Thanks for your comment. We have addressed the review in the following aspects. First, we compare the difference between this paper and other related studies in the related work. Second, similar to other existing related studies, this paper assumes *operator cost* (w_v) is given and independent to different execution plans, which is the key concern of RLAS. we have implemented an alternative optimization algorithm (RLAS_fix) as a better baseline algorithm. It utilizes the same searching process of RLAS but assumes each operator has a fixed processing capability that does not vary under different execution plans. It leads to suboptimal

performance for different workloads and in varying NUMA architectures. The results are included in Section 6.4.

C4-7: In Fig 15, it seems the benefit brought by RLAS is not particularly high. But it is acceptable. Why only reporting the figures for LR?

R4-7: Thanks for your comments. We have added more comparison results in Figure 12 of the revised draft.

C4-8: Some details of the experimental configuration are missing. For example, what are the input rates of the different streams in LR?

R4-8: Thanks for your comments. We have added the selectivity of different operators and different streams of LR in Appendix B of the revised draft, where we discuss more application settings.

Reviewer: 5

C5-1: Unclear why the discussed problem is a stream-specific problem, instead of a query-processing-on- multicores one.

R5-1: Thanks for your comments. NUMA-awareness system optimization has been previously studied in the context of relational database [26,38,47]. However, those works are either 1) based on cardinality estimation [26], which is unknown in executing queries over potentially infinite input streams, 2) focused on different optimization goals (e.g., better load balancing among CPU sockets [47]) or 3) based on different system architectures [38]. These works provide highly valuable techniques, mechanisms and execution models but none of them uses the knowledge at hand to solve the problem we address, which is how to find an execution plan that maximizes streaming application throughput on multicore machines under the NUMA effect. Due to the difference in problem assumptions, we adopt the rate-based optimization (RBO) approach [51], where output rate of each operator is estimated. However, the original RBO assumes processing capability of an operator is predefined and independent of different execution plans, which is not suitable in our optimization context due to the NUMA effect. We have made this clearer in section 1 of the revised draft.

C5-2: Experimental evaluation omits systems optimized for single-node performance, contains few workloads, and lacks details.

R5-2: Thanks for your comments. We have added comparison experiments to StreamBox, a recent single node DSPS in the revised draft. We have added more detailed description of experimental settings in Section 6.1. We will add more workloads in future work.

C5-3: System implementation section proposes already established techniques.

R5-3: Thanks for your comments. 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 [55], much of the overhead come from the system design itself. For example, average processing time per tuple (T^e) maybe significantly larger than average fetching time per tuple (T^f), and NUMA effect has a minor impact in the plan optimization. We have discussed two design aspects of BriskStream that are specifically optimized for shared-memory architectures that reduce T^e and T^f significantly. Especially, BriskStream reduces T^e, 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. We have made this clearer in Section 5 and Section 6.3 of the revised draft.

C5-4: Discussion of the "three heuristics" is clearer in the intro section that in Section 3. Consider rewriting the latter part for clarity.

R5-4: Thanks for your comments. We have revised the contents accordingly in the revised draft.

C5-5: Unclear why the optimization phases proposed by the authors are stream-processing-specific. The two main concerns, namely i) operator placement and ii) operator replication factor seem directly mappable to NUMA-aware analytical processing and determining the degree of parallelism for each operator; there are numerous research papers addressing those concerns, some of them being data morsels from Leis et al., and NUMA-aware data and task placement by Psaroudakis et al. among others.

R5-5: Thanks for your comments. We have addressed this review in R5-1.

C5-6: The authors mention in passing in Appendix B.2 that operator fusion can impact their design. Pipelined query processing needs to be explicitly mentioned in the main part of the paper, and the authors need to explain whether their contributions are still applicable in the cases where maximizing pipelining in the query plan relaxes resource constraints on the NUMA machine.

R5-6: We appreciate the reviewer's comments. BriskStream shares many similar architectures to modern DSPSs including pipelined processing, where each operator runs independently and exchange information through message passing. In its current stage, each operator (note that, each operator replica is treated as an independent operator to schedule) runs independently in a dedicated thread and pipelining is already maximized. Our proposed optimization problem then concerns how to place each operator (as a thread) into CPU sockets. Obviously, this is not always the best execution strategy. For example, operator fusion combines operators, e.g., two operators can run in the same thread. This reduces cross-operator communication but reduces pipelined parallelism. Our model is general enough to extend to consider operator fusion, but it further complicates the optimization problem (in fact, it adds another dimension to the searching space of the concerned optimization problem). There are also many other optimization techniques available [30], such as operator fission that allows operator been further split into sub-operators, which can be also similarly included into BriskStream. Regardless of whether those (e.g., fission and fusion) techniques are considered, execution plan optimization is required to decide the parallelism of each resulting operator (e.g., as a result of operator fission or fusion) as well its placements. In this paper, we focus on operator placement and replication, and leave the extension to consider other optimization techniques as future work to explore. We have made this clearer in Appendix C.2 of the revised draft.

C5-7: The optimization process that the authors propose comprises two phases, repeated iteratively. Does keeping the two phases separate mean that some potentially optimal configurations will not be considered?

R5-7: Thanks for your comments. The scaling algorithm can be used separately *only if* we ignore the NUMA effect. For example, scaling and placement can be separately optimized under the alternative algorithm RLAS_fix (discussed in R3-2). However, it leads to suboptimal performance. 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 due to the NUMA effect (the key motivation for RLAS), removing bottlenecks has to be done together with placement optimization. We have made it clear in Section 4 of the revised draft. Keeping the two phases separate does not lose optimality. As we

discuss in Appendix C of the revised draft, the scaling algorithm iterate over *all* the possible scaling ways in parallel. The iteration loop ensures that we have gone through *all* the ways of scaling the topology bottlenecks.

C5-8: Figures 4 and 6 should ideally be placed on the page that contains their text description. Also, it is unclear to me what the dotted arrows of Figure 4 refer to.

R5-8: Thanks for your comments. We have revised accordingly in the revised draft.

C5-9: Is there a specific reason that the authors opted to extend Storm instead of Heron, which arguably is the evolved version of Storm?

R5-9: We are sorry for the confusion. BriskStream is a new DSPS implemented from the ground up supporting the same APIs of Storm and Heron. Its architecture 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 open-source DSPSs such as Storm, Heron and Flink, notably including *API design, application topology compiler, pipelined execution engine with communication queue and back-pressure mechanism*. Especially, BriskStream avoids designs that are not suitable for shared-memory multicore architectures. For example, Heron has an operator-per-process design, where each operator of an application is launched in a dedicated JVM process. Despite that it improves debug-ability [35], such design brings significant cross-process overhead, and is not suitable for BriskStream to improve application throughput. In contrast, operators of the same application are launched as Java threads inside the same JVM process in BriskStream, which avoids cross-process communication and allows the efficient pass-by-reference message passing mechanism. We have made these points clear in Section 5 of the revised draft.

