Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core

Navendu Jain*

University of Texas at Austin
Austin, TX 78712
nav@cs.utexas.edu

Lisa Amini, Henrique Andrade, Richard King,
Yoonho Park, Philippe Selo, Chitra Venkatramani
IBM T. J. Watson Research Center
Hawthorne, NY 10532
{aminil, hcma, rpk, yoonho, pselo, chitrav}@us.ibm.com

ABSTRACT

Stream processing applications have recently gained significant attention in the networking and database community. At the core of these applications is a stream processing engine that performs resource allocation and management to support continuous tracking of queries over collections of physically-distributed and rapidly-updating data streams. While numerous stream processing systems exist, there has been little work on understanding the performance characteristics of these applications in a distributed setup. In this paper, we examine the performance bottlenecks of streaming data applications, in particular the *Linear Road* stream data management benchmark, in achieving good performance in large-scale distributed environments, using the Stream Processing Core (SPC), a stream processing middleware we have developed.

First, we present the design and implementation of the Linear Road benchmark on the SPC middleware. SPC has been designed to scale to tens of thousands of processing nodes, while supporting concurrent applications and multiple simultaneous queries. Second, we identify the main performance bottlenecks in the Linear Road application in achieving scalability and low query response latency. Our results show that data locality, buffer capacity, physical allocation of processing elements to infrastructure nodes, and packaging for transporting streamed data are important factors in achieving good application performance. Though we evaluate our system primarily for the Linear Road application, we believe it also provides useful insights into the overall system behavior for supporting other distributed and large-scale continuous streaming data applications. Finally, we examine how SPC can be used and tuned to enable a very efficient implementation of the Linear Road application in a distributed environment.

*This work was done while the author was an intern at IBM T. J. Watson Research Center.

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

1. INTRODUCTION

With the widespread use of digital systems, there is an emerging set of applications that involve processing of high-volume, continuous data such as text and transactional data, digital audio, video and image data, instant messages, network packet traces, and sensor data, for purposes such as filtering, aggregation, and correlation. The Stream Processing Core (SPC) [6] is a distributed stream processing middleware designed with the goal of supporting such stream data processing applications over large-scale configurations. SPC provides a new application execution environment for user-developed processing elements that ingest, filter, and most importantly, analyze data streams.

There exists a large body of work on stream processing systems such as Aurora [18], Borealis [5], TelegraphCQ [11, 14] and STREAM [8]. However, there are significant differences in the goals and architecture of the SPC and these systems. SPC has been designed to support distributed, large-scale stream mining applications. The distinguishing features of the SPC architecture are dynamic application composition, stream discovery, and reuse across applications.

Although there has been a lot of interest in the field of distributed stream processing, there is a dearth of good benchmarking tools, in particular, application and low-level system-benchmarking tools. An exception is the Linear Road benchmark [9] developed by the Aurora and the STREAM teams, which focuses on application-level performance. Linear Road is a streaming data benchmark that can be used to compare the performance characteristics of stream-based data management systems (SDMS) relative to each other and to alternative (e.g., relational database) systems. In essence, the benchmark is designed to evaluate the performance of a given SDMS in processing high-volume continuous and historical queries without violating their accuracy and real-time query response requirements.

What distinguishes our work from previous implementations of Linear Road are two key aspects. First, we evaluate the SPC using the Linear Road application employing multiple distributed configurations. Second, we study and focus on the behavior of the streaming infrastructure support for large-scale continuous and historical queries in terms of scalability, latency, and resource utilization, among others. In this paper, we restrict ourselves to describing and evaluating the SPC support for the Linear Road application, though many of our design decisions and performance results are potentially pertinent to other streaming applications that require distributed and large-scale continuous query processing.

This paper makes two important contributions: (1) We demonstrate a highly scalable distributed implementation of the Linear Road application, and (2) We highlight the importance of addressing performance bottlenecks and tuning the stream processing middleware in order to adapt to the application requirements and to the runtime environment. To the best of our knowledge, this is the first large-scale performance study analyzing system bottlenecks in a streaming data application.

