SCALING DATA STREAM PROCESSING ON MULTICORE ARCHITECTURES

SHU HAO ZHANG
Bachelor of Engineering (Computer Engineering, first class honor)
Nanyang Technological University

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2019

Supervisor:
Associate Professor Bing Sheng He

Examiners:
Associate Professor Chee Yong Chan
Associate Professor Yong Meng Teo
Assistant Professor Felix Xiao Zhu Lin, Purdue University
Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

__________________________
Shu hao Zhang
September 26, 2019
Acknowledgments

This thesis will never have been completed without the generous assistance offered by many people to whom I owe a lot. While it is difficult for me to enumerate all the people to whom I appreciate, I would like to list a few of them below to acknowledge their selfless help.

First and foremost, I would like to thank my PhD advisor Bingsheng He. I will always remember the warm afternoon when I was an undergraduate year 3 student at NTU. I knocked and walked in Prof.He’s office and ask “Hi professor, I sincerely enjoyed your teaching of operating system course, can I do a Final-Year-Project with you regarding system research?” That turns out to be the beginning of my research journey under the supervising of Prof.He of almost 7 years. I thank him for continuously encouraging me to explore any research topic that I am interested in and selfless help and guidance from him. I also thank him for always encouraging me to continue my research whenever I feel frustrated about paper rejection. Beyond the discussion of technical questions, Prof.He had also kindly share with me many advice and lessons, which are insightful for my whole life. I feel extremely grateful for having him standing by me whenever I confront any problems or troubles in either research or career planning. I will never have enough guts to continue my PhD journey without his continuous encouragement.

During the first four years of my PhD study, I am fortunate to work as a research trainee in SAP Innovation Center network, Singapore, where I have been supervised by Dr. Falk Brauer, Dr. Hoang Tam Vo and Dr. Daniel Dahlmeier at a different period. I would like to thank them for their patience in teaching me the basic knowledge of doing research and sharing their past experiences of purchasing the PhD. I feel grateful to have them to be my mentors who always encouraged me whenever I face difficulties during
research and works. I also need to thank Steve Lee, the director of SAP ICN at that time for agreeing to offer me the great SAP PhD Scholarships. Their professional attitude on doing research has undoubtedly encouraged me to dive deeper into the research world.

I feel extremely fortunate to have Prof. Chee Yong Chan, Prof. Yong Meng Teo and Prof. Felix Xiao Zhu Lin to be my thesis examiners. I also thank Prof. Xiao Kui Xiao for being the head of department (HOD) represent of my oral defense. Their insightful comments helped significantly improve the quality of this thesis.

There are many others that I need to appreciate during my PhD journey. I am lucky to be a member of the Xtra Computing Group lead by Prof. He, where I made a wonderful friendship with the group members. I’m also glad to have been working with my seniors, Thomas Heinze, Jiong He, Chi Zhou, Zeke Wang, Feng Zhang, and Yingjun Wu. Their continuously kind advice and feedback greatly help me to improve my research skills.

Last, but not least, I need to thank my family for their enduring support throughout this long and stressful period. I especially want to thank my wife, Mengjie Zhang, for her continuous encouragement, support and love.
Abstract

Data stream processing systems (DSPSs) enable users to express and run stream applications to continuously process data streams. Witnessing the emergence of modern commodity machines with massively parallel processors, researchers and practitioners find shared-memory multicore architectures an attractive platform for DSPSs. However, fully exploiting the computation power delivered by multicore architectures can be challenging, and scaling DSPSs on modern multicore architectures remains to be notoriously challenging. This is because processing massive amounts of data can confront several performance bottlenecks inherited from different DSPS components, which altogether put a strict constraint on the scalability of the DSPSs on multicore architectures.

In this thesis, we present the evaluation, design and implementation of DSPSs aiming at high-performance stream processing on multicore architectures. First, as modern DSPSs are mainly designed and optimized for scaling-out using a cluster of low-end servers, we present a comprehensive performance study of popular DSPSs (e.g., Apache Storm, Flink) on multicore architectures and try to identify whether the current implementation matches with modern hardware (e.g., non-uniform memory access – NUMA, multicore). Second, our detailed profiling study shows that existing DSPSs severely underutilized the underlying complex hardware micro-architecture and especially show poor scalability due to the unmanaged resource competition and unaware of NUMA effect. We hence present our efforts on a complete revolution in designing next-generation stream processing platform, namely BriskStream, specifically optimized for shared-memory multicore architectures. A novel NUMA-aware execution plan optimization scheme, namely Relative-Location-Aware-Scheduling (RLAS) is proposed to address the NUMA effect for stream computation.
Third, DSPS with transactional state management relieves users from managing state consistency by themselves. We introduce TStream, a DSPS with built-in efficient transactional state management. Compared to previous works, it guarantees the same consistency properties while judiciously exploits more parallelism opportunities – both within the processing of each input event and among a (tunable) batch of input events. To confirm the effectiveness of our proposal, we compared TStream against three prior solutions on a four-socket multicore machine. Our extensive experiment evaluations show that TStream yields up to 6.8 times higher throughput comparing with existing approaches with similar or even lower end-to-end processing latency for varying application workloads.
Contents

List of Figures xiii
List of Tables xv

1 Introduction 1

2 Preliminaries 5
  2.1 Multi-Socket Multi-core Processors 5
    2.1.1 Complex execution pipelines of modern processor 5
    2.1.2 Non-uniform memory access (NUMA) 7
  2.2 Data Stream Processing Systems 8
  2.3 Data Stream Processing Benchmark 10
  2.4 Consistent Stateful Stream Processing 14

3 Literature Review 19
  3.1 DSPSs on Modern Hardware 19
    3.1.1 Multicore Stream Processing 20
    3.1.2 GPU-Enabled Stream Processing 21
    3.1.3 FPGA-Enabled Stream Processing 22
  3.2 Key Issues of Data Stream Processing 23
    3.2.1 Execution Optimization Techniques 23
    3.2.2 Out-of-order handling 25
    3.2.3 State Management in DSPSs 26
  3.3 Performance Evaluation for DSPSs 28

4 Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors 31
  4.1 Introduction 31
  4.2 Methodology 34
    4.2.1 Evaluation Goals 34
4.2.2 Profiling Tools 34
4.3 Performance Evaluation 35
4.4 Study the impact of common designs 38
  4.4.1 Execution time breakdown 39
  4.4.2 Massively parallel execution model 40
  4.4.3 Message passing and stream partitioning 42
  4.4.4 JVM Runtime Environment 44
4.5 Towards more efficient DSP systems 46
  4.5.1 Non-blocking Tuple Batching 46
  4.5.2 NUMA-Aware Executor Placement 48
  4.5.3 Put It All Together 51

5 BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures 53
  5.1 Introduction 53
  5.2 Execution Plan Optimization 55
    5.2.1 The Performance Model 57
    5.2.2 Problem Formulation 62
  5.3 Optimization Algorithm Design 63
  5.4 Algorithm Details 68
    5.4.1 Discussion 69
    5.4.2 Extension with other optimization techniques 72
  5.5 BriskStream System 72
    5.5.1 Improving Execution Efficiency 73
    5.5.2 Improving Communication Efficiency 74
    5.5.3 Discussion on Elasticity 74
  5.6 Implementation details 75
  5.7 Evaluation 76
    5.7.1 Experimental Setup 77
    5.7.2 Performance Model Evaluation 79
    5.7.3 Evaluation of Execution Efficiency 81
    5.7.4 Evaluation of RLAS algorithms 86
    5.7.5 Factor Analysis 92
6 Scaling Consistent Stateful Stream Processing on Shared-Memory Multicore Architectures

6.1 Introduction 93
6.2 Existing Solutions Revisited 96
6.3 TStream Overview 98
6.4 TStream Design 99
   6.4.1 Programming model 99
   6.4.2 Punctuation Signal Slicing Scheduling 100
   6.4.3 Fine-grained Parallel State Access 103
   6.4.4 More Processing Optimizations 105
6.5 Discussion 106
6.6 Evaluation 107
   6.6.1 Experimental Setup 107
   6.6.2 Benchmark Workloads 108
   6.6.3 Overall Performance Comparison 111
   6.6.4 Workload Sensitivity Study 114
   6.6.5 System Sensitivity Study 117

7 Conclusion 119
   7.1 Contributions 119
   7.2 Discussions 120

8 Future Directions 123

9 Publications 125
   9.1 Selected Publications 125
   9.2 Other Publications 125
## List of Figures

2.1 Pipeline execution components of processor: (left) various stalls caused in the pipeline, (middle) pipelines interactions with cache and memory systems and (right) the interactions among the cache, TLB, and memory systems. 6  
2.2 NUMA topology and peak bandwidth of our four-sockets server. 7  
2.3 Interconnect topology for our eight-sockets servers. 8  
2.4 Message passing mechanism. 9  
2.5 Word-count execution graph. 9  
2.6 Topologies of seven applications in our benchmark. *DataSource* can be further divided into a linear chain of *Spout* and *Parser* operators. 12  
2.7 Toll Processing (*TP*). 15  
4.1 Performance evaluation results on Storm and Flink. 38  
4.2 Execution time breakdown. 39  
4.3 Front-end stall breakdown. 40  
4.4 Instruction footprint between two consecutive invocations of the same function. 41  
4.5 Varying number of executors of Map-Matcher operator of TM when running Storm with four CPU sockets. 44  
4.6 Back-end stall breakdown. 45  
4.7 Normalized throughput of tuple batching optimization. 49  
4.8 Normalized latency of tuple batching optimization. 49  
4.9 NUMA-aware executor placement. 51  
4.10 Normalized throughput with all optimizations enabled. 51  
5.1 Word Count (WC) as an example application. 56  
5.2 CDF of profiled average execution cycles of different operators of WC. 61  
5.3 RLAS Optimization example. 64
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>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.</td>
</tr>
<tr>
<td>5.5</td>
<td>Example job overview in BriskStream.</td>
</tr>
<tr>
<td>5.6</td>
<td>Throughput speedup.</td>
</tr>
<tr>
<td>5.7</td>
<td>End-to-end latency of WC on different DSPSs.</td>
</tr>
<tr>
<td>5.8</td>
<td>Scalability evaluation.</td>
</tr>
<tr>
<td>5.9</td>
<td>Gaps to ideal.</td>
</tr>
<tr>
<td>5.10</td>
<td>Execution time breakdown.</td>
</tr>
<tr>
<td>5.11</td>
<td>Comparing with StreamBox.</td>
</tr>
<tr>
<td>5.12</td>
<td>RLAS w/ and w/o considering varying RMA cost.</td>
</tr>
<tr>
<td>5.13</td>
<td>Placement strategy comparison under the same replication configuration.</td>
</tr>
<tr>
<td>5.14</td>
<td>CDF of random plans.</td>
</tr>
<tr>
<td>5.15</td>
<td>Communication pattern matrices of WC on two Servers.</td>
</tr>
<tr>
<td>5.16</td>
<td>A factor analysis for BriskStream. Changes are added left to right and are cumulative.</td>
</tr>
<tr>
<td>6.1</td>
<td>Issues of prior solutions.</td>
</tr>
<tr>
<td>6.2</td>
<td>AUX_Tuple maintenance.</td>
</tr>
<tr>
<td>6.3</td>
<td>Example workflow of switching between phases.</td>
</tr>
<tr>
<td>6.4</td>
<td>Transaction decomposition example.</td>
</tr>
<tr>
<td>6.5</td>
<td>Three more applications in our benchmark workloads.</td>
</tr>
<tr>
<td>6.6</td>
<td>Throughput (log-scale) comparison of different applications under different execution schemes. There are 25% multi-partition transactions for GS, SL and OB.</td>
</tr>
<tr>
<td>6.7</td>
<td>Runtime breakdown per state transaction of TP.</td>
</tr>
<tr>
<td>6.8</td>
<td>Multi-partition transaction evaluation.</td>
</tr>
<tr>
<td>6.9</td>
<td>Varying application workload configurations of GS.</td>
</tr>
<tr>
<td>6.10</td>
<td>Effect of varying punctuation interval.</td>
</tr>
<tr>
<td>6.11</td>
<td>99th percentile end-to-end processing latency.</td>
</tr>
<tr>
<td>6.12</td>
<td>Effect of work-stealing.</td>
</tr>
<tr>
<td>6.13</td>
<td>Varying processing configurations.</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Summary of Terminologies 14
4.1 JVM profile flags 35
4.2 Processor measurement components 36
4.3 Detailed specification on our testing environment 36
4.4 CPU and memory bandwidth utilization on a single CPU socket 37
4.5 LLC miss stalls when running Storm with four CPU sockets 43

5.1 Summary of terminologies 58
5.2 Operator selectivity of LR 78
5.3 Characteristics of the two servers we use 79
5.4 Average processing time per tuple (T) under varying NUMA distance. The unit is nanoseconds/tuple 80
5.5 Model accuracy evaluation of all applications. The performance unit is K events/sec 81
5.6 99-percentile end-to-end latency (ms) 83
5.7 Placement strategies 88
5.8 Tuning compression ratio (r) 91
CHAPTER 1

Introduction

Data stream processing system (DSPS) is a software that allows users to run their streaming applications which continuously process infinite data streams in real-time. Unlike conventional database management systems (DBMSs), modern DSPSs are featured in supporting continuous lower-latency analytics over real-time data streams. Due to its unique characteristics, a large body of system research [CFMKP13, WT15, GSHW14, GSS15, CKE+15, TTS+14, KBF+15] has focused on designing and implementing new DSPSs to meet the fast increasing and more and more diverging application demands. Arguably starting from 2000’s, DSPSs have been investigated by a large number of research groups in the database community [ACc+03, CCD+03, AAB+05, NRNK10, TTS+14, KBF+15, CKE+15, DZSS14, NPP+17, NRNK10], and leading enterprises including SAP [ZVDH17], IBM [GAW+08], Google [ABB+08] and Microsoft [CGB+14]. With the proliferation of high-rate streaming sources, numerous streaming applications are deployed in real-world use cases [Tra18] that involve continuously low-latency, complex analytics over massive data streams. This trend has accelerated the development of next-generation performance-critical DSPSs.

Most existing DSPSs are designed and optimized for scaling out using a cluster of low-end machines (e.g., [TTS+14, CKE+15, XCTS14, GSHW14, ABQ13, PHH+15]). In particular, substantial research efforts have been devoted on providing mechanisms to handle the inherent challenges from the distributed environment settings, such as dueling with network communication overhead [XCTS14, ABQ13, PLS+06], fault-tolerance [WT15, DZSS14, CFMKP13, CEF+08] and elastic scaling [GSHW14, HJHF14]. Despite the successes achieved during the last several decades, these DSPSs are now facing great challenges when supporting a wide range of emerging time-critical
Chapter 1. Introduction

applications, which generally require the underlying DSPSs to achieve low end-to-end latency when processing huge volumes of data. Any unpredictable latency spikes (e.g., in TCP/IP network) inside the distributed DSPS can cause serious issues in those applications such as hospital infection-control monitoring and online abnormally trajectory detection.

Witnessing the emergence of modern commodity machines with massively parallel processors, researchers and practitioners find shared-memory multicore architectures an attractive alternative platform [ZHD+04, CDW17], and several in-memory single-node DSPSs are recently proposed [KWCF+16, MPJ+17, ZHZH19]. One of the key benefits of single-node DSPSs is the completely avoidance of network communication latency among multi-node inside the DSPSs. In addition to that, several heavyweight components, such as (de)serialization can be completely avoided, which both simplify the system development and improve execution efficiency. Furthermore, optimizing the performance of stream processing on a single-node is critical even in a distributed configuration for an obvious reason – it reduces the number of machines required to achieve the same performance objective.

Thanks to the great achievements made in the hardware community, modern commodity machines are now equipped with massively parallel processors and larger memory capacity and demonstrated superior performance for real-world applications [AGN+13]. For example, recent scale-up servers can accommodate even hundreds of CPU cores and multi-terabytes of memory [MF17] providing abundant computing resources and emerging network technologies such as Remote Direct Memory Access (RDMA) and 10Gb Ethernet significantly improve system ingress rate making I/O no longer a bottleneck in many practical scenarios [MPJ+17, CDW17]. However, fully exploiting the computation power delivered by multicore architectures can be challenging. On one hand, the on-chip cache hierarchies that support large core counts are getting larger, deeper, and more complex to utilize. Furthermore, as modern machines scale to multiple sockets, non-uniform memory access (NUMA) becomes an important performance factor for data management systems (e.g., [LBKN14, LPM+13]). On the other hand, little work has been done on studying common design aspects of modern DSPSs on shared-memory multicore architectures.
To fully understand the potential difficulties that may be confronted when building modern multicore main-memory DSPSs, we study the performance bottlenecks inherited from three common design aspects in different modern DSPSs. Then, we discuss a novel execution plan optimization scheme addressing NUMA effect to improve scalability of DSPSs on multicore architectures. After that, we discuss a new scalable design of transactional state management in DSPSs.

The discussion on the different aspects discussed above illustrate the challenges we may confront in building a high performance DSPSs that can effectively utilize modern multicore architectures. These aspects are tightly coupled with each other, and the redesign of a single component can directly affect the effectiveness of others. Witnessing these problems, in this thesis, we study the problem of building scalable multicore DSPSs from a systematic perspective. In particular, we discuss the design and implementation of two core components of DSPSs, including execution plan optimization and state management. Throughout this thesis, we conduct comprehensive performance study and propose novel mechanisms to address the issues identified above. In addition, we also point out future directions in designing and implementing next-generation high-performance DSPSs.

The road map of the proposed thesis are listed as follows.

- **Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors.** This work has been published in a conference paper [ZHD+04]. We revisit three common design aspects of modern DSPSs on modern multi-socket multi-core architectures, including a) pipelined processing with message passing, b) on-demand data parallelism, and c) JVM-based implementation. Particularly, we conduct detailed profiling studies with micro benchmark on Apache Storm and Flink. Our results show that those designs have underutilized the scale-up architectures in these two key aspects: a) The design of supporting both pipelined and data parallel processing results in a very complex massively parallel execution model in DSPSs, which causes high front-end stalls on a single CPU socket; b) The design of continuous message passing mechanisms between operators severely limits the scalability of DSPSs on multi-socket multi-core architectures. We further present two optimizations to address those performance issues and demonstrate
Chapter 1. Introduction

promising performance improvements.

• **BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures.** This work has been published in a conference paper [ZHZH19]. We introduce 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 a 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.

• **Scaling Consistent Stateful Stream Processing on Shared-Memory Multicore Architectures.** This work is submitted to a conference [ZWZH19]. We introduce TStream with a new design for scaling transactional state management in stream processing. In order to take advantage of multicore architectures, TStream detaches the state management from the streaming computation logic, and performs its internal state maintenance asynchronously. By eliminating the expensive synchronization primitives, TStream aggressively extracts parallelism opportunities by revealing the operation dependencies at runtime. Our experimental results show that TStream achieves up to 6.8x higher throughput over existing solutions with similar or even smaller end-to-end processing latency at the scale of millisecond.