C5-10: The optimizations mentioned in Sections 4.1 and 4.2 are straightforward code refactorings that one would perform to turn a distributed system into a single-node solution, and in some cases explicitly address current shortcomings of Storm. In addition, Figure 12 indicates that the gains of BriskStream stem, to a great extent, from these optimizations.

R5-10: Thanks for your comments. We have addressed this review in R5-3.

C5-11: The description of the experimental setup is rather limited, and points to the appendix and to a previous paper for more details. The main body of the current paper needs to be self-contained.

R5-11: Thanks for your comments. We have revised accordingly in the revised draft.

C5-12: The queries that the authors use are rather limited. Worse, most of the plots end up revolving around two queries: 'WC' or 'LR'.

R5-12: Thanks for your comments. We have improved the revised draft by adding more experimental results and Figures. Specifically, Figure 12 and 16 include all applications. We will add more workloads in future work.

C5-13: Figure 7 offers little insight, given that it revolves around (only) 20 random plans that were actually randomly generated. Given the number of operators in the LR query, the choice of 20 plans seems arbitrary and small, while the reader is also unable to figure out why the plan chosen by BriskStream is of better quality.

R5-13: Thanks for your comments. We have added a discussion of two properties of optimized plan of RLAS, which are also found in randomly generated plans with relatively good performance. Those observations may shed light on future research on the concerned optimization problem. The detailed revision is in Section 6.4, where we discuss correctness of heuristics.

C5-14: Through the entire paper, the authors offer little comparison to existing stream processing systems that target single-box deployment (e.g., Trill), as well as streaming systems that evolved from OLTP solutions, such as S-Stream, and thus are prone to be very CPU-efficient.

R5-14: Thanks for your comments. We have added comparison experiments to StreamBox, a recent single node DSPS in Section 6.3. We aim to demonstrate the importance of relative-location awareness (the key motivation of RLAS) during streaming execution plan optimization under NUMA effect. We hence leave comparison to other systems (like Trill and SStore) as future work to explore.

C5-15: Likewise, the evaluation section compares BriskStream against essentially distributed systems, which naturally pay a"(de)serialization and distributed execution tax".

R5-15: We are sorry for the confusion. During the execution, Storm/Flink checks whether the targeting consumer is a local thread or in different process (e.g., in another machine) and applies different transmission approaches accordingly. (De)serialization is *not* actually executed in existing DSPSs running in shared-memory environment. Nevertheless, those unnecessary components bring many conditional branch instructions, which are completely avoided in BriskStream. We have explained this in section 5 of the revised draft.

C5-16: Should Figure 12 have bars for Storm that also include RMA?

R5-16: Thanks for your comments. In Figure 12 (Figure 10 of the revised draft), we aim to show the heavy execution process (large T^e) of Storm compared to BriskStream. RMA cost in Storm is similar to that in BriskStream, and hence omitted. Yet, it becomes more critical to address in BriskStream due to the much-improved efficiency of other components (e.g., significantly reduced T^e).

C5-17: In Section 6, the authors indicate that "the pipeline execution model of DSP systems requires different cost models". Why is that, and what would be the difference when comparing the DSP paradigm to i) operator- at-a-time OLAP (i.e., a la Vectorwise / C-Store), and to ii) pipelined OLAP (i.e., a la HyPer)?

R5-17: Thanks for your comments. We have addressed this review in the following aspects.

First, Leis et al. [37] proposed a novel morsel-driven query execution model in HyPer, which integrates both NUMA-awareness and fine-grained task-based parallelism. Similar execution model is adopted in StreamBox [41], which we compared in our experiments. The results are included in Section 6.3, which reaffirm the superiority of BriskStream in addressing NUMA effect.

Second, DSPSs need to handle potentially infinite input stream, where cardinality based model (e.g., [51]) cannot be applied as cardinality is unknown in our case. Due to the differences in problem assumption, we adopt the rate-based optimization framework that predicts output rate of each operator. However, the original RBO leads to suboptimal performance in our case due to the unaware of varying processing capability of each operator due to the NUMA effect. We have also added more discussion of related work in Section 7 of the revised draft. Specifically, 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, placement strategy such as RR balances resource utilization among CPU sockets, but shows suboptimal performance.

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

ABSTRACT

Modern multicore architectures have demonstrated superior performance for real-world applications. However, fully exploiting the computation power delivered by multicore architectures can be challenging. This paper introduces BriskStream, an in-memory data stream processing system (DSPSs) specifically designed for efficient stream computation on 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 when processing different types of workloads.

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

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]. Witnessing the emergence of modern commodity machines with massively parallel processors, researchers and practitioners find shared-memory multicore architectures an attractive streaming platform [42, 55]. However, fully exploiting the computation power delivered by multicore architectures can be challenging. Prior studies [55] have shown that existing DSPS underutilize the underlying complex hardware micro-architecture and especially show poor scalability due to the unmanaged resource competition and unaware of non-uniform memory access (NUMA) effect.

Many DSPSs, such as Storm [5], Heron [35], Flink [4] and Seep [24], 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 ingestion rate, each operator can be replicated into multiple instances running in parallel. 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 physical resources (i.e., operator placement). In this paper, we address the question of how to find a streaming execution plan that maximizes processing throughput of DSPSs under NUMA effect.

NUMA-awareness system optimization has been previously studied in the context of relational database [26, 38, 48]. However, those works are either 1) based on cardinality estimation [26], which is unknown in executing queries over potentially infinite input streams, 2) focused on different optimization goals (e.g., better load balancing [48]) or 3) based on different system architectures [38]. 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 optimizing 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 very different to some related work [26, 33, 51], which assume a predefined and fixed processing capability (or cost) of each operator.

SIGMOD'19, June 2019, Amsterdam, The Netherlands
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

While RLAS provides a more accurate estimation of the application behavior under the NUMA effect, the resulting placement optimization problem becomes much harder 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 makes classical approaches like dynamic programming not applicable as it is hard to find common sub-problem. Additionally, the placement decisions may conflict with each other and ordering is introduced into the problem. For instance, scheduling of an operator 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, 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.

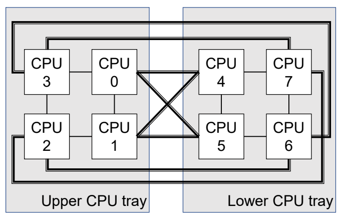
RLAS optimizes both replication and placement at the same time. The key to optimize replication configuration of a streaming application is to remove bottlenecks in its streaming pipeline. As each operator's throughput and resource demand may *vary* in different placement plans due to the NUMA effect,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

We implemented RLAS in BriskStream, a new DSPS supporting the same APIs as Storm and Heron. Our extensive experimental study on two eight-sockets modern multicores servers show that BriskStream achieves much higher throughput and lower latency than existing DSPSs.