This paper is organized as follows. Section 2 describes the related work. In particular, we briefly highlight the key features and differences between SPC and other streaming systems highlighted in order to set the stage for the performance evaluation we conducted. Section 3 provides a brief architectural overview of the SPC, describing some of its key system design features. Section 4 presents background description of the Linear Road benchmark. Section 5 describes the prototype implementation of the Linear Road application using SPC. Section 6 covers the extensive experimental evaluation of the application and the middleware to provide better performance. And, finally, Section 7 summarizes our findings and describes some of the ongoing work in terms of improving the SPC middleware architecture, as a result of our experimental study.

2. RELATED WORK

Recently a large number of projects have aimed at supporting streamed data processing. DataCutter [10] is a middleware for decomposing applications into processing *filters* responsible for implementing *subset* and *reduction* operations over streams. Here, the operators are well known with predictable performance, and the connections between application components are determined statically, i.e., before the application is deployed. StreaMIT [13, 17] creates streams based on compile-time connection determination and does not accommodate dynamic application composition. A different approach is employed by stream processing systems such as TelegraphCQ [11, 14], Aurora [4], Borealis [5], and STREAM [8] where significant progress has been made in providing support for streamed data manipulation from a database-centric perspective. In these systems, the stream operators correspond to relational operators that process streams of tuples individually or over windows.

SPC, on the other hand, was designed to address large-scale (i.e., leveraging potentially thousands of computational nodes) distributed stream mining applications. It is designed with the assumption that the system is constantly overloaded with respect to the available resources. For this reason, SPC has to use resources intelligently in order to minimize the loss of useful data. A key distinguishing feature of SPC is dynamic application composition which enables stream connections to be made and broken dynamically as new applications and new data sources join and leave the system. Further, this dynamic composition renders the support of apriori query operators having predictable performance useless.

Benchmarks for evaluating stream processing systems are beginning to emerge and there has been some work in evaluating stream processing systems using the Linear Road application. Linear Road was first proposed as a stream data management application benchmark by Arasu et al. [9] and is a congestion-based tolling application for vehicles on expressways. The authors reported Linear Road performance of 2.5 expressways over a pre-release version of Aurora's commercial product running on one node. The STREAM [8] system implemented the benchmark by expressing queries in Continuous Query Language (CQL) using a single node. However, their study did not report any performance numbers. The Infosphere project [15] also employed the Linear Road application using STREAM to evaluate quality of service issues. In a related study [16], the application was separated into three modules – a data source, a STREAM server, and a data sink. Although each module was hosted on a different node, a distributed implementation of the application was not employed and the effects of system parameters on the performance of the application were not studied. Moreover, their implementation only provided support for a small subset of the continuous queries supported by Linear Road. In comparison, we use SPC to create a completely distributed version of Linear Road supporting all continuous and historical queries as well as perform a detailed study of the effects of some of the system parameters such as buffer sizes, system overheads, and resource allocations on the application.

3. SPC ARCHITECTURE

In this section, we first describe the architecture of the SPC stream processing system and later highlight various optimizations in SPC to address performance challenges of streaming applications in Section 3.1.

The Stream Processing Core [6] is a distributed stream processing infrastructure that provides the architectural substrate and services to enable highly efficient mining of large amounts of streaming data.

The infrastructure enables the execution of multiple stream-processing applications simultaneously on a large cluster of machines. Each application is expressed in terms of a dataflow graph consisting of processing elements (PEs) that consume and produce streams of data through input and output *ports*, respectively. Each PE processes stream data objects that it receives, and possibly filters or annotates them with additional information and publishes them. Each PE is associated with a *PE descriptor* that specifies its function, as well as its input and output ports. The system supports a model where stream connections are created between input and output ports based on a publish-subscribe model. That is, PEs specify the streams that they produce on their output ports using a *stream descriptor* and declare the characteristics of the streams that they are interested in consuming on their input ports using a *flow specification* expression. The system then dynamically determines the stream connections among the PEs at run-time by matching stream descriptors to flow specifications. This allows PEs to *discover* new streams that match their flow specifications as and when these new streams become available, which is essential for live, sense-and-respond applications.