The outline of this thesis is listed as follows. We begin in Chapter 2 a preliminary background information of DSPSs and multicore architectures we are targeting at. Chapter 3 covers a comprehensive literature review of the state-of-the-art mechanisms in the design and implementation of DSPSs. In Chapter 4, we then provide a detailed evaluation of modern DSPSs on shared-memory multicore architectures. In Chapter 5, we present BriskStream with a novel NUMA-aware execution plan optimization scheme. We next discuss TStream in Chapter 6, a new DSPS with built-in scalable transactional state management on multicores. We summarize this thesis in Chapter 7. In Chapter 8, we provide some suggestions in future work of enhancing DSPSs.
Preliminaries

In this section, we first introduce the background of the multi-socket multi-core processors. Then, we introduce three design aspects of two DSPSs we studied, namely Apache Storm [TTS+14] and Flink [CKE+15]. After that, we introduce our designed micro-benchmark for data stream processing systems covering seven popular applications. Finally, we motivate consistent stateful stream processing with a running example.

2.1 Multi-Socket Multi-core Processors

2.1.1 Complex execution pipelines of modern processor

Modern processors consist of multiple different hardware components with deep execution pipelines, as shown in Figure 2.1. We also illustrate the stalls and the interactions among pipelines and memory systems in the figure. The pipeline can be divided into the front-end component and the back-end component [Int19].

The front-end is responsible for fetching instructions and decodes them into micro-operations (μops). It feeds the next pipeline stages with a continuous stream of micro-ops from the path that the program will most likely execute, with the help of the branch prediction unit. The front-end component is composed of the following parts. First, the instruction fetch units are responsible for instruction prefetching from L1-ICache. Second, the Instruction Length Decoder (ILD) performs pre-decoding on the fetched instructions. Third, the instruction queue (IQ) is used to store pre-decoded instructions. Fourth, the instruction decode units, i.e., instruction decoders and instruction decode
Figure 2.1: Pipeline execution components of processor: (left) various stalls caused in the pipeline, (middle) pipelines interactions with cache and memory systems and (right) the interactions among the cache, TLB, and memory systems.

queue (IDQ), are responsible for decoding instructions stored in IQ into $\mu$ops.

Starting from Sandy Bridge micro-architecture, Intel introduces a special component called Decoded ICache (D-ICache), which is essentially an accelerator of the traditional front-end pipeline. D-ICache maintains up to $1.5k$ of decoded $\mu$ops. Future references to the same $\mu$ops can be served by it without performing the fetch and decode stages. D-ICache is continuously enhanced in terms of size and throughput in the successor generations of Intel processors. Note that, every $\mu$ops stored in D-ICache is associated with its corresponding instruction in L1-ICache. An L1-ICache miss also causes D-ICache to be invalidated.

The **back-end** is where the actual instruction execution happens. It detects the dependency chains among the decoded $\mu$ops (from IDQ or the D-ICache), and executes them in an out-of-order manner while maintaining the correct data flow.
2.1.2 Non-uniform memory access (NUMA)

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. Figure 2.2 illustrates the NUMA topology of our sever with four sockets. Each CPU socket has its local memory, which is uniformly shared by the cores on the socket. Sockets are connected by a much slower (compared to local memory access) channel called Quick Path Interface (QPI).

Different NUMA configurations exist nowadays market, which further complicates the software optimization on them. Figure 2.3 illustrates the NUMA topologies of two of our eight-sockets 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 [HPE18]) 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 5.7).
2.2 Data Stream Processing Systems

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

**Pipelined processing with message passing.** We describe the execution model with a general definition [GSS15]. A streaming application is represented by a graph, where nodes in the graph represent either data source operators or data processing operators, and edges represent the data flow between operators. In general, there are two types of operators defined in the topology. 1) a *data source* operator generates (or receives from the external environment) events to feed into the topology, and 2) a *data processor* operator encapsulates specific processing logics such as filtering, transforming or user-defined function.

In a shared-memory environment, an operator (continuously) writes its output data into the local memory. For each output data, the operator also pushes a tuple consisting of a reference (i.e., pointer) of the output data into its output queue. The corresponding consumer (continuously) fetches the tuple from the queue, and then accesses on the output data generated by the producer operator through memory fetches. In other words, the communication between two operators are through the data reference. This pass-by-reference message passing approach avoids duplicating data in a shared-memory environment and is the common approach adopted by most modern DSP
systems. Figure 2.4 illustrates an example of message passing between operators in a shared-memory environment, where the producer and consumer are scheduled to CPU socket 0 and socket 1, respectively. The producer first writes its output data to the local memory of socket 0 (step 1) and emits a tuple containing a reference to the output data to its output queue (step 2). The consumer fetches from the corresponding queue to obtain the tuple (step 3) and then accesses the data by the reference (step 4). This example also demonstrates remote memory accesses across sockets during message passing in DSP systems, which we will study in details in Section 4.4.3.

**On-demand data parallelism.** Modern DSP systems such as Storm and Flink are designed to support task pipeline and data parallelism at the same time. The actual execution of an operator is carried out by one or more physical threads, which are referred to as executors. Input stream of an operator is (continuously) partitioned among its executors. The number of executors for a certain operator is referred to
as the *parallelism level* and can be configured by users in the topology configuration. A topology at the executor level is called an execution graph. An example execution graph of the word-count application is shown in Figure 2.5. In this example, the split, count, and sink operators have three, two and one executors, respectively. Streams are partitioned and delivered to specific destination executors according to the grouping strategy specified by the user. In the previous example, the shuffle grouping strategy used in the data source operator uniformly distributes the tuples to each split executor. Meanwhile, each split executor sends tuples to count executors according to the attribute of the tuple (specified as field grouping) so that the same key (i.e., the same word) is always delivered to the same count executor.

**JVM-based implementation.** Both Storm and Flink are implemented with JVM-based programming languages (i.e., Closure, Java, and Scala), and their execution relies on JVM. Two aspects of JVM runtime are discussed as follows.

*Data reference:* As we have mentioned before, message passing in DSP systems always involves passing the reference. That is, operators access the data through the reference in the tuple, which may lead to pointer chasing and stress the cache and TLB of the processor heavily.

*Garbage collection (GC):* Another important aspect of the JVM-based system is the built-in memory management. Modern JVM implements generational garbage collection which uses separate memory regions for different ages of objects. The significant overhead of GC has been reported in many existing studies (e.g., [SP14, ABVA15]). To this end, some DSP systems have even implemented its own memory management besides JVM (e.g., Flink).

### 2.3 Data Stream Processing Benchmark

In order to examine existing DSP systems as well as to verify the effectiveness of our proposed optimization techniques, we design our streaming benchmark according to the four criteria proposed by Jim Gray [Gra92]. As a start, we design the benchmark consisting of seven streaming applications including Stateful Word Count (WC), Fraud Detection (FD), Spike Detection (SD), Traffic Monitoring (TM), Log Processing (LG),
Chapter 2. Preliminaries

Spam Detection in VoIP (VS), and Linear Road (LR).

We briefly describe how they achieve the four criteria. 1) Relevance: the applications cover a wide range of memory and computational behaviors, as well as different application complexities so that they can capture the DSP systems on scale-up architectures; 2) Portability: we describe the high-level functionality of each application, and they can be easily applied to other DSP systems; 3) Scalability: the benchmark includes different data sizes; 4) Simplicity: we choose the applications with simplicity in mind so that the benchmark is understandable.

Our benchmark covers different aspects of application features. First, our applications cover different runtime characteristics. Specifically, TM has highest CPU resource demand, followed by LR, VS and LG. CPU resource demand of FD and SD is relatively low. The applications also have variety of memory bandwidth demands. Second, topologies of the applications have various structural complexities. Specifically, WC, FD, SD, and TM have single chain topologies, while LG, VS, and LR have complex topologies. Figure 2.6 shows the topologies of the seven applications.

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

Stateful Word Count (WC): The stateful word-count counts and remembers the frequency of each received word unless the application is killed. The topology of WC is a single chain composed of a Split operator and a Count operator. The Split operator parses sentences into words and the Count operator reports the number of occurrences for each word by maintaining a hashmap. This hashmap is once created in the initialization phase and is updated for each receiving word. The input data of WC is a stream of string texts generated according to a Zipf-Mandelbrot distribution (skew set to 0) with a vocabulary based on the dictionary of Linux kernel (3.13.0-32-generic).

Fraud Detection (FD): Fraud detection is a particular use case for a type of problems known as outliers detection. Given a transaction sequence of a customer, there is a probability associated with each path of state transition, which indicates the chances of fraudulent activities. We use a detection algorithm called $missProbability$ \[Gho13\] with sequence window size of 2 events. The topology of FD has only one operator, named
Figure 2.6: Topologies of seven applications in our benchmark. DataSource can be further divided into a linear chain of Spout and Parser operators.

as Predict, which is used to maintain and update the state transition of each customer. We use a sample transaction with 18.5 million records for testing. Each record includes customer ID, transaction ID, and transaction type.

Log Processing (LG): Log processing represents the streaming application of performing real-time analyzing on system logs. The topology of LG consists of four operators. The Geo-Finder operator finds out the country and city where an IP request is from, and the Geo-Status operator maintains all the countries and cities that have been found so far. The Status-Counter operator performs statistics calculations on the status codes of
HTTP logs. The Volume-Counter operator counts the number of log events per minute. We use a subset of the web request data (with 4 million events) from the 1998 World Cup Web site [AJ00]. For data privacy protection, each actual IP address in the requests is mapped to randomly generated but fixed IP address.

**Spike Detection (SD):** Spike detection tracks measurements from a set of sensor devices and performs moving aggregation calculations. The topology of SD has two operators. The Moving-Average operator calculates the average of input data within a moving distance. The Spike-Detection operator checks the average values and triggers an alert whenever the value has exceeded a threshold. We use the Intel lab data (with 2 million tuples) [Mad04] for this application. The detection threshold of moving average values is set to 0.03.

**Spam Detection in VoIP (VS):** Similar to fraud detection, spam detection is a use case of outlier detection. The topology of VS is composed of a set of filters and modules that are used to detect telemarketing spam in Call Detail Records (CDRs). It operates on the fly on incoming call events (CDRs), and keeps track of the past activity implicitly through a number of on-demand time-decaying bloom filters. A detailed description of its implementation can be found at [BdN11]. We use a synthetic data set with 10 million records for this application. Each record contains data on a calling number, called number, calling date, answer time, call duration, and call established.

**Traffic Monitoring (TM):** Traffic monitoring performs real-time road traffic condition analysis, with real-time mass GPS data collected from taxis and buses\(^1\). TM contains a Map-Match operator which receives traces of an object (e.g., GPS loggers and GPS-phones) including altitude, latitude, and longitude, to determine the location (regarding a road ID) of this object in real-time. The Speed-Calculate operator uses the road ID result generated by Map-Match to update the average speed record of the corresponding road. We use a subset (with 75K events) of GeoLife GPS Trajectories [ZZXM09] for this application.

**Linear Road (LR):** Linear Road (LR) is used for measuring how well a DSP system can meet real-time query response requirements in processing a large volume of streaming and historical data [ACG+04, JAA+06]. It models a road toll network, in which tolls

---

\(^1\) A real deployment at http://210.75.252.140:8080/infoL/sslk_weibo.html.
depend on the time of the day and level of congestions. Linear Road has been used by many DSP systems, e.g., Aurora [ACc+03], Borealis [AAB+05], and System S [GAW+08]. LR produces reports of the account balance, assessed tolls on a given expressway on a given day, or estimates cost for a journey on an expressway. We have followed the implementation of the previous study [SC14] for LR. Several queries specified in LR are implemented as operators and integrated into a single topology. The input to LR is a continuous stream of position reports and historical query requests. We merge the two data sets obtained from [ACG+04] resulting in 30.2 million input records (including both position reports and query requests) and 28.3 million historical records.

### 2.4 Consistent Stateful Stream Processing

In this section, we use a simplified toll processing query (TP) from the Linear Road Benchmark [ACG+04] as a running example to motivate consistent stateful stream processing. TP calculates a toll every time a vehicle reports a position in a new road segment, in which tolls depend on the level of congestion of the road. The congestion status (i.e., average speed and count of vehicles) of a road segment are maintained and updated for every vehicle position report (i.e., input events).
One common way to implement TP is illustrated in Figure 2.7a. Each vertex denotes an operator, and edge denotes data flow between operators. **Parser** parses each input event into position report containing <timestamp, geographic position, speed>. Each position report is broadcast to three downstream operators: 1) **Road Speed** computes average traffic speed of a road segment; 2) **Vehicle Cnt** computes the average number of unique vehicles of a road segment; 3) The computed road statistics are passing to **Toll Notification**, which computes the toll of a vehicle referencing to the traffic speed and number of unique vehicles of the road segment, where the vehicle is. The toll report is continuously send to **Sink** for output.

The congestion status (i.e., speed and count) of road segments are **application states**, which need to be maintained for future reference during processing. In most of today’s DSPSs, there are two challenges requiring the application developer to handle by themselves. *Firstly*, to sustain high input stream ingress rate, each operator need to be carried in multiple **executors** (e.g., Java threads), which handle multiple input events concurrently. Subsequently, in order to ensure **state consistency**, report of the same road must be always handled by the same executor of **Road Speed**, **Vehicle Cnt** and **Toll Notification** operator, respectively. This is commonly achieved by key-based
stream partitioning (e.g., using geo-position value of each position report as the key to partition position report stream) and is not generally suitable. For example, suppose the processing of one position report needs to access not only the road segment of where the current report belonging, but also other road segments information (e.g., if nearby roads are empty, pay extra penalty for driving on the current road segment) which may be maintained by other executors. Then, the executor needs to access application states maintained by other executors (e.g., through status broadcasting), which involves costly synchronization and communication overhead. A skewed stream will further worse the performance [KLC08]. Secondly, the system needs to always ensure that the position report passed to any executor of Toll Notification is processed only after it receives the updated road congestion status from Road Speed and Vehicle Cnt operators. Unfortunately, as all operators (and their executors) are running independently, such ordering constraint is challenging to preserve. Most existing DSPSs left such burden to developer, and it is commonly achieved by manually embedding tuple buffering and sorting operations (i.e., the Sort operation in Figure 2.7a) inside the operator logic [SC14], which is error prone and involves significant performance penalty.

To relieve users from managing state consistency by themselves, consistent shared state management [AMC17] are recently emerged. We follow previous works [MTZ+15, AMC17] of employing transactional semantics [BG81] on managing shared state access. Specifically, we adopt the following two definitions from previous studies.

**Definition 1** (state transaction). We define the set of state accesses triggered by processing of a single input event $e_{ts}$ at one executor as one state transaction, denoted as $txn_{ts}$. Timestamp $ts$ of a state transaction is defined as the same as its triggering event.

Concurrent accesses to shared state can cause data inconsistency. Previous studies [AMC17, BFKT12, MTZ+15] advocate that the execution of multiple state transactions concurrently should satisfy ACID properties. In addition, it is also required that state access order follows event timestamp ordering (please refer to [AMC17] for a detailed description).

**Definition 2** (consistent property). We define a consistency property satisfying both ACID properties and ordering constraint as ACID+O properties. A DSPS under ACID+O needs to ensure the state transaction schedule must be conflict equivalent to $txn_1 \prec \ldots \prec txn_n$. 
An implementation of TP utilizing consistent shared state management is illustrated in Figure 2.7b. It is implemented with the same operators but road congestion status are shared among all operators (and their executors). Specifically, the road status are maintained in two tables, one for maintaining average road speed, and one for count of unique vehicles. Both tables are read- and write-able by all executors of Road Speed, Vehicle Cnt and Toll Notification operators. The key of both table is the road segment ID. Input events can be round-robin shuffled among all executors to process concurrently to ensure load balancing, and the state consistency is enforced by the system through wisely managing concurrent state accesses to shared states and enforcing the aforementioned ACID+O properties.
CHAPTER 3

Literature Review

The design and implementation of DSPSs have attracted a great amount of effort from both the research and the industry communities during the last decade. In this chapter, we provide a comprehensive and multidisciplinary literature review on DSPSs. Instead of performing a survey of all related works, we focus on those closely related to our works.

3.1 DSPSs on Modern Hardware

In the last decade, a great number of different academic prototypes as well as commercial products of DSPSs have been built. The first-generation of DSPSs were often built as extensions of existing database engines. Representatives of those earlier DSPSs including TelegraphCQ [CCD+03], Aurora [ACc+03] and STREAM [G+03, ABW06]. SQL-variant programming language (e.g., CQL [ABW06]) is proposed together with the system. Due to the hardware limitation, first-generation DSPSs primarily focus on single-core execution. Driven by new demands of streaming applications, second-generation DSPSs, such as Borealis [AAB+05], System-S [GAW+08], are equipped with advanced features such as fault tolerance [AAB+05], adaptive query processing [RDS+04] and more complex query expressions [BCM06] (e.g., complex event processing). Modern DSPSs (i.e., third-generation) are strongly driven by the trend towards cloud computing. Two key features of them are scaling over a cluster of machines and highly robustness on faults. Some important examples including Apache Storm [TTS+14], Flink [CKE+15], Spark Streaming [DZSS14], Samza [NPP+17] and S4 [NRNK10]. Despite their different architectures and design focuses, majority of them are designed with the aim of
optimizing the performance of DSPs in a distributed environment.

Great achievements have been made in the computer architecture community. Modern commodity machines are now equipped with hundreds-cores processors [MF17], massively parallel accelerators, multi-terabytes of memory capacity [ZHZH19], and gigabytes per second network bandwidth [CDW17]. To fulfill the fast growing demand in handling real-time streams, researchers and practitioners have investigated and proposed many hardware-conscious DSPs in exploiting the potential of accelerating stream processing on modern hardware. However, fully utilizing hardware capacity is notoriously challenging, and a large number of studies were proposed in recent years (e.g., [CGB+14, KWCF+16, NSJ16, GBY09, KCDP16, SAG+10, ZHD+04, MPJ+17, ZHZH19, ZMK+01]). This thesis aims to push further the state-of-the-art of hardware-conscious DSPs. In the following, we survey related works on accelerating stream processing by utilizing multicore CPUs, GPUs and FPGAs.

3.1.1 Multicore Stream Processing

**Language and Compiler.** Multicore architectures have been ubiquitous. However, programming models and compiler techniques for employing multicore features are still lag behind hardware improvements. Kudlur et al. [KM08] were among the first to develop a compiler technique to map stream application to a multicore processor. By taking the Cell processor as an example, they study how to compile and run a stream application expressed in their proposed language StreamIt. The compiler works in two steps, 1) operator fission optimization (i.e., split one operator into multiple ones) and 2) assignment optimization (i.e., assign each operator to a core). The two-step mapping is formulated as an integer linear programming (ILP) problem and requires a commercial ILP solver. Noting its NP-Hardness, Farhad et al. [FKBS11] later presented an approximation algorithm to solve the mapping problem. Note that, the mapping problem from Kudlur et al. [KM08] considers only CPU loads, and ignores communications bandwidth. In response, Carpenter et al. [CRA09] developed an algorithm that maps a streaming program onto a heterogeneous target, further taking communication into consideration. To utilize a SIMD-enabled multicore system, Hormati et al. [HCW+10] proposed to vectorize stream applications. Relying on high-
level information, such as the relationship between operators, they were able to achieve better performance than general vectorization techniques. Agrawal et al. [AFK+12] proposed a cache-conscious scheduling algorithm for mapping stream application on multicore processors. Particularly, they developed the theoretical lower bounds on cache misses when scheduling a streaming pipeline on multiple processors, and the upper bound of the proposed cache-based partitioning algorithm called seg_cache. They also experimentally found that scheduling solely based on the modelled cache effects can be often more effective than the conventional load-balancing (based on computation cost) approaches.