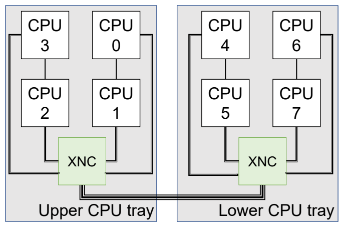
Organization. The remainder of this paper is organized as follows. Section 2 covers the necessary background of scale-up servers and overview of DSPSs. Section 3 discuss the performance model and problem definition of RLAS, followed by a detailed algorithm design in Section 4. Section 5 introduces BriskStream with two optimizations 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 gives an overview of DSPSs.



(a) Server A (glue-less)



(b) Server B (glue-assisted)

Figure 1: Interconnect topology for our servers.

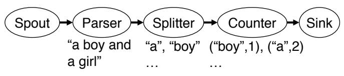
2.1 Modern Scale-up Servers

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 in a socket is accessing "local" or "remote" memory. Such NUMA effect requires ones to carefully align the communication patterns accordingly to get good performance.

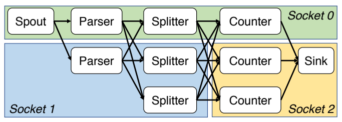
Different NUMA configurations exist in nowadays market, which further complicates the software optimization on them. Figure 1 illustrates the NUMA topologies of our servers. In the following, we use "Server A" to denote the first, and "Server B" to denote the second. Server A can be categorized into the glue-less NUMA server, where CPUs are connected directly/indirectly through QPI or vendor custom data interconnects. Server B employs 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

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 follows. *Spout* continuously generates new tuple containing a sentence with ten random words. *Parser* drops tuple with a null 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 value as the number of occurrence of the corresponding word. Every time it receives a word from Splitter, it updates the hashmap and emits a tuple containing



(a) Logical view of WC.



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

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

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 [55]. 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 well known for its advantage of low processing latency and being adopted by many DSPSs such as Storm [5], Flink [4], Seep [24], and Heron [35]. Figure 2(b) illustrates one example execution plan of WC, where parser, splitter and counter are replicated into 2, 3 and 3 instances, and they are placed in three CPU sockets (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. Operator experiences additional remote memory access (RMA) penalty during input data fetch when it is allocated in different CPU sockets to its producers. A bad execution plan may introduce unnecessary RMA communication overhead and/or oversubscribe a few CPU sockets that induces significant resource contention. In this section, we discuss the performance model that guides optimization process and the formal definition of our concerned optimization 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 [26]. Yet, those works are based on cardinality estimation, which is unknown in streaming workloads. Due to the difference in problem assumptions, we adopt the rate-based optimization (RBO) approach [51], where output rate of each operator is estimated. However, the original RBO assumes processing capability of an operator is predefined and independent of different 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 instance 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 form the performance model that we are interested in. To simplify the presentation, we omit and assume selectivity is one in the following discussion. In our experiment, the selectivity statistics of each operator is pre-profiled before the optimization applies. In practice, they can be periodically collected during the application running and the optimization needs to be re-performed accordingly.

Table 1: Summary of terminologies

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

Model overview. In the following, we refer to the output rate of an operator by using the symbol r_o, while r_i refers to its input rate. 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].

We define r_i of an operator as number of tuples available for it to process. 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 ingestion rate). Conversely, upon obtaining the reference, operator then needs to fetch the actual data during its processing, where the actual data fetch delay depends on NUMA distance of it and its producer. Subsequently, the varying efforts spend in data fetching affects r_o in varying execution plans. 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 aggregation of input tuples from all producers arrived during t, and t_p is the total time spent on processing those input tuples.

$$\begin{aligned} r_o &= \frac{num}{t_p}, \\ where\ num &= \sum_{producers} r_i * t \\ t_p &= \sum_{producers} r_i * t * T(p). \end{aligned} \tag{1}$$

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

T^e stands for time required in actual function execution and emitting outputs 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 fetch (local or remotely) 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,

$$T^f = \begin{cases} 0 & \text{if collocated with producer} \\ \lceil N/S \rceil * L_(i,j) & \text{otherwise} \end{cases}$$
$$\text{, where i and j are determined by } p. \tag{2}$$

When the operator is collocated with its producer, the data fetch cost is already covered by T^e and hence T^f is 0. Conversely, 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 prior work from Surendra and et al. [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 ($i \neq j$). Despite its simplicity, 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., $t_p > t$. 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 [1]. Therefore, $r_o(s)$ is proportional to the proportion of the corresponding input ($r_i(s)$), that is, $\overline{r_o(s)} = r_o * \frac{r_i(s)}{r_i}$.

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

Given an execution plan, we can then identify operators that are over-supplied by comparing its input rate and output rate. Those over-supplied operators are essentially the "bottlenecks" of the 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 search 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.

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

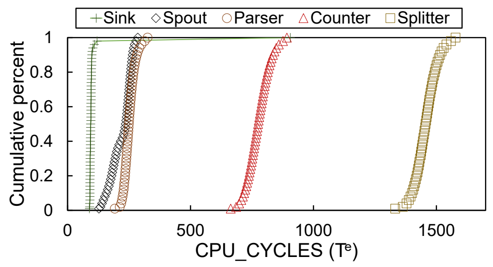


Figure 3: CDF of profiled average execution cycles of different operators of WC.

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 [22], we need to profile the application to determine operator specifications. To prevent 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 our execution plan optimization (RLAS) 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 successfully guides our system to scale in multi-socket multicores.

3.2 Problem Formulation

The goal of our optimization is to maximize the application processing throughput under given input stream ingestion rate, where we search for the optimal replication level and placement of each operator. Note that, each replica is considered as an operator to be scheduled. For one CPU socket, denote its available CPU cycles as C cycles/sec, the maximum attainable local DRAM bandwidth as B bytes/sec, and the maximum attainable remote channel bandwidth from socket S_i to S_j as $Q_{i,j}$ bytes/sec. Further, denote average tuple size, memory bandwidth consumption and processing time spent per tuple of an operator as N bytes, M bytes/sec and T cycles, respectively, The problem can be mathematically formulated as Equation 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 3 enforces that the aggregated demand of CPU resource requested to anyone CPU socket must be smaller than the available CPU resource. Constraint 4 enforces that the aggregated amount of bandwidth requested to a CPU socket must be smaller than the maximum attainable local DRAM bandwidth. Constraint 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 will be and only be allocated exactly once. This matters because an operator may have multiple producers that are allocated at different places. In this case, the operator may be collocated with only a subset of its producers.