SPC also allows new applications to reuse streams that are being produced in the system by other applications. This can result in significant resource savings and in the ability to discover useful information from knowledge extracted by other applications over an ever-changing set of data sources. The publish-subscribe model and the stream reuse features enable dynamic application composition, which is one of the key features that distinguishes SPC from other systems.

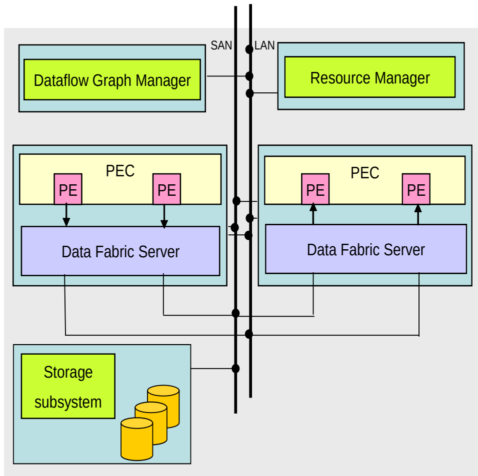

Figure 1: The SPC consists of various components that together provide services to run stream-processing applications.

Figure 1 provides an architectural overview of the SPC system. The main components of the SPC system are:

Dataflow Graph Manager (DGM): The DGM is the component responsible for determining streaming connections among PEs. It matches stream descriptions of output ports with the flow specifications of input ports and performs dynamic application composition, conveying routing information to the Data Fabric.

Data Fabric (DF): The DF is the distributed data transport component comprising a set of daemons, one on each node supporting the system. It is in charge of establishing the transport connections between PEs and moving stream data objects (SDOs) from producer PEs to consumers. The DF is also responsible for choosing the appropriate transport and for making local resource adjustments to achieve appropriate flow-balancing among PEs. This is to ensure stable operation in the face of bursty traffic workloads. The adaptive control algorithm used by the Data Fabric is described in a different study [7].

Resource Manager (RM): The RM determines placement of PEs and periodically makes global resource allocation decisions (e.g., shares of CPU and network bandwidth assigned to each processing element) for PEs and streams based on runtime resource usage measurements gathered from the DF daemons and the PE Execution Containers.

PE Execution Container (PEC): The PEC is a container for processing elements, providing a runtime context and access to the SPC middleware. It acts as a security barrier, preventing user-written PE code from adversely affecting the middleware and other processing elements. The PEC also monitors resource usage on behalf of the Resource Manager.

We have developed a fully functional prototype of the SPC that is being stress-tested under different application workload profiles on a large Linux cluster employing 85 nodes interconnected by a Gigabit switched Ethernet network.

3.1 Performance Challenges

Some of the key characteristics of distributed streaming applications are the large volume of data to be analyzed, and the high data rates generated by multiple distributed data sources such as audio and video. Early stages of the application cull out a lot of noise from the torrents of source data and later stages in the application perform "deeper" processing on smaller amounts of data. Hence the processing elements constituting the system range from those that perform small amount of processing on large volumes of data to those that perform a large amount of processing on relatively lower volumes of data, leading to a mix of I/O-bound as well as CPU-bound processing elements. Also, since the volume of data is high, it is unrealistic for applications to store the full history of a stream in memory. In that respect, they are also memory-bound. Given these constraints, the system has to effectively apportion resources such as CPU, network bandwidth, and memory such that the overall system utility is maximized.

Towards this end, SPC supports many optimizations, some of which are evaluated in the context of Linear Road in this paper.

- SDO filtering: Besides the stream-level flow-specification, PEs can specify a filter expression to subscribe to a subset of stream data objects in streams. The Data Fabric then filters out unwanted objects at the source and delivers only the matching SDOs to PEs, thereby saving resources.
- Events: PEs can subscribe to system events, which let them know about resource allocation changes, stream connections to their ports as well as their SDO-filter expressions. A PE may adapt its algorithm or processing based on this information.
- Dynamic copies of PEs: Since the system uses a dynamic stream-connection model using flow specifications, the application composer or the resource manager can deploy multiple copies of a bottleneck PE and split the data among the copies based on the values of attributes in the filter expression.

We refer the interested reader to [6] for additional details on the SPC architecture.