**Multicore-aware DSPSs.** Recently, there has been a fast growing interest in building multicore-friendly DSPSs. Instead of statically compiling a program as done in StreamIt [KM08, FKBS11, CRA09], these DSPSs provide better elasticity for application execution. They also allow the usage of general purpose programming languages (e.g., Java, Scala) to express stream applications. Tang et al. [TG13] studied the data flow graph to explore the potential parallelism in a DSPS and proposed an auto-pipelining solution that can utilize multicore processors to improve the throughput of stream processing applications. For economic reasons, power efficiency has become more and more important in recent years, especially in HPC domains. Kanoun et al. [KRAVDS14] proposed a multicore scheme for stream processing that takes power constraint into consideration. Trill [CGB+14] is a single-node query processor for temporal or streaming data. Contrary to most distributed DSPSs (e.g., Storm, Flink) adopting continuous operator model, Trill runs the whole query only on the thread that feeds data to it. Such an approach has shown to be especially effective when applications contain no synchronization barriers [ZMK+01].

### 3.1.2 GPU-Enabled Stream Processing

GPUs are the most popular heterogeneous processors due to their high compute capacity and cost-effectiveness. However, due to their unique execution model, special designs are required to efficiently adapt stream processing to GPUs.

**Single-GPU.** Verner et al. [VSS11] presented a general algorithm for processing data streams with real-time stream scheduling constraint on GPUs. This algorithm assigns...
data streams to CPUs and GPUs based on their incoming rates. It tries to provide an assignment that can satisfy different requirements from various data streams. Zhang et al. [ZHH15] developed a holistic approach to build DSPSs using GPUs. They design a latency-driven GPU-based framework, which mainly focuses on real-time stream processing. Due to the limited memory capacity of GPUs, the window size of the stream operator plays an important role in system performance. Pinnecke et al. [PBS15] studied the influence of window size, and proposed a partitioning method for splitting large windows into different batches, considering both time and space efficiency. SABER [KWCF+16] is a window-based hybrid stream processing framework using GPUs.

**Multi-GPUs.** Multi-GPUs systems provide tremendous computation capacity, but also pose challenges like how to partition or scheduling among different GPUs. Verner et al. [VSSM12] extend their method [VSS11] to a single node with multiple GPUs. A scheduler manipulates stream placement and guarantees that the requirements among different streams can be met. GStream [ZM11] is the first data streaming framework for GPU clusters. GStream supports stream processing applications in the form of a C++ library; it uses MPI to implement the data communication between different nodes, and uses CUDA to conduct stream operations on GPUs. In addition, GStream includes a language abstraction for users to describe different applications. Alghabi et al. [ASK15] first introduced the concept of stateful stream data processing on a node with multiple GPUs. This work describes processing graph-like arrangements of different processing modules. Nguyen et al. [NL16] considered the scalability with the number of GPUs on a single node, and developed a GPU performance model for stream workload partitioning in multi-GPU platforms with high scalability. Chen et al. [CXT+15] proposed G-Storm, which enables Storm [TTS+14] to utilize GPUs and can be applied to various applications that Storm has already supported.

### 3.1.3 FPGA-Enabled Stream Processing

FPGAs are programmable integrated circuits whose hardware interconnections can be configured by users. Due to their low latency, high energy efficiency, and low hardware engineering cost, FPGAs have been explored in various application scenarios, including
Chapter 3. Literature Review

stream processing.

Hagiescu et al. [HWBR09] first elaborated challenges to implementing stream processing on FPGAs, and proposed algorithms that optimizes processing throughput and latency for FPGAs. Mueller et al. [MTA09] provided Glacier, which is an FPGA-based query engine that can process queries from streaming networks. The operations in Glacier include selection, aggregation, grouping, and windowing. Experiments show that using FPGAs help achieve much better performance than using conventional CPUs. A common limitation of an FPGA-based system is its intensive synthesis process, which takes significant time to compile the application into hardware designs for FPGAs. This makes FPGA-based systems inflexible in adapting to query changes. In response, Najafi et al. [NSJ13] demonstrated Flexible Query Processor (FQP), an online reconfigurable event stream query processor that can accept new queries without disrupting other queries under execution.

3.2 Key Issues of Data Stream Processing

In this section, we discuss a few key issues in data stream processing, and especially highlights prior works related to our proposed techniques.

3.2.1 Execution Optimization Techniques

Various research communities have greatly influenced the development of DSPSs including database, operating systems, and complex event processing. They have independently developed optimizations for stream processing. A detailed survey is provided at [HSS+03]. We now focus on the following two categories that are closely related to our works.

**Tuple Batching.** The general idea of batching in stream processing is straightforward. Instead of transmit and process one data item at one time, process multiple data times in a single batch. The benefits are mainly two folds. First, batching can amortize the cost of bringing the execution instructions into cache over multiple data items. Second, it also amortizes the cost of communication over operators. Sax et al. [SC14] proposed to create an independent batching buffer for each consumer in order to batch all tuples that will
be processed by the same consumer to avoid wrong fields grouping problem. However, the additional explicit buffering delay in every executor may introduce serious negative impact on the system latency. Das et al. [DZSS14] studies the effect of batch sizes and other parameters on the throughput and end-to-end latency of the system, and proposes an algorithm based on Fixed-Point Iteration to automatically adapt batch sizes as the circumstance varies, which targets to minimize the end-to-end latency while keeping the system stable. In our profiling study, we also found batching is particularly helpful in improving DSPS performance by reducing L1-ICache miss stalls.

**Operator Placement and Parallelism.** Operator placement in a distributed DSPS determines its scalability to a greater extent. Many algorithms and mechanisms [ABQ13, XCTS14, PHH15, CGLPN16, CGLPN17, Kha09] are developed to allocate (i.e., schedule) operators of a job into physical resources (e.g., compute node) in order to achieve certain optimization goal, such as maximizing throughput, minimizing latency or minimizing resource consumption, etc. Aniello et al. [ABQ13] 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 [XCTS14] 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 [PHH15] 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. [CGLPN16, CGLPN17] propose a general mathematical formulation of the problem of optimizing operator placement for distributed data stream processing. Recently, Li et al. [LXTW18] present a machine-learning based framework for minimizing end-to-end processing latency on DSPSs. However, based on our study, executor placement inside a single machine also needs to be considered due to the NUMA effect.

Many DSPSs, such as Storm [TTS14], Heron [KBF15], Flink [CKE15] and Seep [CFMKP13], 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
Chapter 3. Literature Review

Ingress 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). A natural question raised in deploying DSPSs on multicore architecture is how to find a streaming execution plan that maximizes processing throughput of DSPSs under NUMA effect. Due to NUMA, operator experiences additional remote memory access (RMA) penalty during input data fetch when it is allocated in different CPU sockets to its producers. As a result, the processing speed and resource demand of an operator is not fixed but related to how it is allocated in different execution plans. An algorithm ignoring such varying properties of the concerned problem 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. Although NUMA-awareness system optimization has been previously studied in the context of relational database [GARH14, PSM+16, LBKN14], those works are either 1) focused on different optimization goals (e.g., better load balancing [PSM+16] and minimizing resource usage [GARH14]) or 2) based on different system architectures [LBKN14]. They provide highly valuable techniques, mechanisms and execution models but none of them uses the knowledge at hand to solve the problem we address, that is how to find a streaming execution plan that maximizes processing throughput of DSPSs under NUMA effect.

3.2.2 Out-of-order handling

In a real production environment, out-of-order\(^1\) input data is not uncommon. A stream operator is considered order-sensitive if it requires input events to be processed in a certain predefined order (e.g., chronological order). Handling out-of-order input data in order-sensitive operator often turns out to be a performance bottleneck as there is a fundamental conflict between data parallelism and order-sensitive – the former seeks to improve the throughput of an operator by letting more than one thread operate on different events concurrently, possibly out-of-order.

\(^1\)other issues such as delay and missing can be seen as special cases of out-of-order.
Gulisano et al. [GNPT15] are among the first to handle out-of-order for high-performance stream join on multicore. The proposed algorithm called scalejoin first merges all incoming tuples into one stream (through a data structure called scallegate) and then distributes them to processing threads (PTs) to perform join. The output also needs to be merged and sorted before exiting the system. StreamBox [MPJ17] handles out-of-order with punctuation based technique on multicore processors. Relying on a novel data structure called cascading container to track dependencies between epochs (a group of tuples delineated by punctuations), StreamBox is able to maintain the processing order among multiple concurrently executing containers that exploit the parallelism of modern multicore hardware. Kuralenok et al. [KMTN18] attempt to balance the conflict between order-sensitive and multicore parallelism with an optimistic approach falling in the third approach. The basic idea is to conduct process without any regulations, but apologize (i.e., sending amending signals) when processing order is violated. They show that the performance of the proposed approach depends on how often reorderings are observed during runtime. In the case when the input order naturally preserved, there is almost no overhead. However, it leads to extra network traffic and computations when reorderings are frequent. To apply such an approach in practical use-cases, it is hence necessary to predict the probability of reordering, which could be interesting future work. Despite the significant efforts, existing DSPSs are still far from ideal in exploiting the potential of modern hardware. For example, as observed in our experiments [ZHZZ19], the same DSPS (i.e., StreamBox) delivers several times lower performance as a result of enabling ordering-guarantee. In this thesis, we hence assume applications are ordering-insensitive and we focus on mitigate the gaps between stream processing and modern hardwares while allowing out-of-order processing.

3.2.3 State Management in DSPSs

Emerging stream applications often require the underlying DSPS to maintain large application state so as to support complex real-time analytics [TSM18] – a feature often being called as state management. Representative example states kept during stream processing include graph data structure [ZCC17] and transaction records [MTZ15].

Researchers are exploring efficient approaches to support analytical query workload
An NVM-aware storage layout for tables is presented based on a multi-dimensional clustering approach and a block-like structure to utilize the entire memory stack. The storage structure designed on NVM may serve as the foundation in supporting features like transactional stream processing system [Sat19]. Recent works [LES+18] have also investigated the performance limitations of current state management approaches on SSDs and show that query-aware optimizations can significantly improve the performance of stateful query processing on SSDs.

Concurrency Control. An operator may need to manage shared mutable states [AMC17], where the same application state may be concurrently accessed (read/write) by multiple instances of the operator (called executors) running in different threads. In order to maintain the consistency of shared states and to guarantee the correctness of stream computation results, a DSPS that attempts to concurrently manipulate the shared states must perform access operations with ACID guarantees. Concurrency control (CC) protocols have been investigated widely in decades [BG81, FTA14, YBP+14, WCT16] for guaranteeing ACID properties. To improve the transaction processing performance, several works have been proposed to improve the effectiveness of concurrency control protocols. Transaction chopping [SLSV95], a widely adopted technique to partition transactions, have been applied to several modern database applications. Faleiro et al. [FTA14] proposed a technique for lazily evaluating transactions, and this technique improves database performance for certain kinds of workloads. Based on a similar design principle, the same authors improved the MVCC performance by decoupling concurrency control protocol and version management from transaction execution [FA15]. Wang et al. [WMC+16] presented a method for decomposing transaction and evaluate them in a fine-grained manner while still guaranteeing state consistency. Different from the traditional database architectures, several deterministic DBMSs, including H-Store [SMA+], Hyper [KN11], and Calvin [TA10], have been proposed. These DBMSs divide the underlying storage into multiple partitions, each of which is protected by a lock and is assigned a single-threaded execution engine with exclusive access.

Transactional State Management. More than simply guaranteeing the ACID properties, DSPSs further need to guarantee the state access order follows the input event timestamp order [CDK+14] (e.g., a write request shall not affect read request of earlier event with a
smaller timestamp). Wang et al. [WRE11] conducted an early study on the importance of supporting transactional state management. In their proposal, DSPSs must support a new computing paradigm, called active complex event processing (ACEP), to enable complex interactive real-time analytics. Subsequently, different rules are maintained and shared by multiple concurrently running stream operators, which has to be guaranteed with transactional semantics. To this end, two types of locking based algorithms for concurrent state transaction execution are described. Botan et al. [BFKT12] presented an unified transactional model for streaming applications, called UTM. MeeHan et al. [MTZ+15] attempted to fuse OLTP and streaming processing together and developed the S-Store system. Affetti et al. [AMC17] proposed a state consistency model with ACID+O properties guaranteed. However, prior works commonly heavily rely on locks in guaranteeing state consistency during state transaction execution. There is also a recent launch of a commercial system, called Streaming Ledger [Tra18] for extending Flink with transactional state management capability. However, it is unfortunately close-sourced and little information has been exposed to the public.

### 3.3 Performance Evaluation for DSPSs

Despite the successes achieved during the last decade, DSPSs are now facing great challenges when supporting a wide range of emerging time-critical applications, which generally require the underlying DSPSs to achieve low end-to-end latency when processing huge volumes of data. Witnessing the emergence of modern commodity machines with massively parallel processors, researchers and practitioners find shared-memory multicore architectures an attractive alternative platform [CDW17], and several in-memory single-node DSPSs are recently proposed [KWCF+16, MPJ+17]. Unfortunately, there is no one “standard” implementation of DSPSs, and there have been a few studies on comparing different DSPSs. A comparison of S4 and Storm [MBM09] uses a micro-benchmark to understand the performance issues of the systems regarding scalability, execution time and fault tolerance. A similar study [CCM12] has been conducted to compare the performance characteristics of three DSPSs, including System-S, S4, and Esper. A recent study [SC] comparing Flink, Storm, and Spark Streaming has shown that, Storm and Flink have sub-second latency with relatively low throughputs,
while Spark streaming has higher throughput at a relatively high latency. However, those evaluation study treat each DSPS as a black box, and little attention has been paid to the research on the key and common design aspects of various DSPSs on modern multi-core processors.
4.1 Introduction

Many data stream processing systems (DSPs) have recently been proposed to meet the increasing demand of processing streaming data, such as Apache Storm [TTS+14], Flink [CKE+15], Spark Streaming [DZSS14], Samza [NPP+17] and S4 [NRNK10]. Regardless of the different architectures of those DSP systems, they are mainly designed and optimized for scaling out using a cluster of commodity machines (e.g., [XCTS14, ABQ13, PHH+15]). We observe the following three common design aspects in building those existing DSPs:

a) **Pipelined processing with message passing:** A streaming application is usually implemented as multiple operators with data dependencies, and each operator performs three basic tasks continuously, i.e., receive, process and output. Such a pipelined processing design enables DSPs to support very low latency processing, which is one of the key requirements in many real applications that cannot be well supported in batch-processing systems.

b) **On-demand data parallelism:** Fine-grained data parallelism configuration is supported in many DSP systems. Specifically, users can configure the number of threads in each operator (or function) independently in the streaming application. Such an on-demand data parallelism design aims at helping DSP systems scale for high throughput.

c) **JVM-based implementation:** Recent DSP systems are mostly built on top of JVM (Java
Virtual Machine). Although the use of JVM-based programming language makes the system development more productive (e.g., built-in memory management), many JVM runtime performance issues such as data reference and garbage collection are transparent to programmers.

Modern servers are being deployed in the cluster environment. More CPU cores are being put on the same die. Subsequently, the on-chip cache hierarchies that support these cores are getting larger, deeper, and more complex. Furthermore, as modern machines scale to multiple sockets, non-uniform memory access (NUMA) becomes an important performance factor for data management systems (e.g., [LBKN14, LPM+13]). For example, recent NUMA systems have already supported hundreds of CPU cores and multi-terabytes of memory [MF17]. However, there is a lack of detailed studies on profiling the above common design aspects of DSP systems on modern architectures.

In this work, we experimentally revisit those common design aspects on a modern machine with multiple CPU sockets. We aim to offer a better understanding of how current design aspects of modern DSP systems interact with modern processors when running different types of applications. We use two DSP systems (i.e., Apache Storm [TTS+14] and Flink [CKE+15]) as the evaluation targets. Note that the major goal of this study is to evaluate the common design aspects of DSP systems on scale-up architectures using profiled results so that our results and findings can be applicable to many other DSP systems, rather than to compare the absolute performance of individual systems. There has been no standard benchmark for DSP systems, especially on scale-up architectures. Thus, we design our micro benchmark with seven streaming applications according to the four criteria proposed by Jim Gray [Gra92].

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

First, the design of supporting both pipelined and data parallel processing leads to a very complex massively parallel execution model in DSP systems, which poorly utilizes modern multi-core processors. Based on our profiling results, a significant portion (∼40%) of the total execution time is wasted due to L1-instruction cache (L1-ICache) misses. The significant L1-ICache misses are mainly due to the large instruction footprint between two consecutive invocations of the same function.
Second, the design of continuous message passing between operators causes a serious performance degradation to DSP systems running on multiple CPU sockets. Furthermore, the current design of data parallelism in DSP systems tends to equally partition input streams regardless of the location of executors (i.e., they may be scheduled on different CPU sockets), which overlooks the NUMA effect. The throughput of both Storm and Flink on four CPU sockets is only slightly higher or even lower than that on a single socket for all applications in our benchmark. The costly memory accesses across sockets severely limit the scalability of DSP systems.

Third, the JVM runtime brings two folds of overhead to the execution, and they are moderate in DSP systems. 1) The translation lookaside buffer (TLB) stalls take 5~10% and 3~8% of the total execution time for most applications on Storm and Flink, respectively. The major causes include the frequent pointer referencing issues in data accesses and Java execution. 2) The overhead from garbage collection (GC) accounts for only 1 ~ 3% of the total execution time. The observed minor impact of GC is very different from previous studies on other data-intensive platforms with large memory footprints (e.g., [ABVA15, SP14]).

Addressing the above-mentioned issues should allow DSPSs to exploit modern scale-up architectures. As initial attempts, we evaluate two optimizations: 1) non-blocking tuple batching to reduce the instruction footprint for processing a tuple so that the instruction cache performance can be improved; 2) NUMA-aware executor placement to make thread placement aware of remote memory accesses across sockets. The evaluation results show that both optimizations are effective in improving the performance of DSPSs on multi-socket multi-core processors. Putting them altogether achieves 1.3~3.2x and 1.2~3.1x throughput improvement on Storm and Flink, respectively.

To the best of our knowledge, this is the first detailed study of common design aspects of DSPSs on scale-up architectures with a wide range of applications. Improving DSPSs on the scale-up architectures is also beneficial for the scale-out setting, by either offering a better performance with the same number of machines or reducing the number of machines to achieve the same performance requirement.
Chapter 4. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors

4.2 Methodology

We conduct an extensive set of experiments to profile the performance of Storm and Flink on a modern scale-up server using different applications. In this section, we first present the evaluation goals of this study. Next, we introduce our profiling tools, followed by our benchmark.

4.2.1 Evaluation Goals

This study has the following design goals.