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

$$\text{s.t.}, \forall i, j \in 1, .., n,$$

$$\sum_{operators\ at\ S_i} \overline{r_o} * T \le C, \tag{3}$$

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

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

Assuming each operator (in total $|o|$ excluding replicas) can be replicated at most k replicas, we have to consider in total $k^{|o|}$ different replication configurations. In addition, for each replication configuration, there are m^n different placements, where m is the number of CPU sockets and n stands for the total number of replicas ($n \ge |o|$). 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 placement (i.e., CPU affinity) of each operator at the same time guided by our performance modelling. 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

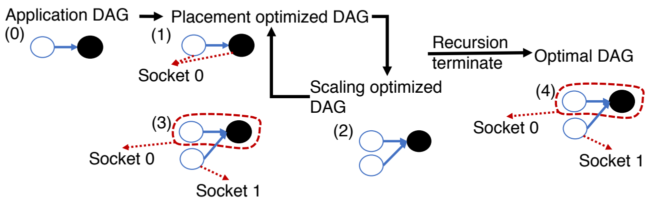


Figure 4: RLAS Optimization example.

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 are determined during placement optimization. The bottleneck operator is defined as the operator that has a larger input rate than its processing capability (see Section 3.1 case 1). Figure 4 shows 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 based technique [43] is applied to solve our placement optimization problem assuming operator replication is given and fixed. We focus on discussing our bounding function and unique heuristics that improve the searching efficiency.

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

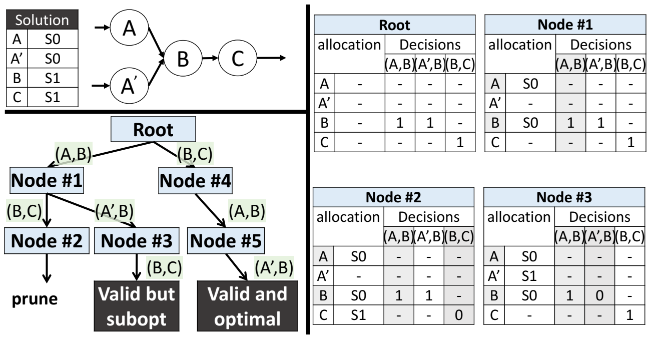
Algorithm complexity: Naively in each iteration, there are $\binom{n}{1} * \binom{m}{1} = n * m$ possible solutions to branch, i.e., schedule *which* operator to *which* socket and an average n depth as one operator is allocated in each iteration. In other words, it will still need to examine on *average* $(n * m)^n$ candidate solutions [41]. In order to further reduce the complexity of the problem, heuristics have to be applied.

The bounding function. If the bounding function value of an intermediate node is worse than the solution node obtained so far, we can safely prune it and all of its children nodes. This does not affect the optimality of the algorithm because the value of a live node must be better than all its children node after further exploration. In other words, the value of a live node is the theoretical upper bound of the subtree of nodes. The bounded problem that we used in our optimizer originates from the same optimization problem with relaxed constraints. Specifically, the bounded value of every live node is obtained by fixing the placement of *valid* operators and let *remaining* operators to be collocated with all of its producers, which may violate resource constraints as discussed before, but gives the upper bound of the output rate that the current node can achieve.

Consider a simple application with operators A, A' (replica of A) and B, where A and A' are producers of B. Assume at one iteration, A and A' are scheduled to socket 0 and 1, respectively (i.e., they become valid). We want to calculate the bounding function value assuming B is the sink operator, which remains to be scheduled. In order to calculate the bounding function value, we simply let B be collocated with both A and A' at the same time, which is certainly invalid. In this way, its output rate is maximized, which is the bounding value of the live node. The calculating of our bounding function has the same cost as evaluating the performance model since we only need to mark T^f (Formula 2) to be 0 for those operators remaining to be scheduled.

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

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.



Solution	
A	S0
A'	S0
B	S1
C	S1

Root				
allocation		Decisions		
		(A,B)	(A',B)	(B,C)
A	-	-	-	-
A'	-	-	-	-
B	-	1	1	-
C	-	-	-	1

Node #1				
allocation		Decisions		
		(A,B)	(A',B)	(B,C)
A	S0	-	-	-
A'	-	-	-	-
B	S0	1	1	-
C	-	-	-	1

Node #2				
allocation		Decisions		
		(A,B)	(A',B)	(B,C)
A	S0	-	-	-
A'	-	-	-	-
B	S0	1	1	-
C	S1	-	-	0

Node #3				
allocation		Decisions		
		(A,B)	(A',B)	(B,C)
A	S0	-	-	-
A'	S1	-	-	-
B	S0	1	0	-
C	-	-	-	1

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.

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 least remaining resource. If there are multiple equal choice, we only branch to one of them to reduce problem size.

3) Compress graph: The third provides a mechanism to tune the trade-off between optimization granularity and searching space. Under large replication level setting, the execution graph becomes very large and the searching space is huge. We compress the execution graph by grouping multiple (determined by *compress ratio*) replicas of an operator into a single large instance that is scheduled together. Essentially, the compress ratio trade off the optimization granularity and searching space. By setting the ratio to be one, we have the most fine-grained optimization but it takes more time to solve. In our experiment, we set the ratio to be 5, which produces a good trade-off.

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

In this example, we assume the aggregated resource demands of any combinations of grouping three operators together exceed the resource constraint of a socket, and the only optimal scheduling plan is shown beside the topology. The bottom left of the Figure shows how our algorithm explores the searching space by expanding nodes, where the label on the edge represents the collocation decision considered in the current iteration. The detailed states of four nodes are illustrated on the right-hand side of the figure, where the state of each node is represented by a two-dimensional matrix. The first (horizontal) dimension describes a list of collocation decisions, while the second one the operators that interests in this decision. A value of '-' means that the respective operator is not interested in this collocation decision. A value of '1' means that the collocation decision is made in this node, although it may violate resource constraints. An operator is interested in the collocation decision involving itself to minimize its remote memory access penalty. A value of '0' means that the collocation decision is not satisfied and the involved producer and consumer are separately located.

At root node, we consider a list of scheduling decisions involving each pair of producer and consumer. At Node #1, the collocation decision of A and B is going to be satisfied, and assume they are collocated to S0. Note that, S1 is identical to S0 at this point and does not need to repeatedly consider. The bounding value of this node is essentially collocating all operators into the same socket, and it is larger than solution node hence we need to further explore. At Node #2, we try to collocate A' and B, which however cannot be satisfied (due to the assumed resource constraint). As its bounding value is worse than the solution (if obtained), 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 [55] , much of the overhead come from the system design itself. For example, T^e may be significantly larger than T^f, and NUMA effect has a minor impact in the plan optimization. This is further validated in our experiments later.

We integrate RLAS optimization framework into BriskStream [2], a new DSPS implemented from the ground up supporting the same APIs as Storm and Heron. Its architecture 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*

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

queue and back-pressure mechanism. 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 the extension of BriskStream with elasticity in Section 5.3.