4. LINEAR ROAD BENCHMARK

Linear Road simulates the traffic characteristics of a simple urban expressway system using *variable tolling* where tolls are calculated based on dynamic traffic conditions such as traffic congestion and accident proximity. The application processes an input data stream of *position reports* specifying the position of a vehicle on an expressway. The application is also responsible for answering *historical queries* (e.g., travel-time estimation, account balances, etc.) issued by a vehicle with some fixed probability every time it emits a position report. Linear Road measures the performance of an SDMS in terms of the number of expressways that the system can support while meeting both response time and correctness requirements of queries.

We first give an overview of the Linear Road benchmark (Section 4.1), and later describe the associated continuous and historical queries along with their response time and accuracy requirements (Section 4.2). More details on Linear Road can be found in the Linear Road benchmark paper [9] and the associated web site [2].

4.1 Linear Road Overview

In the Linear Road benchmark, there are L multi-lane expressways each 100 miles long, that run parallel to each other 10 miles apart. Every expressway has four lanes each in east and west directions: 3 travel lanes (# 1-3) and one lane for both entry (lane #0) and exit (lane#4) ramps. Each expressway is divided into 100 mile-long segments in each direction (east and west), and each segment has 100 entrance ramps and 100 exit ramps. The entrance and exit ramps for every segment are each a third of a mile long, and allow vehicles to accelerate or decelerate. For simplicity, there are no highways that run north- or southbound. Figure 2 shows the geometry of a segment of the Linear Road expressway system.

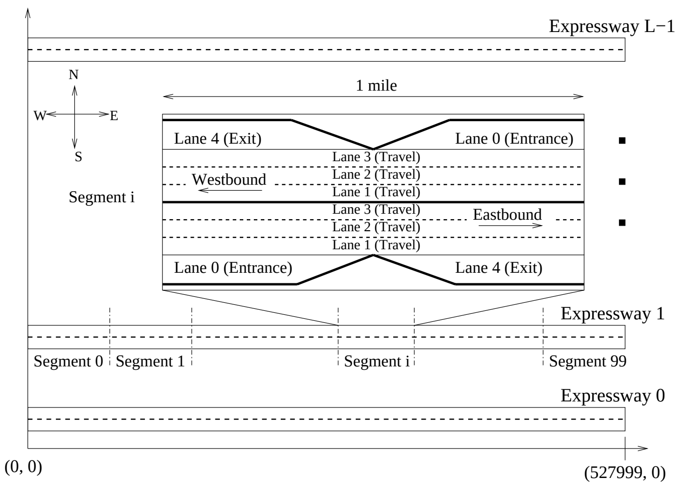

Figure 2: Geometry of a segment of a Linear Road Expressway.

The input data to the Linear Road benchmark is generated by the MIT Traffic Simulator (MITSIM) [1] and stored in a data file. The traffic simulator generates a set of vehicles, each of which undertakes at least one journey that begins at an entry ramp in a segment and finishes at an exit ramp in another segment on the same expressway. For each trip, the selected source location of a vehicle is uniformly distributed over all the chosen ramps on the chosen expressway. A separate module, the data driver, performs the task of reading the file generated by the traffic simulator, and feeding the data to an SDMS simulating its arrival in real-time.

We next provide an overview of the traffic simulator in Section 4.1.1 and describe the composition of each record generated by the simulator in Section 4.1.2.

4.1.1 Traffic Simulation

The traffic simulator outputs a position report every 30 seconds for every vehicle v traveling on a given expressway in the system that identifies v's coordinates and speed. Every position report has a timestamp, which is the number of seconds elapsed since the start of the simulation. The simulator limits a vehicle's maximum speed to 100 MPH to ensure that each vehicle emits at least one position report from every segment it travels in. Likewise, every vehicle is guaranteed to average 40 MPH or less when entering (or exiting) an expressway and, therefore, it will emit at least one position report from an entrance (or exit) ramp for each vehicle trip.

Further, the simulator generates one *accident* in a random location on each expressway for every 20 minutes of simulation time. The input data also includes historical query requests issued by a vehicle with 1% probability every time it sends a position report. About 50% of the historical queries comprise account balances, 10% for total daily tolls on a given expressway, and the remaining 40% for travel time predictions.