First, we aim to identify the common designs of modern DSP systems, and to understand how those designs (i.e., pipelined processing with message passing, on-demand data parallelism, and JVM-based implementation) interact with modern processors when running different types of applications. Second, with the detailed profiling study, we hope to identify some hardware and software approaches to resolving the bottleneck and point out the directions for the design and implementation of future DSP systems.

4.2.2 Profiling Tools

**JVM profile.** Table 4.1 lists the JVM flags that we use to monitor the performance of JVM. We are mainly interested in two kinds of activities, including those in just-in-time (JIT) compilation and GC. We only enable those trace logs when we need to analyze the corresponding activities. Otherwise, the trace logs are disabled. We use Performance Inspector [CC] for gathering detailed instruction-tracing information. We measure the size of the objects created at runtime using the MemoryUtil tool from the Classmexer library [Cof08].

**Processor profile.** We systematically categorize where the processor time is spent for executing Storm and Flink to identify common bottlenecks of their system designs when running on multi-socket multi-core processors. We use Intel Vtune [Vtu18] for profiling at the processor level.

Similar to the recent study [SP14], we break down the total execution time to the following components: 1) computation time, which is contributed by the issued μops
Table 4.1: JVM profile flags

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JIT Logging</strong></td>
<td>Trace just-in-time compilation activities</td>
</tr>
<tr>
<td>UnlockDiagnosticVMOptions</td>
<td>Enable processing of flags relating to field diagnostics</td>
</tr>
<tr>
<td>TraceClassLoading</td>
<td>Trace all classes loaded</td>
</tr>
<tr>
<td>LogCompilation</td>
<td>Enable log compilation activity</td>
</tr>
<tr>
<td>PrintAssembly</td>
<td>Print assembly code</td>
</tr>
<tr>
<td><strong>GC Logging</strong></td>
<td>Trace garbage collection activities</td>
</tr>
<tr>
<td>PrintGCTimeStamps</td>
<td>Print timestamps of garbage collection</td>
</tr>
<tr>
<td>PrintGCDetails</td>
<td>Print more details of GC including size of collected objects, time of objects promotion</td>
</tr>
</tbody>
</table>

that subsequently be executed and retired; 2) branch misprediction stall time ($T_{Br}$), which is mainly due to the executed µops that will however never be retired; 3) front-end stall time ($T_{Fe}$), which is due to the µops that were not issued because of the stalls in any components in the front-end; 4) back-end stall time ($T_{Be}$), which is due to the µops that were available in the IDQ but were not issued because of resources being held-up in the back-end.

Table 4.2 shows the measurement components for individual stalls. We have conducted an extensive measurement on stalls from front-end, back-end, and branch misprediction.

All our experiments are carried out on a four-sockets server with the Intel Xeon Sandy Bridge EP-8 processors. Table 4.3 shows the detailed specification of our server and relevant settings in Storm and Flink.

4.3 Performance Evaluation

In this section, we present the performance evaluation results of different applications on Storm and Flink on multi-core processors. We tune each application on both Storm and Flink according to their specifications such as the number of threads in each operator.

**Throughput and resource utilization on a single socket.** Figure 4.1a shows the
Table 4.2: Processor measurement components

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_C$</td>
<td>Effective computation time</td>
</tr>
<tr>
<td>$T_{B_p}$</td>
<td>Branch misprediction stall time</td>
</tr>
<tr>
<td>$T_{F_e}$</td>
<td>Front-end stall time</td>
</tr>
<tr>
<td>ITLB stalls</td>
<td>Stall time due to ITLB misses that causes STLB hit or further cause page walk</td>
</tr>
<tr>
<td>L1-I cache stalls</td>
<td>Stall time due to L1 instruction cache misses</td>
</tr>
<tr>
<td>ILD stalls</td>
<td>Instruction Length Decoder stalls</td>
</tr>
<tr>
<td>IDQ stalls</td>
<td>Instruction Decoder Unit stalls</td>
</tr>
<tr>
<td>$T_{B_e}$</td>
<td>Back-end stall time</td>
</tr>
<tr>
<td>DTLB stalls</td>
<td>Stall time due to DTLB misses, which causes STLB hit or further cause page walk</td>
</tr>
<tr>
<td>L1-D Stalls</td>
<td>Stall time due to L1 data cache misses that hit L2-Cache</td>
</tr>
<tr>
<td>L2-Cache Stalls</td>
<td>Stall time due to L2-Cache misses that hit in LLC</td>
</tr>
<tr>
<td>LLC stalls (local)</td>
<td>Stall time due to LLC misses that hit in local memory</td>
</tr>
<tr>
<td>LLC stalls (remote)</td>
<td>Stall time due to LLC misses that hit in memory of other socket</td>
</tr>
</tbody>
</table>

Table 4.3: Detailed specification on our testing environment

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Xeon E5-4640, Sandy Bridge EP</td>
</tr>
<tr>
<td>Cores (per socket)</td>
<td>8 * 2.4GHz (hyper-threading disabled)</td>
</tr>
<tr>
<td>Sockets</td>
<td>4</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32KB Instruction, 32KB Data per core</td>
</tr>
<tr>
<td>L2-Cache</td>
<td>256KB per core</td>
</tr>
<tr>
<td>Last level cache</td>
<td>20MB per socket</td>
</tr>
<tr>
<td>Memory</td>
<td>4 * 128GB, Quard DDR3 channels, 800 MHz</td>
</tr>
<tr>
<td>Apache Flink</td>
<td>version 1.0.2 (checkpoint enabled)</td>
</tr>
<tr>
<td>Apache Storm</td>
<td>version 1.0.0 (acknowledge enabled)</td>
</tr>
<tr>
<td>Java HotSpot VM</td>
<td>java 1.8.0_77, 64-Bit Server VM, (mixed mode) -server -XX:+UseG1GC -XX:+UseNUMA</td>
</tr>
</tbody>
</table>
Chapter 4. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors

Table 4.4: CPU and memory bandwidth utilization on a single CPU socket

<table>
<thead>
<tr>
<th></th>
<th>WC</th>
<th>FD</th>
<th>LG</th>
<th>SD</th>
<th>VS</th>
<th>TM</th>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storm</td>
<td>CPU Utilization</td>
<td>62%</td>
<td>39%</td>
<td>61%</td>
<td>28%</td>
<td>75%</td>
<td>98%</td>
</tr>
<tr>
<td></td>
<td>Memory Utilization</td>
<td>20%</td>
<td>16%</td>
<td>10%</td>
<td>7%</td>
<td>19%</td>
<td>60%</td>
</tr>
<tr>
<td>Flink</td>
<td>CPU Utilization</td>
<td>75%</td>
<td>27%</td>
<td>31%</td>
<td>13%</td>
<td>92%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>Memory Utilization</td>
<td>53%</td>
<td>16%</td>
<td>18%</td>
<td>6%</td>
<td>17%</td>
<td>52%</td>
</tr>
</tbody>
</table>

throughput and Table 4.4 illustrates the CPU and memory bandwidth utilisations of running different applications on Storm and Flink on a single CPU socket. We measure the overall resource utilization during stable execution by avoiding the beginning and ending phases of each application. We have two observations. First, the comparison between Storm and Flink is inconclusive. Flink has higher throughput than Storm on WC, FD, and SD, while Storm outperforms Flink on VS and LR. The two systems have similar throughput on TM and LG. Second, our benchmark covers different runtime characteristics. Specifically, VS and TM have high CPU utilization. CPU utilization of LG and LR is median, and that of WC and SD is low.

It is noteworthy that the major goal of this study is to identify the issues in common designs of DSP systems on scale-up architectures, rather than to compare the absolute performance of different DSP systems. We present the normalized performance results in the rest of this work.

Scalability on varying number of CPU cores. We vary the number of CPU cores from 1 to 8 on the same CPU socket and then vary the number of sockets from 2 to 4 (the number of CPU cores from 16 to 32). Figures 4.1b and 4.1c show the normalized throughput of running different applications with varying number of cores/sockets on Storm and Flink, respectively. The performance results are normalized to their throughputs on a single core.

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

### 4.4 Study the impact of common designs

In the following section, we investigate the underlying reasons for the performance degradation and how the three design aspects (i.e., pipelined processing with message passing, on-demand data parallelism, and JVM-based implementation) interact with multi-socket multi-core processors. Specifically, we first show an execution time breakdown on running different applications on Storm and Flink on a single socket. Then, we study the impact of massively parallel execution model, message passing and stream partitioning and JVM runtime environment.
Chapter 4. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors

4.4.1 Execution time breakdown

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

Figure 4.2 shows the execution time breakdown of Storm and Flink running on a single socket on different processor components as introduced in Section 2.1. We find that 59\textendash}77\% and 58\textendash}69\% of the overall execution time are spent on stalls (Branch misprediction stalls, Front-end stalls, Back-end stalls) for all applications running on Storm and Flink, respectively.

Front-end stalls account for 35\textendash}55\% and 25\textendash}56\% of the total execution time of Storm and Flink, respectively. This result is significantly different from the batch processing framework (e.g., [ABVA15]). Back-end stalls account for approximately 13\textendash}40\% and 7\textendash}40\% of the total execution time of Storm and Flink, respectively. Branch misprediction stalls are low, ranging from 3 \textendash} 4\% for all applications.

In the following, we examine the processor stalls in more details with respect to the three designs of DSP systems (i.e., pipelined processing with message passing, on-demand data parallelism, and JVM-based implementation).
4.4.2 Massively parallel execution model

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

Figure 4.3 illustrates the breakdown of the front-end stalls in running Storm and Flink on a single socket. Each of the L1 instruction cache (L1-ICache) miss and instruction decoding (I-Decoding) stalls contributes nearly a half of the front-end stalls.

**L1-ICache miss stalls:** Our investigation reveals that there are two primary sources responsible for the high L1-ICache miss. *First,* due to the lack of a proper thread scheduling mechanism, the massive threading execution runtime of both Storm and Flink produces frequent thread context switching. *Second,* each thread has a large instruction footprint. By logging JIT compilation activities, we found that the average size of the native machine code generated per executor thread goes up to 20KB. As the size of current L1-ICache (32KB per core) is still fairly limited, it cannot hold those instructions at runtime, which eventually leads to L1-ICache thrashing.

We now study the details of the instruction footprints between two consecutive invocations of the same function. In order to isolate the impact of user-defined functions, we test a “null” application, which performs nothing in both Storm and Flink (labeled as “null”). Figure 4.4 illustrates the cumulative density function (CDF) of instruction footprints on a log scale, which stands for the percentage that instruction footprint is no
Chapter 4. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors

Figure 4.4: Instruction footprint between two consecutive invocations of the same function.

larger than a certain number of distinct instructions.

We add three solid vertical arrows to indicate the size of L1-ICache (32KB), L2-Cache (256KB), and LLC (20MB). With the detailed analysis on the instruction footprint, we make three observations. *First*, two turning points on the CDF curves are observed at $x=1$KB and $x=10$MB for Storm and $x=1$KB and $x=1$MB for Flink, which reflects the common range of their instruction footprints during execution. *Second*, the cross-over points of L1-ICache line and different CDF curves are between $0.5 \sim 0.7$ for Storm and $0.6 \sim 0.8$ for Flink. It means, around $30 \sim 50\%$ and $20 \sim 40\%$ of the instruction footprints between two consecutive calls to the same functions are larger than the L1-ICache in Storm and Flink, respectively. This causes severe L1-ICache stalls. It also shows that Flink has a better instruction locality than Storm on L1-ICache. *Third*, Storm has similar tracing on instruction footprint with or without running user applications. This indicates that many of the instruction cache misses may come from Storm platform itself. This also explains the reason that different applications show similar L1-ICache miss in Storm. In contrast, the platform of Flink has a smaller instruction footprint.
Chapter 4. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors

I-Decoding stalls: The high instruction decoding (I-Decoding) stalls are related to the high L1-ICache miss issue. The I-Decoding stalls can be further broken down into instruction length decoder (ILD) stalls and instruction decoding queue (IDQ) stalls.

The ILD stalls further consist of instruction queue (IQ) full stalls and length change prefix (LCP) stalls. IQ is used to store the pre-fetched instructions in a separate buffer while the processor is executing the current instruction. Due to the large footprint between two consecutive invocation of the same function, IQ full stalls are frequent and contribute nearly 20% of front-end component stall time for all applications on both Storm and Flink. On the other hand, the LCP stalls are negligible, with less than 0.05% for all applications according to our measurement.

Another important aspect of I-Decoding stalls is the IDQ stalls, which consist mainly of decoded instruction cache (D-ICache) stalls. D-ICache enables skipping the fetch and decode stages if the same \( \mu \)ops are referenced later. However, two aspects of D-ICache may offset its benefits, or even degrade the performance. First, when L1-ICache miss occurs, the D-ICache also needs to be invalidated, which subsequently causes a switch penalty (i.e., the back-end has to re-fetch instructions from the legacy decoder pipeline). Second, if a hot region of code is too large to fit in the D-ICache (up to 1.5k \( \mu \)ops), the front-end incurs a penalty when \( \mu \)op issues switch from the D-ICache to the legacy decoder pipeline. As we have shown earlier that L1-ICache misses are high during Storm and Flink execution, this issue propagates to a later stage, which causes frequent misses in the D-ICache and eventually causes high IDQ stalls.

4.4.3 Message passing and stream partitioning

Finding (3): The design of message passing between operators causes a severe performance degradation to the DSPSs running on multiple CPU sockets. During execution, operators may be scheduled into different sockets and experience frequent costly remote memory accesses during the fetching of input data. Furthermore, the current design of data parallelism has overlooked the NUMA effect.

Recently, the NUMA-aware allocator has already been implemented in the Java HotSpot Virtual Machine to take advantage of such infrastructures, which provides automatic memory placement optimizations for Java applications. We enable this optimization
Chapter 4. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors

Table 4.5: LLC miss stalls when running Storm with four CPU sockets

<table>
<thead>
<tr>
<th></th>
<th>WC</th>
<th>FD</th>
<th>LG</th>
<th>SD</th>
<th>VS</th>
<th>TM</th>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC Miss (local)</td>
<td>0%</td>
<td>5%</td>
<td>3%</td>
<td>4%</td>
<td>4%</td>
<td>1%</td>
<td>7%</td>
</tr>
<tr>
<td>LLC miss (remote)</td>
<td>6%</td>
<td>16%</td>
<td>17%</td>
<td>13%</td>
<td>17%</td>
<td>24%</td>
<td>22%</td>
</tr>
</tbody>
</table>

in our JVM by specifying the *useNUMA* flag. However, our experiments have already shown that this flag is insufficient for reducing the NUMA impact and we have observed poor scalability in both Storm and Flink on multiple sockets. The main problem is the high remote memory access overhead due to the heavy pipelined message passing design. During execution, each executor needs to fetch data from the corresponding producer continuously. On the multi-socket and multi-core architectures, an executor can have three kinds of data accesses. (1) In cache: the input data is accessed in the cache. This comes with minimum access penalty. (2) In local memory: access data with a miss in the cache but a hit in its local memory. This happens when producer and consumer executors are located in the same CPU socket, and it comes with a cost of local memory read. (3) In remote memory: access data with a miss in the cache and a further miss in its local memory. This comes with very high access penalty, and it happens when producer and consumer executors are located in different CPU sockets.

As a result, the data access cost is depending on the location of the producer and consumer, which creates significant performance divergence among parallel executors of even the same operator. However, neither Storm nor Flink is aware of such performance heterogeneity issues and continuously distributes equal amounts (in the case of shuffle grouping) of tuples among executors. Table 4.5 shows the LLC miss stalls for executing on Storm with four CPU sockets. We have similar observations when running the applications on Flink with four CPU sockets enabled.

We take TM on Storm as an example to further study the impact of stream partitioning. We start with the tuned number of threads (i.e., 32) of Map-Matcher operator of TM on four sockets, and further increase the number of threads up to 56. Figure 4.5a shows a significant increase in the standard derivation of executors’ latencies with the increasing of the number of executors when running Storm on four CPU sockets. Those executors experience up to 3 times difference in the average execution latency in the case of 56
Chapter 4. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors

4.4.4 JVM Runtime Environment

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

Both Storm and Flink are implemented using JVM-based programming language. The efficiency of JVM runtime is crucial to the performance of Storm and Flink. As we have mentioned before, the back-end of the processor is where the actual execution happens. Figure 4.6 breaks down the back-end stalls into L1-DCache stalls, L2-Cache stalls, LLC stalls, and DTLB stalls when running Storm and Flink on a single socket.

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

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

**Garbage collection overhead:** We use G1GC [DFHP04] as the garbage collector in our JVM. The garbage collection (GC) is infrequent in running all applications on both Storm and Flink, and the same observation is made even if we run the benchmark for hours. Based on GC logs, we find that no major GC occurs during the execution and minor GC contributes only $1 \sim 3\%$ to the total execution time across all the applications for both Storm and Flink. As a sanity check, we also study the impact of using an older version of GC named parallelGC. When the parallelGC mechanism is used, the overhead of GC increases to around $10 \sim 15\%$, which indicates the effectiveness of G1GC. Nevertheless, we plan to evaluate the impact of GC with more extensive applications.
4.5 Towards more efficient DSP systems

In this section, we present our initial attempt to address the performance issues found in the previous section. We present two optimization techniques, including non-blocking tuple batching (to reduce instruction cache stalls) and NUMA-aware executor placement (to reduce remote memory accesses). We evaluate the effectiveness of the techniques by first studying their individual impacts and then combining both techniques together. We note that, the two optimization techniques address the efficiency issues of the two common designs (i.e., pipelined processing with message passing and on-demand data parallelism), and we conjecture that the optimizations can be applied to other DSP systems with the same designs.

4.5.1 Non-blocking Tuple Batching

Our profiling results suggest that the large instruction footprints between two consecutive invocations of the same function cause severely performance issues including L1-ICache miss and I-Decoding stalls, which lead to high front-end stalls. One of the solutions is batching multiple tuples together before passing to the consumer executor for processing. In this way, each function invocation can process multiple tuples so that the instruction footprint between two consecutive invocations of the same function is reduced. Similar ideas of tuple batching are already proposed [SC14] or in use in some DSP systems [CKE+15]. However, those techniques rely on a buffering stage, which introduces wait delay in execution. For example, Sax et al. [SC14] proposed to create an independent batching buffer for each consumer in order to batch all tuples that will be processed by the same consumer. Tuples are not emitted until the corresponding buffer becomes full. However, the additional explicit buffering delay in every executor may introduce serious negative impact on the system latency. In order to preserve low latency processing feature of DSP system, we develop a simple yet effective non-blocking tuple batching strategy to address this issue.

The basic idea of our solution is as follows. Consider an executor processes a batch of tuples and outputs multiple tuples, we try to put its output tuples together as a batch, or multiple batches each containing tuples belonging to the same key. Once the
corresponding consumer receives such a batch, it can then process multiple tuples from
the batch with a single function invocation and further batching its output tuples in
a similar manner. Our solution requires that the data producer to prepare the initial
batches of tuples, where the size of batch $S$ is a parameter that we will tune in later
experiments. When the Data Producer of an application generates multiple tuples (more
than $S$) each time, it simply groups them into batches with size up to $S$ and feeds to the
topology. Otherwise, we can let the data producer accumulate $S$ tuples before feeding
to the topology. As Data Producer is usually relatively light-weight compared to other
operators in an application, this kind of batching has little overhead.