5.1 Improving Execution Efficiency

As shown in the previous work [55], instruction footprint between two consecutive invocations of the same function in existing DSPSs is large and resulting in significant instruction cache misses stalls. We avoid 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). Note that, those components may *not* actually be executed in existing DSPSs running in shared-memory environment. For example, during execution, Storm/Flink checks whether the targeting consumer is in the same or different processes (e.g., in another machine). It applies different transmission approaches accordingly, and (de)serialization is not actually involved. Nevertheless, those unnecessary components bring many conditional branch instructions, which are completely avoided in BriskStream.
Furthermore, we carefully revise the critical execution path to not create unnecessary/duplicate temporary objects. For example, as an output tuple is exclusively accessible by its targeted consumer and all operators share the same memory address, we do not need to create a new instance of the tuple when the consumer obtains it.

5.2 Improving Communication Efficiency

Most modern DSPSs [4, 5, 55] employ buffering strategy to accumulate multiple tuples before sending 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 one-time access to the communication queue and effectively amortizing the insertion overhead. As a result, both T^e and T^f are significantly reduced.

5.3 Discussion on Elasticity

To examine the maximum system capacity, we assume input stream ingestion 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. Application needs to be re-optimized in response to workload changes [18, 25, 49]. In our context, the model instantiation may not have to be re-performed as operator specification, such as average time spent on handling each tuple, does not vary if only input rates change. It needs to be re-performed, however, if characteristics, such as tuple size, change in the input data stream. In both cases, the execution plan needs to be re-optimized. As a result, both operator replication and placement may vary over time – system elasticity is needed.

Elasticity of BriskStream can be easily achieved for stateless operators. Specifically, to achieve operator re-replication and re-placement, we only need to consolidate or re-spawn operator replicas to targeting CPU sockets. Conversely, *state migration* is needed for stateful operators, which requires state movement and synchronization [25]. Both can bring considerable overhead to the system. As a result, execution plan optimization needs to incorporate 1) the overhead of state migration, and 2) runtime of re-optimization including potential model re-instantiation into consideration. As it may be harmful if the overhead outweigh the gain in deploying a new execution plan, adding elasticity to BriskStream is, by itself, a nontrivial question. Previous study [29] introduces a model to estimate the movement cost in terms of end-to-end latency. Similar techniques may be applied, which we leave as future work to further enhance our system.

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 three 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 several of BriskStream's optimization techniques.

6.1 Experimental Setup

We pick four common applications from the previous study [55] with different characteristics to evaluate BriskStream. These tasks are word-count (WC), fraud-detection (FD), spike-detection (SD), and linear-road (LR)

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

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 ingestion rate (I) to its maximum attainable value (I_{max}) to keep the system busy and report the stable system performance [3]. 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. NUMA characteristics, such as local and inter-socket idle latencies and peak memory bandwidths, are measured with Intel Memory Latency Checker [7]. These two machines have different NUMA topologies, which lead to different access latencies and throughputs across CPU sockets. The 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, it 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 tray (max hops latency) on both 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.

In addition to runtime statistics evaluation, we also report how much time each tuple spends in different components of the system. We classify these work 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 *round-trip delay* by *Execute* that represents additional overhead. 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 [37]. 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 for using more than 4 sockets.

To validate the overall effectiveness of our performance model, we show the relative error associated with estimating the application throughput by our analytical model. The relative error is defined as $relative\_error = \frac{|R_{meas} - R_{est}|}{R_{meas}}$, where R_{meas} is the measured application throughput and R_{est} is the estimated application throughput by our performance model for the same application.

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

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	1666.53	1991.14	S0-S1	611.4	665.23
S0-S3	1708.2	1994.85	S0-S3	623.07	665.92
S0-S4	2050.63	2923.65	S0-S4	889.92	837.92
S0-S7	2371.31	3196.35	S0-S7	870.23	888.42

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

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

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 single task manager, which shows even worse performance. We also compare BriskStream with StreamBox, a recent single-node DSPS 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 performance for three applications including WC, FD and SD. Flink performs poorly for LR compared to Storm. A potential reason is that Flink requires additional stream merger operator (implemented as the co-flat map) that merges multiple input stream before feeding to an operator with multi-input streams (commonly found in LR). Neither Storm nor BriskStream has such additional overhead.

Following the previous work [23], 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. This is one of the key metrics in DSPS that significantly differentiate itself to traditional batch based system such as MapReduce. 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. Despite that our optimization focuses on maximizing system throughput, the much-improved throughput reduces queueing delay [23] and consequently reduces latency.

Evaluation of scalability on varying CPU sockets. Our next experiment shows that BriskStream scales effectively as we increase the numbers of sockets. RLAS re-optimizes the execution plan under a different number of sockets enabled. Figure 9a shows the better scalability of BriskStream than existing DSPSs on multi-socket servers by taking LR as an example. Both 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 taking 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. Furthermore, we still need to re-optimize the execution plan to achieve optimal performance in the presence of changing RMA cost (e.g., in this extreme case, it is reduced to zero).

Per-tuple execution time breakdown. To better understand the source of performance improvement, we show the per-tuple execution time breakdown 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 for its simplicity. We perform analysis in two groups: *local* stands for allocating all operators to the same socket, and *remote* stands for allocating

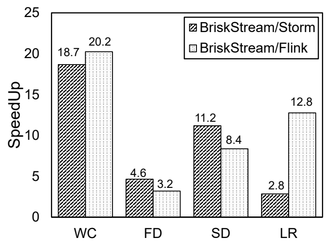


Figure 6: Throughput speedup.

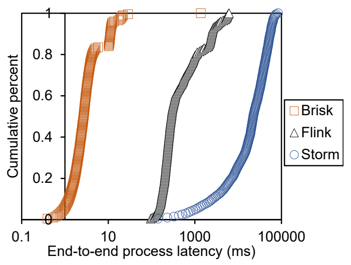


Figure 7: End-to-end latency of WC on different DSPSs.

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

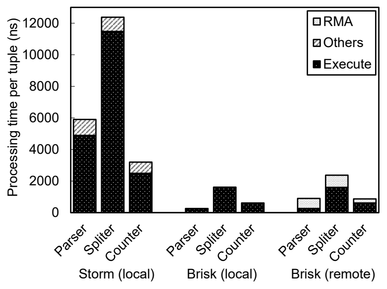
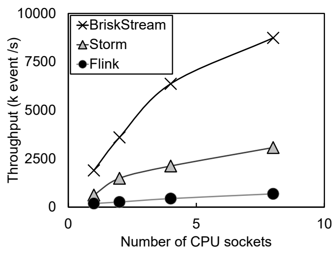
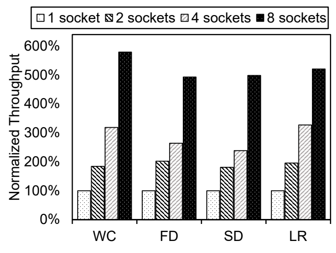


Figure 8: Execution time breakdown.



(a) Systems



(b) Applications

Figure 9: Scalability evaluation.