4.1.2 Input Stream Data Format

The simulator output, comprising of the vehicles' position reports and historical queries is stored in the form of records in CSV format in a flat data file per expressway. Each record consists of an ordered set of 15 attributes: Type, Time, VID, Spd, XWay, Lane, Dir, Seg, Pos, QID, Sinit, Send, DOW, TOD, and Day. Type determines the type of the query (0: position report, 2: account balance, 3: daily expenditure, 4: travel time), Time (0 . . . 10799) denotes the simulation time in seconds, VID (0 . . . MAXINT) is an integer vehicle identifier, Spd (0 . . . 100) denotes a vehicle's speed in MPH, and XWay (0 . . . L-1), Lane (0 . . . 4), Dir (0,1), Seg (0.99), Pos (0 . . . 527999) describe a vehicle's global coordinates in terms of the expressway, lane, direction, segment, and position respectively. QID is an integer query identifier (Type=2, 3, 4), and Sinit, Send, DOW, and TOD denote the start and end segments, the day of the week (1 . . . 7), and the time of the day in minutes (1 . . . 1440), respectively, for travel time queries. Finally, Day (1 . . . 69) denotes the day of travel of the vehicle's trip (1 is yesterday, 69 is 10 weeks ago) for daily expenditure queries.

4.2 Query Requirements

The Linear Road benchmark requires an SDMS to process a fixed set of continuous and historical queries with hard real-time query response-latency and accuracy requirements. While the continuous queries in Linear Road demand real-time processing of streaming data, the historical queries require keeping track of potentially a large volume of past (10 weeks) and present data for answering queries.

We next describe the continuous queries in Section 4.2.1 and the historical queries in Section 4.2.2.

4.2.1 Continuous Queries

There are two types of continuous queries for computing toll notifications and accident notifications, respectively. We briefly discuss each of them below.

Toll Notification: The Linear Road benchmark requires an SDMS to calculate a toll every time a vehicle reports a position from a new segment in order to notify the toll amount back to that vehicle's driver. The toll at any given time is computed as a function of the number of vehicles, the average speed of vehicles on the segment, and the proximity of accidents. The toll amount for a given segment is notified to the vehicle on entering the segment but gets charged to the vehicle's account when it crosses over to the next segment. The complete history of toll assessments made to a vehicle's account needs to be tracked in order to answer historical queries. The response time for toll notification must be within 5 seconds between the timestamp of the position report and the time the toll is notified to the vehicle.

Accident Notification: An accident occurs if two vehicles report the same position on an expressway in four consecutive position reports. During the accident, the traffic in the affected segment moves at a reduced speed determined by the traffic spacing model. The duration of the accident ranges from 10 to 20 minutes, and it is deemed to be cleared after either of the involved vehicles emits a position report different from the accident location. Once an accident is detected, every vehicle that enters a segment in the vicinity (within five segments upstream) of the accident must be sent an accident notification within 5 seconds of its detection.

4.2.2 Historical Queries

The benchmark describes three historical queries: account balance, daily expenditure, and travel time estimation. The account balance query for a vehicle v at time t requires the sum of all tolls assessed to v's account as of t. This query must be answered within 5 seconds from the time of query emission and the returned balance must be accurate at some time in the last 60 seconds prior to t. The daily expenditure query requires the sum of all tolls charged to a vehicle's account for a given expressway on a given day within 10 seconds of issuance of the query. Finally, the travel time estimation query demands a prediction of time to travel between two given segments on an expressway on the basis of traffic statistics over the previous 10 weeks.

5. PROTOTYPE IMPLEMENTATION

In this section, we describe the design and implementation of the Linear Road benchmark on the SPC stream processing system. We first present the key design principles for achieving maximum query performance for Linear Road in Section 5.1. Many of these principles are in fact general guidelines which would also be applicable to the design and implementation of other streaming data applications. In Section 5.2, we describe the implementation of the Linear Road query network over SPC which employed these principles. Finally, in Section 5.3, we discuss the key differences between Aurora's centralized Linear Road implementation on a single node and the distributed Linear Road implementation over SPC.