It is rather straightforward to implement the non-blocking tuple batching algorithm for
any grouping policy (Section 2.2) except the key-grouping policy (i.e., fields grouping),
as we can simply group together all the output tuples of an executor. However, if an
executor uses fields grouping, simply putting output tuples into one batch may cause
errors [SC14] due to wrongly sending output tuples targeting at different consumers
based key in each tuple. Existing batching techniques rely on a buffering stage in order
to resolve such issue [SC14]. In contrast, we develop an algorithm for non-blocking
tuple batching of fields grouping, as illustrated in Algorithm 4.1.

The basic idea is to store multiple output tuples into a multi-valued hash map (at lines
10-12), and the fields (i.e., keys) used in choosing consumer are re-computed based on
the fields originally declared (at lines 10-11). At line 4, the HashMultimap is the multi-
value hash map used to batch multiple values with the same key (implemented based
on org.apache.storm.guava.collect.HashMultimap). At line 10, we use a concatenate function
to combine the original multiple fields. In this way, we guarantee the correctness by
always generating the same new key from the same original fields while batching as
many tuples as possible (i.e., it may generate the same new key for tuples with different
original fields which are can be safely batched together).

We now study the impact of tuple batching optimization by varying $S$. Figure
4.7 illustrates the normalized throughput of Storm and Flink with tuple batching
optimization for different applications on a single CPU socket. Results are normalized to
the original non-batch setting of Storm and Flink (denoted as non-batch). As expected,
tuple batching can significantly reduce instruction cache misses and hence improve the
Algorithm 4.1: Non-blocking tuple batching for fields grouping

Data: batch \( B \)
Data: \( t_o \): temporary output tuple
Data: \( t_o.attributeList \): fields grouping attributes
Data: \( n \): the number of executors of the consumer

1. Initialize \( cache \) as an empty HashMultimap
2. Initialize \( newkey \) as an empty object
3. for each tuple \( t_i \) of \( B \) do
   
   \[ t_o \leftarrow \text{function_process}(t_i); \] // Perform custom function of the operator.
   
   \[ temp \leftarrow \text{Combine}(t_o.attributeList); \]
   
   \[ newkey \leftarrow \text{hash value of } temp \mod n; \]
   
   Store the \( <newkey, t_o> \) pair in \( cache \), where the values of the same key are maintained in a list \( L \)
4. foreach each key \( K_i \) of the key sets of \( cache \) do
5.   Get the \( <K_i, L> \) pair from \( cache \); // Emit multiple tuples of the same key as a batch.
6.   \[ \text{emit}(<K_i, L>); \]

performance of most applications.

With tuple batching, the processing latency of each tuple may be increased as they are not emitted until all tuples in the same batch are processed. Figure 4.8 shows the normalized average latency per tuple under different batch sizes. Comparing Figures 4.7 and 4.8, we observe a clear trade-off between the throughput and latency. Meanwhile, our non-blocking tuple batching scheme preserves a sublinear increase in process latency for most applications, which is due to the much-improved performance and no explicit buffering delay.

4.5.2 NUMA-Aware Executor Placement

In order to reduce the remote memory accesses among sockets, the executors in a topology should be placed in an NUMA-aware manner. To this end, we develop a
Figure 4.7: Normalized throughput of tuple batching optimization.

Figure 4.8: Normalized latency of tuple batching optimization.

simple yet effective NUMA-aware executor placement approach.

Definition 3. Executor placement. Given a topology execution graph $T$ and the set of executors $W$ in $T$, an executor placement $P(T, k)$ represents a plan of placing $W$ onto $k$ CPU sockets. $k$ can be any integer smaller than or equal to the total number of sockets in a NUMA machine.

Definition 4. We denote the remote memory access penalty per unit as $R$, and the total size of tuples transmitted between any two executors $w$ and $w'$ ($w, w' \in W$) as $\text{Trans}(w, w')$. Each placement $P(T, k)$ has an associated cross-socket communication cost, denoted by $\text{Cost}(P(T, k))$ as shown in Equation 4.1. We denote the set of executors placed onto socket $x$ as $\xi_x$, where $x = 1, \ldots, k$. 
Cost\( (P(T, k)) = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{w \in \xi_i, w' \in \xi_j} R \ast Trans(w, w') \) \hspace{1cm} (4.1)

**Definition 5.** The **optimal executor placement**, denoted by \( P_{opt} \), is defined as \( \text{Cost}(P_{opt}(T, k)) \leq \text{Cost}(P(T, k)), \forall P \in Q \), where \( Q \) is the set of all feasible executor placement solutions.

Our optimization problem is to find \( P_{opt}(T, k) \) for a given topology \( T \) and a number of enabled sockets \( k \). In our experiment, we consider \( k \) from one to four on the four-socket server. We now illustrate that this problem can be mapped into the minimum k-cut problem [GH].

**Definition 6.** The **minimum k-cut** on weighted graph \( G = (V, E) \) produces a vertex placement plan \( (C_{opt}) \) such that \( V \) is partitioned into \( k \) non-empty disjoint sets, and the total weight of edges across disjoint sets is minimized.

Given a topology execution graph \( T \), we can map it to a directed weighted graph \( G \). A mapping from \( T \) to \( G \) is defined as follows: (I) \( \forall \) executor \( w \in W \), there is a one-to-one mapping from \( w \) to a vertex in \( G \). (II) For any producer-consumer \( (< w, w' >, w, w' \in W) \) message passing relationships in \( T \), there is a one-to-one mapping to one edge in \( G \). The communication cost \( (R \ast Trans(w, w')) \) is assigned as the edge weight. The cross-socket communication cost corresponds to the total weight of all edges crossing the disjoint sets. Thus, optimizing \( C_{opt} \) is equivalent to optimizing \( P_{opt} \).

We use the state-of-the-art polynomial algorithm [GH] for solving this problem by fixing \( k \) from one to the number of sockets in the machine. Then, from the results optimized for different \( k \) values, we test and select the plan with the best performance.

Figure 4.9 shows the effectiveness of the NUMA-aware executor placement. Results are normalized to four sockets without optimization. The placement strategy improves the throughput of all applications by \( 7\sim 32\% \) and \( 7\sim 31\% \) for Storm and Flink, respectively.
4.5.3 Put It All Together

Finally, we put both optimizations, namely non-blocking tuple batching ($S = 8$) and NUMA-aware executor placement together. Figure 4.10 illustrates the optimization effectiveness on a single socket and four sockets. Results are normalized to four sockets without optimization. With four sockets, our optimizations can achieve $1.3 \sim 3.2x$ and $1.2 \sim 3.1x$ improvement on the performance for Storm and Flink, respectively. Although our initial attempts have significantly improved the performance, there is still a large room to linear scale-up.
CHAPTER 5

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

5.1 Introduction

Modern multicore processors have demonstrated superior performance for real-world applications [AGN+13] 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 [MF17]. Witnessing the emergence of modern commodity machines with massively parallel processors, researchers and practitioners find shared-memory multicore architectures an attractive streaming platform [ZHD+04, MPJ+17]. Optimizing stream processing in a single node is even mandatory for distributed stream processing – reduce the number of machines required for the same application requirement. However, fully exploiting the computation power delivered by multicore architectures can be challenging. Prior studies [ZHD+04] have shown that existing DSPS underutilize the underlying complex hardware microarchitecture 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 [TTS+14], Heron [KBF+15], Flink [CKE+15] and Seep [CFMKP13], share similar architectures including pipelined processing and operator replication designs. Specifically, an application is expressed as a DAG (directed acyclic graph) where vertexes correspond to continuously running operators, and edges represent data streams flowing between operators. To sustain high input stream ingress 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...
Chapter 5. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures

replication), as well as the way of allocating each operator to the underlying physical resources (i.e., operator placement). In this work, 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 \cite{GARH14, PSM16, LBKN14}. However, those works are either 1) based on cardinality estimation \cite{GARH14}, which is unknown in executing queries over potentially infinite input streams, 2) focused on different optimization goals (e.g., better load balancing \cite{PSM16}) or 3) based on different system architectures \cite{LBKN14}. 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 \textit{varying} processing capability and resource demand of each operator due to \textit{varying} remote memory access penalty under \textit{different} execution plans. Witnessing this problem, we present a novel NUMA-aware streaming execution plan optimization paradigm, called \textit{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 \cite{VN02, GARH14, Kha09}, which assume a predefined and fixed processing capability (or cost) of each operator.

While RLAS provides a more accurate estimation of the application behavior under the NUMA effect, the resulting placement optimization problem 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.

5.2 Execution Plan Optimization

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 5.1(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 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 [ZHD+04]. 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 [TTS+14], Flink [CKE+15], Seep [CFMKP13], and Heron [KBF+15]. Figure 5.1(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).

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.
5.2.1 The Performance Model

Model guided deployment of query plans has been previously studied in relational databases on multi-core architectures, for example [GARH14]. 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 [VN02], 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 5.1. 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.

**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 [VN02].

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 Section 5.6) to utilize
shared-memory environment, the reference passing delay is negligible. Hence, $r_i$ of an operator is simply $r_o$ of the corresponding producer and $r_i$ of spout (i.e., source operator) is given as $I$ (i.e., external input stream ingress rate). Conversely, upon obtaining the reference, 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Notation</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Machine specific.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C$</td>
<td>Maximum attainable unit CPU cycles per socket</td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td>Maximum attainable local DRAM bandwidth</td>
<td></td>
</tr>
<tr>
<td>$Q_{ij}$</td>
<td>Maximum attainable remote channel bandwidth from socket $i$ to socket $j$</td>
<td></td>
</tr>
<tr>
<td>$L_{ij}$</td>
<td>Worst case memory access latency from socket $i$ to socket $j$</td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>Cache line size</td>
<td></td>
</tr>
<tr>
<td><strong>Operator specific.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$M$</td>
<td>Average memory bandwidth consumption per tuple</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>Average time spent on handling each tuple</td>
<td></td>
</tr>
<tr>
<td>$T^f$</td>
<td>Average fetching time per tuple</td>
<td></td>
</tr>
<tr>
<td>$T^e$</td>
<td>Average execution time per tuple</td>
<td></td>
</tr>
<tr>
<td>$N$</td>
<td>Average size per tuple</td>
<td></td>
</tr>
<tr>
<td><strong>Plan inputs</strong></td>
<td>$p$</td>
<td>Input execution plan</td>
</tr>
<tr>
<td></td>
<td>$I$</td>
<td>External input stream ingress rate to source operator</td>
</tr>
<tr>
<td><strong>Model outputs</strong></td>
<td>$r_o$</td>
<td>Output rate of an operator</td>
</tr>
<tr>
<td></td>
<td>$r_o^e$</td>
<td>Expected output rate of an operator</td>
</tr>
<tr>
<td></td>
<td>$r_o(s)$</td>
<td>Output rate of an operator specifically to producer &quot;s&quot;</td>
</tr>
<tr>
<td></td>
<td>$r_i$</td>
<td>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 $I$</td>
</tr>
<tr>
<td></td>
<td>$R$</td>
<td>Application throughput</td>
</tr>
</tbody>
</table>
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 5.1. 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.

\[ r_o = \frac{num}{t_p}, \]

where $num = \sum_{producers} r_i \times t$

\[ t_p = \sum_{producers} r_i \times t \times T(p). \quad (5.1) \]

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 [AcR+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,

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

, where $i$ and $j$ are determined by $p$. \quad (5.2)
by $T^c$ 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. [BXGT04] as illustrated in Formula 5.2. 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 5.2 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. Therefore, $r_o(s)$ is proportional to the proportion of the corresponding input ($r_i(s)$), that is, $r_o(s) = r_o \cdot \frac{r_i(s)}{r_i}$. 

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 $r_o \geq \frac{\sum\text{producers}\cdot r_i}{t_p}$. 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.

---

1It is possible to configure different priorities among different operators here, but is out of the scope of this work.
shortly later in Section 5.3.

**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 [int18]). Similar to the previous work [CAMM13], 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 5.2.2). Task oversubscribing has been studied in some earlier work [IHBZ10], but it is not the focus of this work.

We use the overseer library [Pea11] to measure $T^e$, $M$, and use classmexer library [Cof08] to measure $N$. Figure 5.2 shows the profiling results of $T^e$ of different operators of WC. The major takeaway from Figure 5.2 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 multicore.

5.2.2 Problem Formulation

The goal of our optimization is to maximize the application processing throughput under given input stream ingress 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 5.5.

As the formulas show, we consider three categories of resource constraints that the optimization algorithm needs to make sure the execution plan satisfies. Constraint 5.3 enforces that the aggregated demand of CPU resource requested to anyone CPU socket must be smaller than the available CPU resource. Constraint 5.4 enforces that the aggregated amount of bandwidth requested to a CPU socket must be smaller than the maximum attainable local DRAM bandwidth. Constraint 5.5 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.

\[
\text{maximize } \sum_{\text{sink}} r_{\sigma} \\
\text{s.t., } \forall i, j \in 1, \ldots, n, \\
\sum_{\text{operators at } S_i} r_{\sigma} \cdot T \leq C, \quad (5.3) \\
\sum_{\text{operators at } S_i} r_{\sigma} \cdot M \leq B, \quad (5.4) \\
\sum_{\text{operators at } S_j} \sum_{\text{producers at } S_i} r_{\sigma}(s) \cdot N \leq Q_{i,j}, \quad (5.5)
\]

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 ($n \geq |o|$). Such a large solution space makes brute-force unpractical.

### 5.3 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 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 5.2.1 case 1). Figure 5.3 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
Chapter 5. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures

Figure 5.3: RLAS Optimization example.

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 Section 5.4. In the following, we discuss how the Branch and Bound based technique [MJSS16] 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 $\binom{n}{1} \times \binom{m}{1} = n \times m$ 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 $(n \times m)^n$ candidate solutions [LC99]. 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 5.2) 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.

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 5.4.

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 Figure 5.4 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,
Figure 5.4: 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.

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.
5.4 Algorithm Details

In this section, we first present the detailed algorithm implementations including operator replication optimization (shown in Algorithm 5.1) and operator placement (shown in Algorithm 5.2). After that, we discuss observations made in applying algorithms in optimizing our workload and their runtime (Section 5.4.1). We further elaborate how our optimization paradigm can be extended with other optimization techniques (Section 5.4.2).

Algorithm 5.1 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 5.2 (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., \( \lceil r_i / r_o \rceil \). 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 5.2 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
Chapter 5. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures

Algorithm 5.1: Topologically sorted iterative scaling

Data: Execution Plan: \( p \) // the current visiting plan
Data: List of operators: \( \text{sortedLists} \)
Result: Execution Plan: \( \text{opt} \) // the solution plan

1. \( p.\text{parallelism} \leftarrow \text{set parallelism of all operators to be 1} \)
2. \( p.\text{graph} \leftarrow \text{creates execution graph according to } p.\text{parallelism} \)
3. \( \text{opt}.R \leftarrow 0 \)
4. \( \text{return } \text{Searching}(p); \)

Function \( \text{Searching}(p); \):

5. \( p.\text{placement} \leftarrow \text{placement optimization of } p.\text{graph} \)
6. \( \text{if } p.R > \text{opt}.R \text{ then} \)
7. \( \text{opt} \leftarrow p \) // update the solution plan
8. \( \text{if failed to find valid placement satisfying resource constraint then} \)
9. \( \text{return } \text{opt} \)
10. \( \text{sortedLists} \leftarrow \text{reverse TopologicalSort } (p.\text{graph}) \) // scale start from sink
11. \( \text{foreach } \text{list} \in \text{sortedLists} \text{ do} \)
12. \( \text{foreach } \text{Operator } o \in \text{list} \text{ do} \)
13. \( \text{if } o \text{ is bottleneck then} \)
14. \( p.\text{parallelism} \leftarrow \text{try to increase the replication of } o \text{ by } \left\lceil \frac{r_i}{r_o} \right\rceil \)
15. \( \text{if successfully increased } p.\text{parallelism then} \)
16. \( \text{return } \text{Searching}(p) \) // start another iteration
17. \( \text{else} \)
18. \( \text{return } \text{opt} \)
19. \( \text{return } \text{opt} \)

way of placing the two operators (i.e., up to \( \binom{m}{1} \star \binom{2}{1} \)). 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.

5.4.1 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 5.2) 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 5.1 Line 10), try to further increase the replication level of operator that
Algorithm 5.3: Branching function

1. **Function** `Branching(e);`
2. **Data:** `Node[] children`
3. **foreach** pair of `O_s` and `O_c` in `e.decisions` do
4.   **if** all predecessors of them are already allocated except `O_s` to `O_c` then
5.     `#newAllocate ← 2`
6.     **if** they can be collocated into one socket then
7.       create a Node `n` with a plan collocating them to one socket
8.     else
9.       create a Node `n` with a plan separately allocating them to two sockets
10.      add `n` to `children`
11.   else
12.     `#newAllocate ← 1`
13.     **foreach** valid way of placing `O_s` and `O_c` do
14.       create a new Node and add it to `children`;
15. **foreach** `Node c ∈ children` // update in parallel do
16.   `c.validOperators ← e.validOperators + #newAllocate`
17.   `c.R ← BoundingFunction(c.plan)`
18.   `PushAll(stack, children)`

“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 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.

5.4.2 Extension with other optimization techniques

A number of optimization techniques are available in the literature [HSS+03]. 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 $T_{\text{e\text{\_fused}}}^e$ and $T_{\text{f\_fused}}^f$ to denote the average
execution time and fetch time per tuple of the fused operator, respectively. Then, $T_{\text{f\_fused}}^e$
can be estimated as a summation of $T_{\text{e\_fused}}^e$ of all fused operators. $T_{\text{f\_fused}}^f$ can be estimated as
the $T_{\text{f\_up}}^f$ of the upstream operator ($O_{\text{up}}$) to be fused. That is,

$$
T_{\text{e\_fused}}^e = \sum_{\text{all fused operators}} T_{\text{e\_fused}}^e \\
T_{\text{f\_fused}}^f = T_{\text{f\_up}}^f
$$

The similar idea has been explored in recent work [GARH14]. In this work, we focus
on operator scheduling and replication optimization, and we leave the evaluation of
extension to other optimization techniques as future work.

5.5 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 \([ZHD^{+}04]\), 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 queue and back-pressure mechanism*. Implementation details of BriskStream are given in Section 5.6. According to Equation 5.1, 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 5.5.3.

### 5.5.1 Improving Execution Efficiency

As shown in the previous work \([ZHD^{+}04]\), 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

\(^{2}\)The source code of BriskStream is publicly available at [https://github.com/Xtra-Computing/briskstream](https://github.com/Xtra-Computing/briskstream)
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.

5.5.2 Improving Communication Efficiency

Most modern DSPSs \cite{TT+14, CK+15, ZH+04} 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 Section 5.6). 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.

5.5.3 Discussion on Elasticity

To examine the maximum system capacity, we assume input stream ingestion rate ($I$) is sufficiently large and keeps the system busy. Further, we assume input streams are unbounded without delay, miss, or out-of-order issues – the handling of such is by itself another important line of research. 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 \cite{CNL16, GSW+14, GSW+14}. 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 respawn operator replicas to targeting CPU sockets. Conversely, state migration is needed for stateful operators, which requires state movement and synchronization [GSHW14]. 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 [HJHF14] 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.