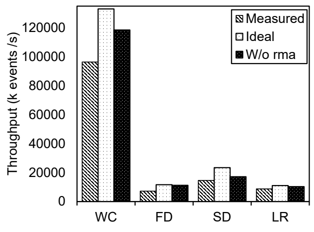


Figure 10: Gaps to ideal.

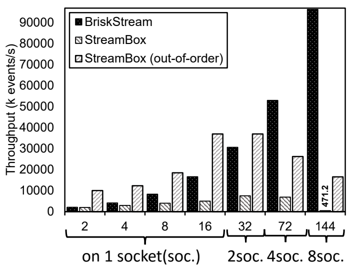


Figure 11: Comparing with StreamBox.

each operator max-hop away from its producer to examine the cost of RMA.

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

In the remote group, we compare the execution of the same operator in BriskStream with or without remote memory access overhead. In comparison with the locally allocated case, the total round trip time of an operator is up to 9.4 times higher when it is remotely allocated to its producer. In particular, Parser has little in computation but has to pay a lot for remote memory access overhead ($T^e << T^f$). The significant varying of 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 *RMA*, which means $T^e >> T^f$ and NUMA effect may have a minor impact in its plan optimizing. 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.

Comparing with single-node DSPS. Streambox [42] is a recently proposed DSPS based on a morsel-driven like execution model – a different processing model of BriskStream. We compare BriskStream with StreamBox [42] using WC application. Results in Figure 11 demonstrate that BriskStream outperforms StreamBox significantly regardless of 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 multi-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 under large core counts. 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 under large core counts, which is sub-optimized for NUMA overhead in its current stage. 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 is only 1.5% of StreamBox's 6 misses/k events.

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 approach essentially falls back to the original RBO model, and is also similar to some previous works in a high level point of view [26, 33]. 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 underutilize 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 optimization process to involve severely thread interference. Over the four workloads, RLAS shows a 119% ∼ 455% 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 and RR are enforced to guarantee resource constraints as much as possible. In case they cannot find any plan satisfying resource constraints, they will gradually relax constraints (i.e., using an increased, yet faked, total resource per socket during determining if the give execution plan satisfying 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. 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 view as a minimizing traffic heuristic-based approach as it greedily allocates neighbor operators (i.e., directly connected) together due to its topologically sorting step. Several related work [13, 53] adopt a similar approach of FF in duelling with operator placement problem in the distributed environment. However, it performs poorly, and we find that during its searching for optimal placements, it often falls into "not-able-to-progress" situation as it cannot allocate the current item (i.e, operator) into any of the sockets because of the violation of resource constraints. This is due to its greedy nature that leads to a local optimal state. Then, it has to relax the resource constraints and repack the whole topology, which often ends up with oversubscribing of a few CPU sockets. The major drawback of RR is that it does not take remote memory communication overhead into consideration, and the resulting plans often involve unnecessary cross sockets communication.

Second, RLAS performs generally better than other placement strategies 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 choices 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.

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 correctness 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 limits. 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.

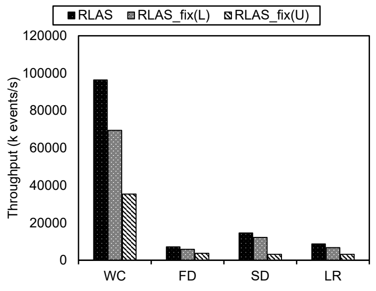
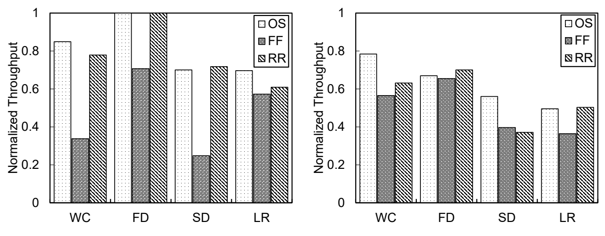


Figure 12: RLAS w/ and w/o considering varying RMA cost.

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



(a) Server A (b) Server B

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

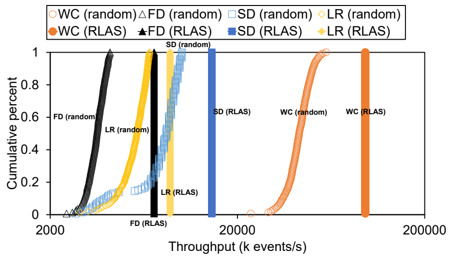


Figure 14: CDF of random plans.

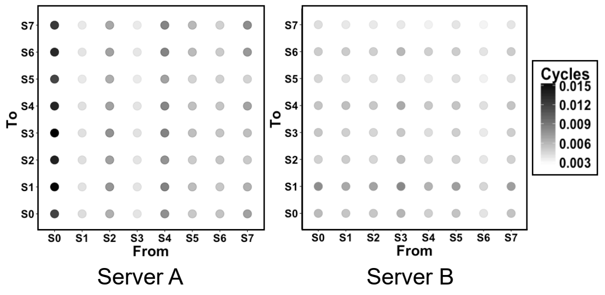


Figure 15: Communication pattern matrices of WC on two Servers.

Table 7: Tuning compression ratio (r)

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

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, of across CPU-tray, should be aggressively avoided in these two applications. Second, resources are highly appreciated in RLAS. Most operators (incl. replicas) end up with being "just fulfilled", i.e., $\overline{r_o} = r_o = r_i$. 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 (S0) to other sockets in Server A, and they are, in contrast, much more uniformly distributed among different sockets in Server B. The main reason is that the remote memory access bandwidth is almost identical to local memory access in Server B thanks to its glue-assisted component as discussed in Section 2, and operators are hence more uniformly placed at different sockets.

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-optimize. This procedure introduces more complexity to the algorithm, which leads to higher runtime as shown in Table 7. Due to space limitation, a detailed discussion is presented in Appendix C.1.

6.5 Factor Analysis

To understand in greater detail 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 (version 1.1.1) directly on shared-memory multicores. *-Instr.footprint* refers

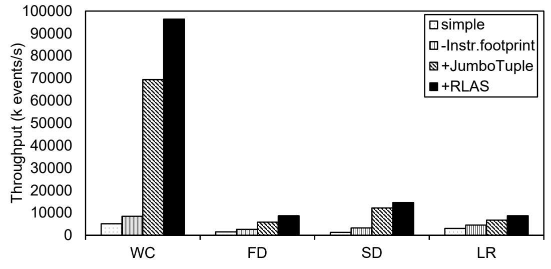


Figure 16: A factor analysis for BriskStream. Changes are added left to right and are cumulative.

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 is critical for DSPSs to scale different applications on modern multicores environment 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, 54]. There have been studies on optimizing the instruction cache performance [27, 56], the memory and cache performance [11, 15, 16, 28] and NUMA [26, 38, 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, placement strategy such as RR balances resource utilization among CPU sockets, but shows suboptimal performance in our experiments. Leis et al. [38] proposed a novel morsel-driven query execution model which integrates both NUMA-awareness and fine grained task-based parallelism. 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 [35]. 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 scales on large multicores as our experiment shows. Previous work [55] 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 hence requires tedious tuning process and 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 [34] 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 multicores environment.