5.6 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 5.5 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 [ZHD+04]. Specifically, tuples produced by operator are stored locally, and pointers as reference to tuple are inserted into a communication queue. Together with the aforementioned jumbo tuple design, reference passing delay is minimized and becomes negligible.

## 5.7 Evaluation

Our experiments are conducted in following aspects. First, our proposed performance model accurately predict the application throughput under different execution plans (Section 5.7.2). Second, BriskStream significantly outperforms three existing open-sourced DSPSs on multicores (Section 5.7.3). Third, our RLAS optimization approach performs significantly better than competing techniques (Section 5.7.4). We also show in Section 5.7.5 the relative importance of several of BriskStream’s optimization techniques.
5.7.1 Experimental Setup

We pick four common applications from the previous study [ZHD+04] with different characteristics to evaluate BriskStream. These tasks are word-count (WC), fraud-detection (FD), spike-detection (SD), and linear-road (LR) with different topology complexity and varying compute and memory bandwidth demand.

**Operator selectivity.** As mentioned in Section 5.2.1, 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 5.2.

To examine the maximum system capacity under given hardware resources, we tune the input stream ingress rate ($I$) to its maximum attainable value ($I_{\text{max}}$) to keep the system busy and report the stable system performance. To minimize interference of operators, we use OpenHFT Thread Affinity Library [pl] 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 5.3 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 [int18]. These two machines have different NUMA topologies, which lead to different access latencies and throughputs across CPU sockets. The three major takeaways from Table 5.3 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

---

3Back-pressure mechanism will eventually slow down spout so that the system is stably running at its best achievable throughput.
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 5.7.2, 5.7.3 and 5.7.5. We study our RLAS optimization algorithms in detail on different NUMA architectures with both two servers in Section 5.7.4.

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

---

**Table 5.2: Operator selectivity of LR**

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Input streams</th>
<th>Output streams</th>
<th>Selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatcher</td>
<td>default</td>
<td>position report</td>
<td>≈0.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>balance_stream</td>
<td>≈0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>daily_exp_request</td>
<td>≈0.0</td>
</tr>
<tr>
<td>Avg_speed</td>
<td>position report</td>
<td>avg_stream</td>
<td>1.0</td>
</tr>
<tr>
<td>Las_avg_speed</td>
<td>avg_stream</td>
<td>las_stream</td>
<td>1.0</td>
</tr>
<tr>
<td>Accident_detect</td>
<td>position report</td>
<td>detect_stream</td>
<td>0.0</td>
</tr>
<tr>
<td>Count_vehicle</td>
<td>position report</td>
<td>counts_stream</td>
<td>1.0</td>
</tr>
<tr>
<td>Accident_notify</td>
<td>detect_stream,</td>
<td>notify_stream</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>position report</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Toll_notify</td>
<td>detect_stream</td>
<td>toll_nofity_stream</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>position report</td>
<td>toll_nofity_stream</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>counts_stream</td>
<td>toll_nofity_stream</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>las_stream</td>
<td>toll_nofity_stream</td>
<td>1.0</td>
</tr>
<tr>
<td>Daily_exp</td>
<td>daily_exp_request</td>
<td>default</td>
<td>0.0</td>
</tr>
<tr>
<td>Account_balance</td>
<td>balance_stream</td>
<td>default</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Table 5.3: Characteristics of the two servers we use

<table>
<thead>
<tr>
<th>Statistic</th>
<th>HUAWEI KunLun Servers (Server A)</th>
<th>HP ProLiant DL980 G7 (Server B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor (HT disabled)</td>
<td>8x18 Intel Xeon E7-8890 at 1.2 GHz</td>
<td>8x8 Intel Xeon E7-2860 at 2.27 GHz</td>
</tr>
<tr>
<td>Power governors</td>
<td>power save</td>
<td>performance</td>
</tr>
<tr>
<td>Memory per socket</td>
<td>1 TB</td>
<td>256 GB</td>
</tr>
<tr>
<td>Local Latency (LLC)</td>
<td>50 ns</td>
<td>50 ns</td>
</tr>
<tr>
<td>1 hop latency</td>
<td>307.7 ns</td>
<td>185.2 ns</td>
</tr>
<tr>
<td>Max hops latency</td>
<td>548.0 ns</td>
<td>349.6 ns</td>
</tr>
<tr>
<td>Local B/W</td>
<td>54.3 GB/s</td>
<td>24.2 GB/s</td>
</tr>
<tr>
<td>1 hop B/W</td>
<td>13.2 GB/s</td>
<td>10.6 GB/s</td>
</tr>
<tr>
<td>Max hops B/W</td>
<td>5.8 GB/s</td>
<td>10.8 GB/s</td>
</tr>
<tr>
<td>Total local B/W</td>
<td>434.4 GB/s</td>
<td>193.6 GB/s</td>
</tr>
</tbody>
</table>

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*.

### 5.7.2 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 5.4 compares the measured and estimated process time per tuple ($T$) of each operator. Our estimation generally captures the correlations between
Table 5.4: Average processing time per tuple ($T$) under varying NUMA distance. The unit is nanoseconds/tuple

<table>
<thead>
<tr>
<th></th>
<th>Splitter</th>
<th></th>
<th>Counter</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>From-to</td>
<td>Measured</td>
<td>Estimated</td>
<td>From-to</td>
</tr>
<tr>
<td>S0-S0(local)</td>
<td>1612.8</td>
<td>1612.8</td>
<td>S0-S0(local)</td>
<td>612.3</td>
</tr>
<tr>
<td>S0-S1</td>
<td>1666.53</td>
<td>1991.14</td>
<td>S0-S1</td>
<td>611.4</td>
</tr>
<tr>
<td>S0-S3</td>
<td>1708.2</td>
<td>1994.85</td>
<td>S0-S3</td>
<td>623.07</td>
</tr>
<tr>
<td>S0-S4</td>
<td>2050.63</td>
<td>2923.65</td>
<td>S0-S4</td>
<td>889.92</td>
</tr>
<tr>
<td>S0-S7</td>
<td>2371.31</td>
<td>3196.35</td>
<td>S0-S7</td>
<td>870.23</td>
</tr>
</tbody>
</table>

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 [LKV12]. 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 $\text{relative_error} = \frac{|R_{\text{meas}} - R_{\text{est}}|}{R_{\text{meas}}}$, where $R_{\text{meas}}$ is the measured application throughput and $R_{\text{est}}$ is the estimated application throughput by our performance model for the same application.

The model accuracy evaluation of all applications under the optimal execution plan on eight CPU sockets is shown in Table 5.5. 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%.
Table 5.5: Model accuracy evaluation of all applications. The performance unit is K events/sec

<table>
<thead>
<tr>
<th></th>
<th>WC</th>
<th>FD</th>
<th>SD</th>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured</td>
<td>96390.8</td>
<td>7172.5</td>
<td>12767.6</td>
<td>8738.3</td>
</tr>
<tr>
<td>Estimated</td>
<td>104843.3</td>
<td>8193.9</td>
<td>12530.2</td>
<td>9298.7</td>
</tr>
<tr>
<td>Relative error</td>
<td>0.08</td>
<td>0.14</td>
<td>0.02</td>
<td>0.06</td>
</tr>
</tbody>
</table>

5.7.3 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 5.6 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 [DZSS14], 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 5.7 shows the detailed CDF of end-to-end processing latency of WC comparing different DSPSs and Table 5.6 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 [DZSS14] 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 5.8a 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 5.8b. 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”)
Chapter 5. BriskStream: Scaling Data Stream Processing on
Shared-Memory Multicore Architectures

Figure 5.7: End-to-end latency of WC on different DSPSs.

Table 5.6: 99-percentile end-to-end latency (ms)

<table>
<thead>
<tr>
<th></th>
<th>Brisk Stream</th>
<th>Storm</th>
<th>Flink</th>
</tr>
</thead>
<tbody>
<tr>
<td>WC</td>
<td>21.9</td>
<td>3788.13</td>
<td>5689.2</td>
</tr>
<tr>
<td>FD</td>
<td>12.5</td>
<td>14949.8</td>
<td>261.3</td>
</tr>
<tr>
<td>SD</td>
<td>13.5</td>
<td>12733.8</td>
<td>350.5</td>
</tr>
<tr>
<td>LR</td>
<td>204.8</td>
<td>16747.8</td>
<td>4886.2</td>
</tr>
</tbody>
</table>

Figure 5.8: Scalability evaluation.
in Figure 5.9. 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 5.9. 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 5.10 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 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
recommends 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 5.5) 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 multicore.

Comparing with single-node DSPS. Streambox [MPJ+17] is a recently proposed DSPS
based on a morsel-driven like execution model – a different processing model of
BriskStream. We compare BriskStream with StreamBox [MPJ+17] using WC application.
Results in Figure 5.11 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 [Vtu18]. 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.

5.7.4 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 [GARH14, Kha09]. 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 5.12. $RLAS$ shows a $19\% \sim 39\%$ 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 $119\% \sim 455\%$ improvement over $RLAS_{fix}(U)$. Potentially, $RLAS_{fix}$ may perform well if it applies a suitable $T_f$ of each operator, but it requires tedious tuning process. In contrast, $RLAS$ automatically determines the optimal operator parallelism and placement addressing the NUMA effect.

**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 5.7. 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 5.13. There are two major takeaways.

First, $RLAS$ generally outperforms other placement techniques on both two Servers. FF
can be viewed as a minimizing traffic heuristic-based approach as it greedily allocates
neighbor operators (i.e., directly connected) together due to its topologically sorting step. Several related work [XCTS14, ABQ13] 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 5.14 show that none of the random plans is better than RLAS. It demonstrates that random plans hurt the performance in a high probability due to the huge optimization space.

We further observe two properties of optimized plans of RLAS, which are also found in randomly generated plans with relatively good performance. First, operators of FD and LR are completely avoided being remotely allocated across different CPU-
tray to their producers. This indicates that the RMA overhead, 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., $r_o = r_o = r_i$. 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 5.15. 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 2.4, 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 5.8. 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 5.8. A detailed discussion is presented in Section 5.4.1.

**Table 5.8: Tuning compression ratio ($r$)**

<table>
<thead>
<tr>
<th>$r$</th>
<th>throughput</th>
<th>runtime (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10140.2</td>
<td>93.4</td>
</tr>
<tr>
<td>3</td>
<td>10079.5</td>
<td>48.3</td>
</tr>
<tr>
<td>5</td>
<td>96390.8</td>
<td>23.0</td>
</tr>
<tr>
<td>10</td>
<td>84955.9</td>
<td>46.5</td>
</tr>
<tr>
<td>15</td>
<td>77773.6</td>
<td>45.3</td>
</tr>
</tbody>
</table>
Figure 5.16: A factor analysis for BriskStream. Changes are added left to right and are cumulative.

5.7.5 Factor Analysis

To understand in greater detail the overheads and benefits of various aspects of BriskStream, we show a factor analysis in Figure 5.16 that highlights the key factors for performance. Simple refers to running Storm (version 1.1.1) directly on shared-memory multicores. -Instr.footprint refers to BriskStream with much smaller instruction footprint and avoiding unnecessary/duplicate objects as described in Section 5.5.1. +JumboTuple further allows BriskStream to reduce the cross-operator communication overhead as described in Section 5.5.2. 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 5.2. The major takeaways from Figure 5.16 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.
Scaling Consistent Stateful Stream Processing on Shared-Memory Multicore Architectures

6.1 Introduction

The recent advances in data stream processing system (DSPS) [CFMKP13, WT15, GSS15, CKE+15, TTS+14, KBF+15] in terms of performance, elasticity, and scalability have accelerated their adoption in many emerging use cases. To meet the increasing performance demand, optimizing stream processing on a single multicore machine is critical [ZHD+04], even in a distributed environment. Recent efforts have demonstrated ultra-fast stream processing on large-scale multicore architectures [KWCF+16, MPJ+17, ZHZH19, ZMK+01].

However, a potential weakness of most existing DSPSs is the missing or inadequate support of consistent stateful stream processing, where the application state is globally shared among multiple threads (called executors) of an operator, and state consistency is guaranteed by the system. Until now, state-of-the-art stateful DSPSs such as Storm [TTS+14], Heron [KBF+15], Flink [CKE+15] still left such burden to the application developer to handle – often through key-based stream partitioning to try to completely avoid state sharing [SC14], which is error prone and sometimes impossible. For example, the processing of an input event at one executor may need to access multiple states that are already partitioned and located at different executors.

Efficiently support consistent stateful stream processing is however challenging. First, there is a fundamental conflict between operator parallelism and shared state consistency. In particular, as the shared application state may be read and updated at the same
time by different executors, any uncoordinated concurrent access will result in state inconsistency [BFKT12]. Second, the system needs to ensure any executor is able to see an exact “current” application state whenever it accesses to the state during processing of an input event [CDK+14]. For example, a read to the same state shall never be affected by a write, which is triggered by an event with a larger timestamp. This requires the state access order follows exactly the input event timestamp order [MTZ+15, AMC17] and again contrasts with high operator parallelism, which aim to processing different events concurrently, possibly out-of-order.

Serially processing each input event eliminates the challenge of preserving state consistency but restricts system concurrency [WRE11]. Subsequently, several solutions have been proposed [WRE11, BFKT12, MTZ+15, CDK+14, AMC17] to improve system concurrency while preserving shared state consistency by employing transactional schematics. Specifically, the set of state accesses triggered by processing of a single input event is defined as a state transaction and the concurrent execution of multiple state transactions are managed by the system using varies concurrency control mechanisms. However, they are still poorly utilizing modern multicore architectures. First, they commonly follow a synchronous scheduling paradigm, where an executor must finish all tasks of processing one input event before work on more input events. Despite its implementation simplicity, it restricts intra-event processing parallelism. Second, although multiple different concurrent execution schemes to ensure state consistency were proposed, we find that they are still not able scale different workloads at a large core counts due to severely contention and interference.

In this work, we present TStream, a novel DSPS supporting consistent stateful stream processing that is specifically designed in the context of main-memory multicore architectures. TStream follows previous works [MTZ+15, AMC17] of employing transactional semantics on managing shared state accesses but with much better scalability. The design of TStream is inspired by two observations. First, stream query is predefined and repeatedly invoked on each input event. This prior knowledge allows TStream to perform efficient processing logic decomposition to overlapping stream processing and state transaction processing efficiently. Second, state transaction commitment order is explicitly determined by corresponding input event timestamp. This allows TStream to parallelize state accesses by carefully leveraging
the dependencies within and across different input events with minimized thread contention.

TStream abstracts consistent stateful stream processing as a recurrent three-step procedure. Specifically, when an executor obtains an input event, it (1) processes (e.g., filter, count) on the event; then (2) optionally issues a state transaction to access shared states; Finally, (3) optionally invoke further process that may depend on state access results (e.g., aggregate the state values). Subsequently, by efficiently decoupling the second step from the processing logic, TStream allows an executor to skip and postpone state transaction processing and immediately process more input events. Issued state transactions are hence accumulated and evaluated as a batch periodically, triggered by punctuation signals [TMSF03]. By exploiting the availability of parameter values of issued state transaction, TStream can decompose a collection of issued state transactions into multiple conflict-free atomic operations, which are dynamically organized into multiple operation chains sorted by operation timestamp. This allows TStream to process a batch of transactions with a much higher degree of parallelism under the same consistency requirement.

In summary, we make the following contributions in this work:

- We propose punctuation signal slicing scheduling that allows computation and state access to be efficiently overlapped.

- Leveraging on the availability of parameter values of issued state transaction, TStream adopts a fine-grained parallel processing model to take advantage of multicore resources for evaluating a collection of state transactions. We also discuss several optimization techniques to further improve system performance.

- We implement TStream as well as several state-of-the-art alternative schemes in BriskStream [ZHZH19], a fully functional DSPS optimized for high-performance multicore stream processing. We also experimentally pointed out scalability bottlenecks of prior solutions.

- Overall, this work is the first to provide a practical solution for scaling consistent stateful stream processing on modern multicore architectures. We open source full code of TStream at https://github.com/Xtra-Computing/
6.2 Existing Solutions Revisited

Due to the unique features of consistent stateful stream processing, a number of prior solutions have been proposed [WRE11, AMC17, BFKT12, Tra18, MTZ15]. In the following, we discuss a few representative ones.

**Ordering-lock-based approach (LAL):** An earlier study from Wang et al. [WRE11] described a strict two-phase locking (S2PL) based algorithm that allows multiple state transactions to run concurrently while maintaining state consistency. To ensure the ACID+O properties, it employs a lockAhead process that compares each transaction’s timestamp against a centralized process (e.g., global counter) to ensure it can proceed to insert locks of that transaction.

**Multi-versioning-based approach (LWM):** The same author [WRE11] propose to improve processing concurrency by leveraging on a multi-versioning storage management, where multiple copies of the same application state modified at different timestamp are maintained by the system. It further maintains a counter (i.e., \( \text{lwm} \)) of each state to guard the state access ordering (i.e., ACID+O). Specifically, transactions that need to access a state need to compare their timestamp with the \( \text{lwm} \) counter of that state. Write lock is permitted only if the transaction’s timestamp is equal to \( \text{lwm} \); while read lock is permitted as long as the transaction’s timestamp is larger than \( \text{lwm} \) so that it can read a readily version of the state. During commits, a transaction needs to update \( \text{lwm} \) of all its modified states.

**Static-partition-based approach (PAT):** S-Store [MTZ15] splits the shared application states into multiple disjoint partitions, and hence only needs to guard accessing order in each partition. We name such approach as PAT, and the system maintains an array of global counters, one for each partition. Each transaction needs first compare its timestamp with global counters of its targeting partitions (may be more than one) to synchronize the accesses to those partitions. It is noteworthy that, despite being partitioned, all executors of an operator may still access any partitions of shared states during stream processing, which is fundamentally different from key-based stream
partitioning [KLC08]. Obviously, two transactions still conflict if they need to access the same partition.

We take TP [ACG+04] as an example to validate the effectiveness of prior solutions, Figure 6.1(a) reveals that abundant parallelism opportunities (denoted as *Theoretical bound*) are overlooked under prior solutions (denoted as *Sync execution*) with increasing number of concurrently issued transactions (N). N is up to 40 (on a 40 core machine) under prior schemes, which puts a hard limit on their processing concurrency. Similar observations are made in other existing solutions, hence there is a need for a more scalable solution. In Figure 6.1(b), we further take PAT as an example and evaluate how well it processes under varying available processor resources. We measure the average time spend in each executor on *state access*, which stands for actual time spent in accessing shared states, *access overhead* primarily due to lock acquisition and contention during state access, and *others* including actual processing time on input event and all other system overheads (e.g., context switching). With the increasing number of cores, the access overhead quickly dominates runtime due to the serious contention overhead. As mentioned before, in order to guarantee the ordering consistency requirement (*ACID+O*), prior approaches compare timestamp of state transactions with a (or a set of) global counter(s) to ensure locks are granted strictly in the correct order, which results in centralized contention points and severely degrades system performance.

**In Summary:** There are two common scalability limitations in prior solutions. Firstly, they commonly adopt a simple synchronous scheduling paradigm, where an executor is scheduled to process one input event at one time, and only when all tasks of processing that event is completed, the executor is available for other input events. Despite its implementation simplicity, the parallelism opportunities inside the same event are overlooked (Figure 6.1(a)). Secondly, despite different types of concurrency control schemes were proposed, they either unnecessarily block more state accesses than required (LAL) or involve high maintenance overhead (LWM). Although PAT (i.e., S-Store) can potentially reduce such overhead by careful partitioning shared states beforehand, it quickly falls back to LAL with more multi-partition transactions – a common problem for all partition-based approaches [PCZ12] (Figure 6.1(b)).
Chapter 6. Scaling Consistent Stateful Stream Processing on Shared-Memory Multicore Architectures

6.3 TStream Overview

TStream models consistent stateful stream processing as a recurrent three-step procedure:
1. process input event (e.g., filter, count) and extract parameter values which are used in next step;
2. optionally issue a state transaction with determined parameter values (Section 6.5);
3. conduct further process (e.g., aggregate the state values) that may depend on state access results. Instead of conducting such three-step procedure synchronously as done in previous works, TStream decouples the second step from the processing procedure and postpones the state transaction to be evaluated later. The processing of each event is hence split into two processing phases, including a computation phase and a state access phase.

A punctuation signal [TMSF03] is used to trigger TStream to switch between two processing phases. By tuning the interval of punctuation signals, TStream is able to achieve high throughput with comparable latency compared to existing works. The resulting alternative periodic execution between stream processing and transaction processing is analogy to timeslicing scheduling technique [MEB88] widely adopted in operating systems. However, the difference in system assumptions and designs require us to revisit the problem. For example, TStream needs to handle on-the-fly events at different threads and tracking state access results (details will be discussed in Section 6.4).

State access phase is often the performance bottleneck due to the strict consistency
requirements, and TStream aims to improve system concurrency for evaluating a collection of state transactions. TStream adopts a sorting-based lock-free design for processing state transactions. By exploiting the availability of parameter values among a collection of postponed transactions, TStream decomposes each transaction into multiple atomic operations and regroups operations targeting at the same state together. The formed group is sorted by timestamp and is called operation chain. Subsequently, multiple operation chains can be evaluated independently and concurrently without any centralized contention points.

6.4 TStream Design

TStream achieves speedy consistent stateful stream processing with a combination of static query refinement and dynamic runtime optimization. In this section, we discuss the design of TStream in detail.

6.4.1 Programming model

Inline with many popular DSPSs, TStream expresses an application as a DAG with an API similar to that of Storm [TTS+14]. In addition to that, in each vertex (i.e., operator), user can declare a transaction manager (TXNMANAGER) as the interface of accessing shared states without worrying state consistency violation. This is exemplified by Algorithm 1 that describes how Toll Notification processes input position reports as discussed previously in Section 2.4.

Algorithm 1 Code template of Toll Notification operator

```
TM ← create TXNMANAGER (RoadSpeed_Table, VehicalCnt_Table); // supporting different algorithms

foreach input event e in input stream do
    t ← create AUX_Tuple(e); // AUX_Tuple is a thread-local auxiliary structure for internal usage
    RoadID ← process (t); // extract the ID of road segment where the current position report belonging
    <Speed, Cnt> ← TM.access(RoadID, t); // obtain average speed and unique vehical count of that road
    post_process (t, Speed, Cnt); // compute toll based on obtained road statistics
```

1 In this work, we focus on single- and multi-keys point query, and devote a separate work for supporting more complex query types.
Query Refinement. Thanks to the usage of shared states, TStream do not need to rely on key-based stream partitioning [KLC08] as each executor is allowed to access any part of application states. During query compilation, TStream can hence fuse all operators into a single joint operator [ZMK+01, HSS+03] to eliminate the impact of cross-operator communication, which is known to be a serious performance issue of stream processing on modern multicore architectures [ZHD+04, ZHZH19, PGJ+18]. In this way, input data are simply passed to each executor of the joint operator in a round-robin manner and we can use a switch-case statement in the joint operator to handle different types of input events targeting at different original operators. For example, Road Speed, Vehicle Cnt and Toll Notification operator can be fused into one joint operator, Process. Subsequently, TStream allows Process to be scaled to any number of executors without worrying state consistency violation. This has both greatly simplified the developing of application logic and also ease the complexity of execution plan optimization [ZMK+01, ZHZH19] with much simpler application topology.

6.4.2 Punctuation Signal Slicing Scheduling

As discussed before, TStream adopts an non-conventional way of processing input events, where state transaction processing is postponed. There are three key components to support such postponing efficiently and correctly.

1) Timestamp Generation. For simplicity, we assign each input event (and punctuation) a unique monotonically increasing timestamp. In fact, TStream only requires punctuation’s timestamp to be monotonically increasing in order to progress correctly. Specifically, input events between two subsequent punctuations may arrive arbitrarily with out-of-order timestamp sequence as their issued transactions will be decomposed and sorted automatically during operation chain construction, which are discussed subsequently.

2) AUX_Tuple Maintenance. A key design decision in TStream is to maintain a thread-local auxiliary data structure, called AUX_Tuple, to track information (e.g., parameter values and processing results) of each postponed transaction. The AUX_Tuple is automatically created when an input event is received by the executor. Once a state
transaction is issued, parameter values of it are extracted from the input event and are subsequently recorded into AUX_Tuple for tracking purpose. Figure 6.2 illustrates an example process of creating three AUX_Tuple during processing of three events at two executors of Process operator (i.e., the joint operator of TP). There are two key points to highlight. First, in this example, parameter values of each transaction are directly extracted from the input position report (i.e., key is the road segment ID, values are the vehicle identifier and vehicle speed). We discuss the case when parameter values are not directly extractable in Section 6.5. Second, events may need to be stored temporarily. This is because they may require further process depending on evaluation results of corresponding state transaction. For example, $e_2$ has to be marked as unfinished and stored locally with the executor since it requires the value of state $B$ for further computation. TStream stores such events by maintaining their AUX_Tuple structure locally with the executor, whose “return value” will be updated during transaction execution to support further stream computation.

3) Processing Phases Switching. A punctuation [TMSF03] is a special type of event guaranteeing that no subsequent input events have a timestamp smaller than any events a prior of it. In TStream, punctuation is periodically injected (as done in [CGB+14]) into the input queue of the source operator (e.g., the Parser). When the source operator receives the punctuation event, it broadcasts the punctuation to all its downstream operators. Executor, which declares transaction manager, is switched from computation to state access phase when it receives punctuation from all of its producers. Subsequently, when all state transactions are processed, executors are resumed to computation phase to process further input events. Such a design effectively amortized synchronization overhead among a batch of events and also enlarge inter-event
parallelism opportunities (Section 6.4.3).

To ensure correctness, TStream artificially adds the following two synchronization barriers. First, only when all executors of the same operator are in state access phase, they can proceed to process postponed transactions. This is to ensure all on-the-fly events before the current punctuation are processed at this operator, and issued transactions from all executors are registered to the system. Second, recall that events may be stored at executors for further processing that depends on state access results. To ensure correctness of the post-processing step, all executors are not resume to stream processing until all postponed transactions are fully processed.

Both synchronization points are implemented as a CyclicBarrier [Cen19b] in TStream. Such synchronously switching between stream processing and transaction processing introduces synchronization overhead among executors in TStream. However, as we demonstrate later in experiments, batching and fine-grained parallel transaction processing significantly improves system throughput and successfully amortizes such synchronization overhead.

Figure 6.3 shows an example workflow of switching between phases. (a) punctuations with a timestamp of 5 are received by every executor of Process; (b) all executors are paused and no further input events (e.g., $e_6, e_7$) are allowed to enter the system. Subsequently, actual transaction processing is started to process all postponed state transactions; (c) when all postponed transactions are processed, executors will be notified to process all stored unfinished events, whose AUX_Tuple now contain the value of desired states; (d) finally, executors are switch back to computation phase to process more input events.
6.4.3 Fine-grained Parallel State Access

Guaranteeing the aforementioned ACID+O properties while concurrently processing multiple state transactions is challenging, especially it needs to enforce the external timestamp sequence (i.e., ACID+O) – ensuring sequential processing is in fundamental conflict with concurrent processing by letting more than one thread operate on different events concurrently, possibly out-of-order. The key spirit of existing solutions is that they need to block any future transactions while allowing the current one with correct (i.e., smallest) timestamp to acquire locks. This model is simple to realize but introduces significant synchronization overhead. By adopting punctuation technique [TMSF03], TStream relax the constraint from synchronizing one transaction to a collection of transactions, and hence amortizing the synchronization overheads. Subsequently, it concentrates on improving processing throughput of every batch of state transactions issued during processing all input events arrived between two consecutive punctuations. To achieve that, there are two key components designed in TStream.

1) Transaction Decomposition. The transaction processing in TStream relies on a key data structure, called operation chain. Specifically, once an event’s AUX_Tuple is constructed and initialized, the executor is ready to postpone the issued transaction. To do that, it decomposes the transaction into multiple primitive state access operations and each operation targets at one application state (e.g., one road segment of TP). Those operations are then dynamically partitioned and inserted into lists (called operation chains), where each list is responsible for one state (e.g., one record of the table). Furthermore, operations on one state is automatically sorted by their timestamps.

Intuitively, any concurrent ordered data structure (e.g., self-balancing trees) can be used to implement the operation chain. However, inappropriate implementation can lead to huge overhead in construction and processing. We consider two properties in operation chain. First, it must allow insertion from multiple executors simultaneously while still guaranteeing operations (of the same chain) are ordered by timestamp. Second, it only requires a sequential look up rather than random searches during processing. We hence adopt ConcurrentSkipList [Cen19a] due to its much higher insertion performance and simpler design with smaller constant overhead compared to other alternative designs, such as balance trees [Pug89].
Figure 6.4 illustrates the decomposition process involving processing of three transactions (the same example of Figure 6.2). \(txn_1\) (issued during processing of \(e_1\)) is decomposed into two operations, \(O_1\) and \(O_2\). Each operation is annotated with timestamp (\(ts\)) of its original transaction, targeting state (\(state\)), access type (\(type\)), and parameters (e.g., \(O_1\) is to increase the value of \(A\) by 10). \(O_2\) and \(O_3\) are grouped together to form an operation chain as they target the same state \(B\). As \(O_2\) has a smaller timestamp than \(O_3\), the chain is sorted as \(O_2 \rightarrow O_3\). Similarly, \(O_1\) and \(O_4\) form another two chains as they target different states. Note that, events need to be embedded (by a reference/pointer) to the operation so that their AUX_Tuple structure can be tracked and updated during transaction processing.

2) **Transaction Processing.** Transactions are processed by evaluating the operation chains and there are two cases that we need to consider. *(Case 1):* there is no data dependency among operations. Each operation’s parameter values are extracted from input events (e.g., in the case of \(TP\)). In such case, we can simply sequentially walk (i.e., evaluate) through each operation chain starting from the top (i.e., operation with the smallest timestamp), and different operation chains can be processed concurrently. *(Case 2):* an operation may depend on other operations. For example, a write operation of one state is dependent on a read operation of another state. The general idea of handling data dependency is to process operations without data dependency on others first. Then, process other operations depend on (processed) operations until no one left. However, to keep track of the data dependency while maintaining access ordering at
such operation level is too costly and we propose to process at operation-chain level utilizing multi-versioning, as illustrated as follows.

Given a collection of operation chains, TStream first concurrently process all operation chains with no data dependency on others (we can tag those operation chains during transaction decomposition). Then, it processes operation chains with dependency on the previously processed ones. This iteration process continue until no operation chains are left unprocessed. Such a design has low overhead at tracking data dependencies (only at operation chain level), but operations may be processed out-of-order: depended operations that have larger timestamp may be processed earlier. To handle this issue, TStream has to record multiple versions (updated by different operations with different timestamp) of a state during process if the state is to be read by other operations (i.e., with data dependency). This allows it to ensure subsequent read operations are able to read the correct version (not necessary the latest version) of targeting state. After the current batch of transactions is processed, all versions except the latest version are expired and can be safely garbage collected (i.e., return to single-version store).

Intuitively, TStream performs the best when there is no data dependency among operations in the workload (e.g., TP). In our experiments, we show that TStream is able to perform better or at least similar to previous solutions when the workload heavily contains data dependencies (e.g., SL).

### 6.4.4 More Processing Optimizations

**Transaction Batching.** TStream focuses on achieving a reasonable latency level, with high throughput. Compared to existing approaches, TStream do not immediately process each issued state transaction. Instead, it processes a batch of state transactions periodically. The interval size of two subsequent punctuations (i.e., punctuation interval) hence plays an important role in tuning system throughput and processing latency. Having a large interval, the system needs to wait for a longer period of time to start transaction processing, which increases worst-case processing latency since some events are waiting (i.e., stored with executor) for their issued transactions to be actually processed. Conversely, having a small interval size, the system throughput may drop with insufficient parallelism to amortize synchronization overhead. We evaluate the
effect of punctuation interval in our experiments.

**NUMA-Aware Processing.** Following previous works [PPB+12, PLTA03], we consider three different design options for processing operation chains targeting on multi-sockets multicore architectures. 1) **Shared-nothing:** In this case, we maintain a task pool of operation chains per core. Essentially, operations are dynamically routed to predefined cores by hash partitioning. Then, only one executor is responsible for processing operation chains in one core. The benefits of such configuration are that it minimizes cross-core/socket communication during execution but it may result in workload unbalance.

2) **Shared-everything:** In this case, we maintain a centralized task pool of operation chains, which is shared among all executors to work with.

3) **Shared-per-socket:** In this case, we maintain a task pool of operation chains per socket. Executors of the same socket can hence share their workloads, but not share across different sockets. Workloads are shared among multiple executors under shared-everything and shared-per-socket configuration. A simple strategy is to statically assign an equal number of operation chains (as tasks) to every executor. Such static approach may not always achieve good load balancing. Dynamic work-stealing [BL99] can be applied to achieve better load balancing, where multiple executors (in the same sharing group) continuously fetch and process operation chain as a task from the shared task pool. Such configuration achieves better workload balancing but pays more for cross-core (and cross-socket in case of shared-everything configuration) communication overhead compared to shared-nothing configuration. In our experiments, we evaluate TStream with different configurations.

### 6.5 Discussion

TStream introduces shared states consistency guarantee to state-of-the-art DSPS, and is able to achieve excellent scalability and efficiency using a mostly single-versioned store (it only records multi-versioning of state that are dependent by others in each batch of transactions) without any centrally contented locks as compared to prior solutions. TStream’s transaction processing is also differentiated from many transaction processing systems in its assumptions and hence in its design.
Determined parameter values. In all of our testing applications, the parameter values (including read/write sets) of a state transaction are directly deduced from the first step processing of the input event. This seems a widely accepted assumption in all applications found in prior works [BFKT12, MTZ+15, WRE11]. When this is not applicable, a potential approach is to analysis the read/write sets of the transaction at runtime as done in previous work [RTA14]. However, this adds additional overhead in transaction processing, and subsequently worse overall stream processing performance. In future work, we plan to study a more optimistic approach [WCT16] to relive such overhead in order to handle more extensive use cases.

Abort and redo. A state transaction may need to be aborted because of a write operation causing state consistency violation. When an operation is aborted, the corresponding transaction (and all of its operations) has to be aborted as well. Subsequently, TStream needs to rollback the effect of all operations of that particular transaction from potentially multiple operation chains. After the rollback, TStream re-evaluates affected operation chains with operations of aborting transaction eliminated. This process has to repeat until all operations are evaluated. Such rollback and redo process can be simply achieved by adopting multi-version storage [WAL+03] but it brings considerable overhead. In this work, we assume application workloads do not have to abort transactions or the aborting transaction only contains single write operation and we can hence simply reject that operation with no cascading effect on other operations.

6.6 Evaluation

In this section, we show that TStream manages to better exploit hardware resources compared to the state-of-the-art by a detailed experimental evaluation.

6.6.1 Experimental Setup

We conduct the experiment on a 4-socket Intel Xeon E7-4820 server with 128 GB DRAM. Each socket contains ten 1.9GHz cores and 25MB of L3 cache. The operating system is Linux 4.11.0-rc2. The number of cores devoted to the system and the size of the punctuation interval are system parameters, which can be varied by the system
We vary both parameters in our experiments. We pin each executor on one core and initialize the shared states evenly in each executor. We hence devote 1 to 40 cores to evaluate the system scalability.

**Evaluation overview.** We first show the overall performance comparison of different algorithms on different applications (Section 6.6.3). Then, we further compare different algorithms under varying workload configurations (Section 6.6.4). We use shared-nothing as the default execution configuration in Section 6.6.3 and 6.6.4. We perform sensitivity study to understand the design trade-off in TStream in Section 6.6.5.

### 6.6.2 Benchmark Workloads

To the best of our knowledge, there is no standard benchmark for consistent stateful stream processing. Prior works typically rely on one to two applications for performance evaluation [BFKT12, WRE11, MTZ+15]. According to the four criteria proposed by Jim Gray [Gra92] (i.e., relevance, portability, scalability and simplicity), we design our own benchmark including both simple application (GS) to be used as microbenchmark and three applications (OB, SL and TP) used in prior works [BFKT12, Tra18] covering different aspects of application features.

**Grep and Sum (GS):** Figure 6.5(a) shows the topology of GS containing four operators. Parser continuously feeds input events to Grep operator. Grep issues a state transaction (read-only or write-only) to access a table of records. If an event triggers a read-only transaction, Grep forward the input event with the returned state values to Sum; otherwise, it updates the state and forwards the input event to Sink. Sum performs a summation of the returned state values from Grep. After Sum finishes computation, it emits the computation results as one event to Sink. We use Sink to record the output stream from Sum and Grep to monitor system performance. A table of 10k unique records is shared among all executors of Grep. Each record has a size of 64 bytes, and every state transaction access 10 records. Despite its simplicity, GS shall be applicable to cover a wide range of different workloads by varying parameters such as the Zipfian skew factor of keys (theta), read-write ratio of state access and state partition. We vary all those factors in our experiments in Section 6.6.4 to study different workload configurations.
Chapter 6. Scaling Consistent Stateful Stream Processing on Shared-Memory Multicore Architectures

Record Table

| Record (10k unique): Key (32 bytes String) + Value (32 bytes String) |

(a) Grep and Sum (GS)

(b) Streaming Ledger (SL)

(c) Online Bidding (OB)

Streaming Ledger (SL): SL is suggested by a commercial DSPS, namely Streaming Ledger [Tra18]. It processes events describing wiring money and assets between different user accounts. Detailed descriptions can be found in the white paper [Tra18] and are omitted here for brevity. We implement it with three operators as depicted in Figure 6.5(b). Parser parses received input events and forward to Analysis, which updates user accounts (and/or asset records) based on the received events. The process results are further passed to Sink. Sink reports the processing results to end users. The account and asset tables (each has 10k unique records) are shared among all executors of Analysis. We set a balanced ratio of transfer and deposit requests (i.e., 50% for each) in Parser. Transaction length is four for transfer request (i.e., each touches four records) and is two for deposit request.
Online Bidding (OB): Figure 6.5(c) represents a simplified bidding system [TZ05], where users bid or sell items online. There are three types of trade requests as input events to process. (1) bid request reduces the quantity of an item if the bid price is larger or equal to the asking price, otherwise rejected. If the item has an insufficient quantity, the bid request is also rejected. (2) alter request modifies prices of a list of items. (3) top request increases the quantity of a list of items. The application can be implemented with the following four streaming operators. Parser continuously emits events describing trade requests from either buyers or sellers. Auth. authenticates the requests such as validating the request’s IP address and dispatches valid requests for further process. Process handles the aforementioned three types of requests. Sink records the output stream from Process. During the processing of each event, Process may need to read/update item table, which contains item id, price and quantity information. Auth. and Process have a selectivity of one, that is, they always generate one output for each input event. The length of both the alter and the top request is 20 so that each request accesses 20 items. The length of the bid request is one. The price and quantity of each item are randomly initialized before execution and is kept the same among different tests. The ratio of the bid, alter, and top requests is configured as 6:1:1.