Execution plan optimization: Both operator placement and operator replication are widely investigated in the literature under different assumptions and optimization goals [36]. In particular, many algorithms and mechanisms [13, 19, 20, 33, 44, 53] are developed to allocate (i.e., schedule) operators of a job into physical resources (e.g., compute node) in order to achieve certain optimization goal, such as maximizing throughput, minimizing latency or minimizing resource consumption, etc. Due to space limitation, we discuss more of them in Appendix D. Based on similar idea 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 scales stream computation towards hundred of cores under NUMA effect. The experiments on eight-sockets machines confirm that BriskStream significantly outperforms existing DSPSs up to an order of magnitude even without the tedious tuning process. We hope our study on relative-location aware scheduling could shed lights on other NUMA-aware execution plan optimization research.

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/intelr-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 and et al. 2005. The Design of the Borealis Stream Processing Engine. In *CIDR*.
[11] Anastassia Ailamaki and et al. 2009. DBMSs on a Modern Processor: Where Does Time Go?. In *VLDB*.
[12] M. Akdere and et al. 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 and et al. 2013. Scale-up vs scale-out for hadoop: Time to rethink?. In *SoCC*.
[15] C. Balkesen and et al. 2013. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *ICDE*.
[16] Peter A. Boncz and et al. 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] V. Cardellini and et al. 2016. Elastic Stateful Stream Processing in Storm. In *HPCS*.
[19] Valeria Cardellini and et al. 2016. Optimal Operator Placement for Distributed Stream Processing Applications. In *DEBS*.
[20] Valeria Cardellini and et al. 2017. Optimal Operator Replication and Placement for Distributed Stream Processing Systems. In *SIGMETRICS Perform. Eval. Rev.*
[21] Sirish Chandrasekaran and et al. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*.
[22] Alvin Cheung and et al. 2013. Speeding up database applications with Pyxis. In *SIGMOD*.
[23] Tathagata Das and et al. 2014. Adaptive Stream Processing using Dynamic Batch Sizing. *SOCC* (2014).
[24] 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*.
[25] B. Gedik, S. Schneider, M. Hirzel, and K. L. Wu. 2014. Elastic Scaling for Data Stream Processing. *TPDS* (2014).
[26] Jana Giceva and et al. 2014. Deployment of Query Plans on Multicores. In *VLDB*.
[27] Stavros Harizopoulos and Anastassia Ailamaki. 2006. Improving Instruction Cache Performance in OLTP. *TODS* (2006).
[28] Bingsheng He and et al. 2005. Cache-conscious automata for XML filtering. In *ICDE*.
[29] Thomas Heinze and et al. 2014. Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems. *TPDS* (2014).
[30] Martin Hirzel and et al. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* (2014).
[31] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng. 2010. Oversubscription on multicore processors. In *IPDPS*.
[32] Navendu Jain and et al. 2006. Design, Implementation, and Evaluation of the Linear Road Bnchmark on the Stream Processing Core. In *SIGMOD*.
[33] Rohit Khandekar and et al. 2009. COLA: Optimizing Stream Processing Applications via Graph Partitioning. In *Middleware*.
[34] Alexandros Koliousis and et al. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *SIGMOD*.
[35] Sanjeev Kulkarni and et al. 2015. Twitter Heron: Stream Processing at Scale. In *SIGMOD*.
[36] Geetika T. Lakshmanan and et al. 2008. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Computing* (2008).
[37] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn&Rsquo;T, and Why. *ACM Trans. Archit. Code Optim.* (2012).
[38] 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*.
[39] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. 2018. Model-free Control for Distributed Stream Data Processing Using Deep Reinforcement Learning. *Proc. VLDB Endow.* (2018).
[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 and et al. 2016. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization* (2016).
[44] Boyang Peng and et al. 2015. R-Storm: Resource-Aware Scheduling in Storm. In *Middleware*.
[45] Achille Peternier and et al. 2011. Overseer: low-level hardware monitoring and management for Java. In *PPPJ*.
[46] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. 2012. OLTP on Hardware Islands. *PVLDB* (2012).
[47] Iraklis Psaroudakis and et al. 2015. Scaling up concurrent main-memory column-store scans: towards adaptive NUMA-aware data and task placement. In *VLDB*.
[48] Iraklis Psaroudakis and et al. 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 and et al. 2016. Dynamic Load Balancing for Ordered Data-Parallel Regions in Distributed Streaming Systems. (2016).
[50] Kian-Lee Tan and et al. 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] W. Wu and et al. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *ICDE*.
[53] J. Xu and et al. 2014. T-Storm: Traffic-Aware Online Scheduling in Storm. In *ICDCS*.
[54] Hao Zhang and et al. 2015. In-Memory Big Data Management and Processing: A Survey. *TKDE* (2015).
[55] Shuhao Zhang and et al. 2017. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors. In *ICDE*.
[56] Jingren Zhou and Kenneth A. Ross. 2004. Buffering Databse Operations for Enhanced Instruction Cache Performance. In *SIGMOD*.

A IMPLEMENTATION DETAILS

BriskStream avoids designs that are not 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 (discussed in the end of this section).

Figure 17 presents an example job overview of BriskStream. Each operator (or the replica) of the application is mapped to one *task*. The task is the fundamental processing unit (i.e., executed by a Java thread), which consists of an *executor* and a *partition controller*. The core logic for each executor is provided by the corresponding operator of the application. Executor operates by taking a tuple from the output queues of its producers and invokes the core logic on the obtained input tuple. After the function execution finishes, it dispatches zero or more tuples by sending them to its partition controller. The partition controller decides in which output queue a tuple should be enqueued according to application specified partition strategies such as shuffle partition. Furthermore, each task maintains output buffers for each of its consumers. Specifically, the output tuple is first enqueued into a local buffer (buffering into *jumbo tuple*), which will then be emitted only when it is filled up to the threshold size.

BriskStream adopted the pass-by-reference message passing approach to avoid duplicating data in a shared-memory environment [55]. Specifically, tuples produced by operator are stored locally, and pointers as reference to tuple are inserted into a communication queue. Together with the aforementioned junbo tuple design, reference passing delay is minimized and becomes negligible.

B APPLICATION SETTINGS

In this section, we discuss more settings of the testing applications in our experiment.

Topologies. We have shown the topology of WC before at Figure 2. Figure 18 shows the topology of the other three applications.

Operator selectivity. As mentioned in Section 3.1, we omit and assume selectivity to be one in our presentation of cost model. 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. Count has a 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 *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 (Section C.1). We further elaborate how our optimization paradigm can be extended with other optimization techniques (Section C.2).