Toll Processing (TP): we use the dataset from previous work [ACG+04], which accesses 100 different road segments with a skew factor of around 0.2. Transaction length of TP is one for each table, hence there is no multi-partition transaction. However, transactions access each road segment non-consecutively, and hence the counter of each partition need to be increased multiple times after each transaction commits under previous solutions.

In summary: Our benchmark covers different aspects of application features. First, our applications cover different runtime characteristics. Specifically, TP spends 39% of total time (run at a single core) in stream processing, and this ratio is 29% and 22% for SL and OB, respectively. GS spends relatively less time in stream processing (13%), and more time is spent in shared state access. Second, they cover different types of state transactions. Specifically, GS has read-only and write-only request. SL has write-only and read-modify-write (w/ cross states dependency) request. OB has write-only, read-modify-write (w/o cross states dependency) request. TP has read-modify-write (w/o cross states dependency) request. It is noteworthy that key-based
stream partition [KLC08] are not applicable here as processing of each input event may request to access arbitrary states and events are randomly shuffled among all executors running concurrently in different threads. As a result, any uncoordinated accesses (i.e., update and read) to the same state can cause state inconsistencies.

### 6.6.3 Overall Performance Comparison

We first compare TStream with other schemes on different applications with varying physical resources. Besides three competing schemes, we also examine the system performance when locks are completely removed from LAL scheme, which is denoted as No-Lock representing the system performance upper bound without consistency guarantee. For GS, SL and OB, we set state access skew factor to be 0.6, and the length and ratio of multi-partition transaction is set to 4 and 25%, respectively under PAT. That is, each multi-partition transaction needs to access four different partitions, and 25% transactions are multi-partition transactions. We also configure their state accesses to each partition with monotonically increasing timestamp of one so that the global counter of each partition is increased exactly once after each access.

**Performance comparison.** The throughput comparison results are shown in Figure 6.6, and there are three major observations. First, TStream outperforms all other schemes while preserving the same consistency properties at large core counts for all applications except SL. However, there is still large room to further improve TStream to achieve the performance upper bound indicated by No-Lock scheme. Second, there is a significantly increased synchronization overhead due to data dependency handling in TStream for SL and it performs similar to prior solutions. Optimistic execution strategy [WCT16] may be adopted to reduce such synchronization overhead and further improve system performance. Due to its considerable complexity, we defer it as future work. Third, PAT performs generally better than LAL and LWM as it avoids blocking when transactions access different partitions. However, PAT performs even worse than LAL for TP although there are no multi-partition transactions. The reason is that transactions access each road segment non-consecutively, and executor needs to increase the global counter of each partition by multiple times (instead of once in case of consecutive access) in order to allow the next transaction to proceed. As a result, excessive access to a large set
of global counters degrades system performance significantly.

**Runtime breakdown.** Following the previous work [YBP+14], we now report how much time is spent on different components of processing a state transaction. 1) *Useful* is the time that the transaction is really operating on records. 2) *Lock* stands for the total amount of time that a transaction spends due to lock insertion. 3) *RMA* stands for remote memory access overhead that a transaction spends in global counter access (in case of LAL, LWM, and PAT) or in transaction decomposition (in case of TStream). Index lookup and actual state access may also cause remote memory access for all schemes run on multi-sockets. 4) *Sync* is the time that a transaction spends due to synchronization. This is a unique component pay for guaranteeing ACID+O properties, where a transaction may need to synchronize among executors per event in LAL, LWM and PAT or synchronize per punctuation interval in TStream. 5) *Others* include all other system overheads including index lookup (w/o RMA), context switching, auxiliary data structure creation, etc.
We use TP as an example to study the runtime breakdown in different algorithms. Similar observations are made in the other three applications. Figure 6.7 compares the time breakdown when the system is run on single or four sockets. There are three major takeaways. First, No-Lock spends more than 60% of the time in Others, which prevents the system to further scale. Our further investigation reveals that the index look-up is the root cause of such performance degradation. We defer the study of more scalable index design in future work and concentrate on concurrent execution control in this work. Second, Sync overhead dominates all lock-based algorithms (LAL, LWM, and PAT) regardless of the NUMA effect. Although LWM does spend less time in Sync compared to LAL as read may not be blocked by write resulting in higher processing concurrency, it spends more time in reading and updating the lwm variable (grouped in Others overhead). Third, NUMA overhead is significant in TStream when running on four sockets. Our fine-grained operation-chains based processing paradigm exploits high parallelism opportunities while still guaranteeing the same consistency requirement as prior solutions. Unfortunately, it also brings high communication overhead, which is unavoidable during transaction decomposition since each executor may issue transaction that touches arbitrary states. Nevertheless, as discussed in Section 6.4.4, TStream supports multiple NUMA-aware configurations for transaction processing. We study their effectiveness later in Section 6.6.5.

Figure 6.7: Runtime breakdown per state transaction of TP.
Chapter 6. Scaling Consistent Stateful Stream Processing on Shared-Memory Multicore Architectures

![Graph](image)

(a) Varying ratio of multi-partition txns (length=6). (b) Varying length of multi-partition txns (ratio=50%).

**Figure 6.8**: Multi-partition transaction evaluation.

![Graph](image)

(a) Read/write workload ratio. (b) State access skewness.

**Figure 6.9**: Varying application workload configurations of GS.

### 6.6.4 Workload Sensitivity Study

We now use GS as an example to evaluate different algorithms under varying workload configurations in detail.

**Multi-partition transaction percentage.** We first study the effect of states partitioning. We use a simple hashing strategy to assign the records to partitions based on their primary keys so that each partition stores approximately the same number of records. As a common issue of all partition based algorithms [PCZ12], the performance of PAT is heavily depended on the length and ratio of multi-sites transactions.

We first configure multi-partition transaction to touch 6 different partitions of the shared states. We then vary the percentage of multi-partition transactions in the workload. The
results are shown in Figure 6.8(a). Since PAT is specially designed to take advantage of partitioning, it has a much lower overhead for synchronization when no multi-partition is required (i.e., ratio=0%). However, it performs worse than TStream even without any multi-portion transaction. This is due to its synchronous execution model, which overlooks parallelism opportunities within the processing of a single event. Furthermore, as expected, its performance degrades with more multi-partition transactions as they reduce the amount of parallelism. A similar observation can be found in Figure 6.8(b), where we vary the length of multi-partition transactions and fix its ratio to be 50%. Note that, there is no difference in performance whether or not the workload contains write request under the PAT scheme as transactions are always bottlenecked by partition locks. In the following studies, we set the multi-partition ratio to be 50% under PAT.

Read request percentage. We now vary the percentage of events that trigger read
request to shared states from 0% (write-only) to 100% (read-only). In this study, we
disable stream computation ($n=0$) of GS and focus on evaluating the efficiency of state
transaction processing. We also set the key skew factor to be 0, and hence the state
is accessed with uniform frequency. Figure 6.9(a) shows the results and there are
two major observations. First, varying read/write requests ratio has a minor effect
on system performance under prior schemes, LAL, LWM and PAT. This is because
their execution runtime is dominated by synchronization overhead. Second, TStream
performs generally worse with more read request. This is because the system has to write
back the state value to event’s (which triggers read request) AUX_Tuple after transaction
evaluation is finished. An interesting point to take note is that TStream’s performance
increases slightly under read-only workload compared to mixture workload due to
hardware prefetchers. When there are both read and write to shared states, hardware
prefetchers are not effective as each prefetch can steal read and or write permissions
for shared blocks from other processors, leading to permission thrashing and overall
performance degradation [JHL06].

State access skewness. In this study, we configure a write-only workload to examine
how algorithms perform under contented state update. To represent a more realistic
scenario, we model the accessing distribution as Zipfian skew, where certain states
are more likely to be accessed than others. The amount of skew in the workload is
determined by the parameter, $\theta$. Figure 6.9(c) shows that TStream achieve high
performance even under high skewness. LWM performs even worse under contented
workload as there is more contention on updating $lwm$ counter of the same record
during transaction commit. PAT performs worse with increasing skewness because
of the more intensive contention on the same partition lock. In contrast, TStream is still able to discover sufficient parallelism opportunities among a large collection of transactions (punctuation interval is set to 500).

### 6.6.5 System Sensitivity Study

In this section, we study the effect of processing optimizations (Section 6.4.4) in TStream.

**Varying punctuation interval.** The number of transactions to handle between two subsequent punctuation plays a critical role in TStream’s performance. Figure 6.10(a) shows that the performance of TStream generally increases with larger punctuation interval. When it is larger than 500 tuples, TStream achieves its best performance for all applications. It also shows that large punctuation interval is especially beneficial for TP. The reasons are two folds. First, there are only 100 different road segments, and transactions are heavily contented within a small batch of transactions. Second, TP has the highest stream processing ratio, TStream achieves a higher stream processing concurrency with large punctuation interval, and the system performance hence increases significantly.

Following the previous work [DZSS14], 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. Figure 6.10(b) shows the processing latency of TStream under varying punctuation interval. Thanks to the significantly improved throughput, TStream achieves very low processing latency. When TStream achieves its best performance (around punctuation interval of 500), its processing latency is only around $0.23 \sim 0.56$ ms, which satisfies many practical use cases [DZSS14], hence TStream is practically useful. Further increasing punctuation interval, the latency becomes worse as there is no significant throughput improvement. Figure 6.11 further shows that TStream achieves comparable and sometimes even lower processing latency compared to the state-of-the-art.

**Effect of NUMA-aware optimizations.** We now compare different NUMA-aware processing configurations of TStream including share-nothing, share-everything, and shared-per-socket. Work-stealing can be further enabled in the latter two configurations. By taking shared-per-socket as an example, we show in Figure 6.12 that work-stealing
does significantly improve the system performance, and shall be enabled when the shared configuration is applied. However, Figure 6.13 shows that TStream achieves the best performance for all applications under shared-nothing configuration. This indicates that cross-core and cross-socket communication shall be always avoided for all testing applications. Nevertheless, we plan to investigate more applications that may be more sensitive to workload unbalancing rather than communication overhead.
Conclusion

In this thesis, we presented our exploration on the design and implementation of scalable multicore main-memory DSPSs for supporting modern stream analytic workloads. We performed a comprehensive study on the DSPS architectures, and optimized the system performance by investigating and addressing the scalability bottlenecks from two major DSPS components, including execution plan optimization and state management.

7.1 Contributions

The main contributions of this thesis are summarized as follows.

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

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

Scaling Consistent Stateful Stream Processing on Shared-Memory Multicore Architectures. We have proposed TStream aiming at scaling stateful stream processing on shared-memory multicore architectures under strict consistency requirement. TStream achieves high scalability via two key designs including 1) punctuation signal slicing scheduling, which allows stream processing and transaction processing to be efficiently overlapped and 2) fine-grained parallel state access, which maximize execution concurrency for processing a collection of state transactions. Our extensive experiments on a 40-core server with different algorithms confirm the superiority of TStream’s designs.

7.2 Discussions

In this section, we discuss a few additional issues that are not covered by previous contents.

Impact of platforms. The experiments conducted throughout the work of this thesis are mostly based on high-end servers each with 4-8 CPU sockets. While commodity servers widely deployed in today’s data centers are still equipped with 1 or 2 sockets for their lower price-per-core, which seems to limit the practical usability of the proposed techniques and systems. We argue the contribution of this thesis from the following two aspects. On one hand, the systems (i.e., BriskStream and TStream) proposed in this thesis are applicable to commodity servers and can significantly outperform existing systems as can be seen from the comparison results at small core counts (e.g., Figure 5.8a). On the other hand, by improving the performance of stream processing at high-end server, this thesis opens opportunities for even cheaper distributed stream processing over a
small size cluster with multiple high-end servers, which may well complement with existing infrastructures. Due to the severely under-utilization of hardware resources at each compute node, existing DSPSs (e.g., Flink) will not be able to take advantage of such “high-end clusters”.

**Ingestion design.** Ingestion concerns how input stream actually flows into the system. In this thesis, we omit the ingestion component by assuming input data are always ready to be consumed at “Spout” operators. Specifically, “Spout” operator initializes an array of input tuples in its local memory and continuously feed the data into the other parts of the application. In reality, “Spout” receives input data from external systems, e.g., receiving xml files over tcp/ip network. How the network adapter is connected to the NUMA node will affect the speed of “Spout”, i.e., it runs faster if it is allocated near the network adapter. Essentially, this turns “Spout” into a consumer operator and our optimization algorithm can be naturally extend to capture such effect: instead of assuming “Spout” has no producer and always run at the same speed no matter it allocates, we can consider a virtual operator being producer of “Spout”. Then, assume the virtual operator is allocated at the NUMA node with network adapter, our algorithm will try to determine the suitable placement and parallelism configuration of the “Spout” operator by taking care of the relative location of “Spout” and the virtual operator, similar to other operators.

**Application demands.** A large volume of data is generated in real-time or near real-time and has grown explosively in the last few years. For example, IoT (Internet-of-Things) organizes billions of devices around the world that are connected to the internet through sensors or Wi-Fi. IHS Markit forecasts [iot19] that 125 billion such devices will be on service by 2030, up from 27 billion last year. With the proliferation of such high-speed data sources, numerous data-intensive applications are or will be deployed in real-world use cases exhibiting latency and throughput requirements that cannot be satisfied by traditional batch processing models. Despite the massive effort devoted to big data research, many challenges remain. For example, the new 5G network promises blazing speeds, massive throughput capability and ultra-low latencies [PC], thus bringing more demands for performance critical stream applications. First, high-throughput stream processing is essential to keeping up with data streams in order to satisfy the *Process and Respond Instantaneously* [ScZ05] requirement. Second, *Keep the Data Moving* [ScZ05]
is one of the fundamental design requirements of DSPs. We hence aims to process input data “on-the-fly” without storing them, which also requires the system to be able to handle input stream as fast as possible without having to queue up input streams. Furthermore, new media applications such as live audio streaming services [WLCH18] also challenges existing systems in terms of new processing paradigms. Despite the massive effort devoted, many challenges remain and we foresee the continuous growing interest in the area that this thesis targets at.
Future Directions

The proliferation of high-rate data sources has accelerated the development of next-generation performance-critical DSPs. For example, the new 5G network promises blazing speeds, massive throughput capability and ultra-low latencies [PC], thus bringing the higher potential for performance critical stream applications. While the mechanism proposed in the previous chapters allow the DSPs to better utilize modern hardware capabilities, many challenges remain. In the following, we list some future directions.

**Computation Optimization.** Complex event processing (CEP) has become increasingly important for stream applications in which arbitrarily complex patterns must be efficiently detected over high-speed streams of events. Online finance, security monitoring, and fraud detection are among the many examples. Pattern detection generally consists of collecting primitive events and combining them into potential (partial) matches using some type of detection model. As more events are added to a partial match, a full pattern match is eventually formed and reported. The costly nature of CEP [ZDI14] and the stringent response time requirements of stream application has created significant interest in accelerating CEP. However, there is no one standard implementation for CEP. Popular CEP mechanisms include nondeterministic finite automata (NFAs) [ZDI14], finite state machines [AcT08], trees [MM09], and event processing networks [REG11]. Meanwhile, several parallelization paradigms have been proposed in the literature including both data parallelism [BDWT13] and task parallelism approach [BGHJ09]. However, there is a lack of a comprehensive evaluation of prior implementations in a modern multicore architectures. We plan to conduct such a study and aim to answer the question of how to design an efficient CEP system to
better utilize shared-memory multicore architectures.

**Communication Optimization.** Most DSPSs are designed with *Keep the Data Moving* [ScZ05] as the fundamental design principle, and hence aims to process input data “on-the-fly” without storing them. Subsequently, message passing is often a key component in current DSPSs. Researchers have attempted to improve the communication efficiency by taking advantage of the latest advancement in network infrastructure, compression using hardware accelerator, and efficient algorithms exploiting new hardware characteristics [MJP+19, PGJ+18, ZMK+01]. However, a model-guided approach to balance the tradeoff between computation and communication overhead is still in general missing in existing works. We are designing a new DSPS with cost-model guided paradigm to decide when compression shall be involved and which compression scheme shall be used in order to achieve optimal system throughput.

**Efficient State Retrieval.** Modern streaming applications with complex execution logic often need to maintain and access large internal states. The naive scan retrieval overhead grows rapidly with larger state size and quickly become the bottleneck of overall system performance. It is therefore important to explore ways, such as index management, to relieve such bandwidth pressure. However, maintaining index structures also brings maintenance overhead [LOWY15]. Witnessing those issues, we are developing new algorithm and mechanism to better balance these conflicting aspects.
Publications

In this chapter, we list the main articles (published or under-review) contributed to the thesis and also articles authored or co-authored during the author’s Ph.D. journey.

9.1 Selected Publications

1. Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, Thomas Heinze, “Revisiting the design of data stream processing systems on multi-core processors”. International Conference on Data Engineering (ICDE, 2017)

2. Shuhao Zhang, Jiong He, Amelie Chi Zhou, Bingsheng He, “BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures”. International Conference on Management of Data (SIGMOD, 2019)

3. Shuhao Zhang, Yingjun Wu, Feng Zhang, Bingsheng He, “Scaling Consistent Stateful Stream Processing on Shared-Memory Multicore Architectures”. To be submitted

9.2 Other Publications

1. Shuhao Zhang, Feng Zhang, Yingjun Wu, Paul Johns, Bingsheng He. “Hardware-Conscious Stream Processing: A Survey”. Currently minor revision in SIGMOD Record

2. Shuhao Zhang, Hoang Tam Vo, Daniel Dahlmeier, Bingsheng He. “Multi-Query Optimization for Complex Event Processing in SAP ESP”. International Conference on Data Engineering, Industrial and Application (ICDE 2017)


6. Jiong He, Shuhao Zhang, Bingsheng He “In-Cache Query Co-Processing on Coupled CPU-GPU Architectures”. *International Conference on Very Large Data Bases (VLDB, 2015)*.
References


References


ACM SIGMOD International Conference on Management of Data, SIGMOD ’13, pages 969–972, New York, NY, USA, 2013. ACM.


References

ConcurrentSkipListSet.html, 2019.


References


References


[iot19] Number of connected IOT devices will surge to 125 billion by 2030, ihs markit says. https://technology.ihs.com/596542/number-of-


[KM08] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. SIGPLAN Not., 43(6):114–124,
June 2008.


References


References


References


References


References


References


References