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

Algorithm 2 illustrates our *Branch and Bound based Placement* algorithm. Initially, no solution node has been found so far and we initialize a root node with a plan collocating all operators (Line 1~5). At Line 7~14, the algorithm explores the current node. If it has better bounding value than the current solution, we update the solution node (Line 10~11) if it is valid (i.e., all operators are allocated), or we need to further explore it (Line 13). Otherwise, we prune it at Line 14 (this also effectively prunes all of its children nodes). The branching function (Line 15~32) illustrates how the searching process branches and generates children nodes to explore. For each collocation decision in the current node (Line 16), we apply the *best fit heuristic* (Line 17~23) and one new node is created. Otherwise, at Line 25~27, we have to create new nodes for each possible way of placing the two

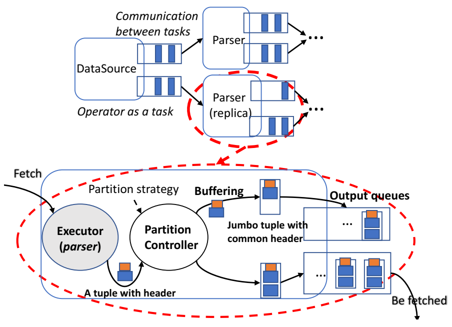
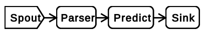
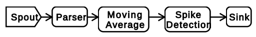


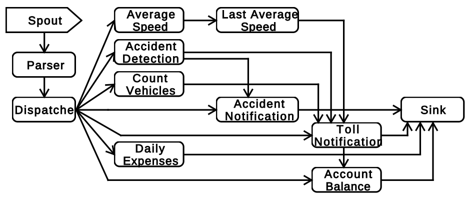
Figure 17: Example job overview in BriskStream.



(a) Fraud Detection (FD)



(b) Spike Detection (SD)



(c) Linear Road (LR)

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
		daliy_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_nofity_stream*	0.0
	position report	*toll_nofity_stream*	1.0
	counts_stream	*toll_nofity_stream*	1.0
	las_stream	*toll_nofity_stream*	1.0
Daily_expen	*daliy_exp_request*	*default*	0.0
Account_balance	*balance_stream*	*default*	0.0

Algorithm 1: Topologically sorted iterative scaling

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

operators (i.e., up to $\binom{m}{1} * \binom{2}{1}$). At Line 28∼ 31, we update the number of valid operators and bounding value of each newly created nodes in parallel. Finally, the newly created children nodes are pushed back to the stack.

C.1 Discussion

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

Observations. We have made some counter-intuitive observations in optimizing our workload. *First*, placement algorithm (Algorithm 2) start with no initial solution (i.e., the *solution.value* is 0 initially at Line 9) by default, and we have tried to use a simple first-fit (FF) placement algorithm to determine an initial solution node to potentially speed up the searching process. In some cases, it accelerates the searching process by earlier pruning and makes the algorithm converges faster, but in other cases, the overhead of running the FF algorithm offsets the gains. *Second*, the placement algorithm may fail to find any valid plan as not able to allocate one or more operators due to resource constraints, which causes scaling algorithm to terminate. It is interesting to note that this *may not* indicates the saturation of the underlying resources but the operator itself is too coarse-grained. The scaling algorithm can, instead of terminate (at Algorithm 1 Line 10), try to further increase the replication level of operator that "failed-to-allocate". After that, workloads are essentially further partitioned among more replicas and the placement algorithm may be able to find a valid plan. This procedure, however, introduces more complexity to the algorithm.

Optimization runtime. The concerned placement optimization problem is difficult to solve as the solution space increase rapidly with large replication configuration. Besides the three proposed heuristics, we also apply a list of optimization tricks to further increase the searching efficiency including 1) memorization in evaluating performance model under a given execution plan (e.g., an operator should behave the same if its relative placement with all of its producers are the same in different plans), 2) 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

Algorithm 2: B&B based placement optimization

```
Data: Stack $stack$ // stors all live nodes
Data: Node $solution$ // stores the best plan found so far
Data: Node $e$// the current visiting node
Result: Placement plan of $solution$ node
// Initilization
$solution.R \leftarrow 0$ // No solution yet
$e.decisions \leftarrow$ a list contains all possible collocation decisions;
$e.plan \leftarrow$ all operators are collocated into the same CPU socket;
$e.R \leftarrow$ BoundingFunction($e.plan$);
$e.validOperators \leftarrow 0$;
Push($stack$,$e$);
while $\neg$IsEmpty($stack$)// Branch and Bound process
do
    $e \leftarrow$ Pop($stack$);
    if $e.R > solution.R$ then
        if $e.validOperators$ == $totalOperators$ then
            $solution \leftarrow e$;
        else
            Branching(e);
    else
        // the current node has worse bounded value
           than solution, and can be safely pruned.
Function Branching($e$):
    Data: Node[] $children$
    foreach pair of $O_s$ and $O_c$ in $e.decisions$ do
        if all predecessors of them are already allocated except $O_s$ to $O_c$
          then
            $\#newAllocate \leftarrow 2$;
            if they can be collocated into one socket then
                create a Node $n$ with a plan collocating them to one
                  socket;
            else
                create a Node $n$ with a plan separately allocating
                  them to two sockets;
            add $n$ to $children$;
        else
            $\#newAllocate \leftarrow 1$;
            foreach valid way of placing $O_s$ and $O_c$ do
                create a new Node and add it to $children$;
    foreach Node $c \in children$// update in parallel
     do
        $c.validOperators \leftarrow$ e.validOperators + $\#newAllocate$;
        $c.R \leftarrow$ BoundingFunction($c.plan$);
    PushAll($stack$, $children$);
```

seconds to optimize placement for a large DAG, and overall scaling takes less than 30 seconds, which is acceptable, given the size of the problem and the fact that the generated plan can be used for the whole lifetime of the application. As the streaming application potentially runs forever, the overhead of generating a plan is not included in our measurement.

C.2 Extension with other optimization techniques

A number of optimization techniques are available in the literature [30]. Many of them can be potentially applied to further improve the performance of BriskStream. Our performance model is general enough such that it can be extended to capture other optimization techniques.

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

$$T^e_{fused} = \sum_{\text{all fused operators}} T^e$$
$$T^f_{fused} = T^f_{up} \qquad (6)$$

The similar idea has been explored in recent work [26]. In this paper, we focus on operator scheduling and replication optimization, and we leave the evaluation of extension to other optimization techniques as future work.

D 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 [53] dynamically assigns/reassigns operators according to run-time statistics in order to minimize inter-node and inter-process traffic while ensuring load balance. R-Storm [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. [19, 20] 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. Recently, Li et al. [39] present a machine-learning based framework for minimizing end-to-end processing latency on DSPSs. It generally remains an open-question whether an optimizer-based (like ours) or model-free approach is more promising [52] in query optimization, and we leave it as a future work to explore the capabilities of the model-free approach in the shared-memory multicores setting.