5.1 Design Principles

We applied the following four design principles for implementing the Linear Road benchmark:

Modularity: Linear Road requires processing both continuous and historical queries which pose the challenge of complex data interdependencies between different queries. To bring the complexity under control, we apply modularization around processing steps to provide control and to share access to common data. In general for streaming data applications, modularity is particularly warranted for three primary reasons: (a) partitioning the application into a set of simpler processing elements helps identify and solve performance bottlenecks as we illustrate in Section 6, (b) decoupling the potentially large number of relations and conceptual processing loops in complex applications into an inter-connected set of independent processing elements allows manageability and ease of resource allocation, and (c) enabling a *pipelined model* of information flow in a distributed environment for scalability.

Data Aggregation: As the data flows between processing elements, the end-to-end system performance depends on the total bandwidth consumed in transferring data between them. To address the challenges of high-volume and rapidly-updating streaming data, we apply the principle of aggregating information from multiple stream updates wherever possible before transmitting it down the pipeline. This results in reduction of network bandwidth for data transmission as well as off-loading a part of the computational burden of downstream processing elements. Note that data aggregation is intended as a *summarizing mechanism* doing information reduction rather than just concatenating multiple stream data objects into a single one.

Network and Data Locality: In conjunction with the amount of data exchanged between processing elements, the query response time also depends on the efficiency of *information exchange*, i.e., the latency incurred in transmitting data. Therefore, the stream processing system must exploit (a) *network locality* to consume data close to where it is produced, and (b) *temporal data locality* to re-use data computation results as long as they are valid. In Linear Road's implementation over SPC, incorporating network and data proximity results in low query response latency as we will show in our experiments.

Flexible Programming Environment: The queries in Linear Road require rich processing functionality such as user-defined functions and custom query operators, not supported by a pre-defined set of relational operators alone provided by most existing systems. In our implementation, we rely on SPC's design where a *flexible* programming environment is available, enabling application developers to write their own query operators and not be constrained by the limitations of the underlying SDMS.

We next describe the Linear Road implementation on SPC in detail.

5.2 Linear Road in SPC

We implemented the Linear Road benchmark on SPC based on the design guidelines presented in Section 5.1. As we described in Section 3, SPC uses a *data graph* model of inter-connected processing elements for constructing queries over streaming data. The processing element function as well as its input-output information flows are specified by the application developer. As in the Aurora implementation of Linear Road, we did not implement travel time estimation queries and input for this type of query is ignored.

Figure 3 shows the query network infrastructure comprising 15 processing elements, decomposed functionally. The data file generated by the traffic simulator is delivered to the query network by the Generator PE. The Annotator PE acts both as a forwarding agent for each of the four query types, and as a replicating agent to feed data to different query sub-networks. The Sink PE provides a data sink for travel time estimation queries. The daily expenditure historical query has been implemented using the HistTolls PE that reads the input query request, performs a table lookup, and outputs its results directly. The account balance historical query is handled by the Tolls PE, which also provides support for toll notifications. The continuous queries process their input data, vehicle position reports, using three query

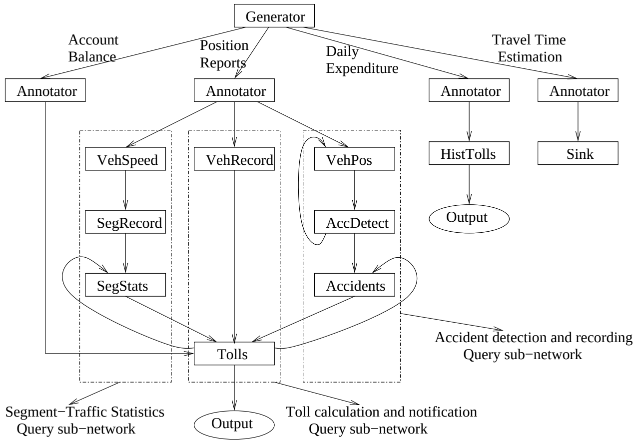

Figure 3: The Linear Road processing elements.

sub-networks: segment-traffic statistics, accident detection and recording, and toll calculation and notification.

As previously stated, all the three query sub-networks exhibit the principles of modularity, data aggregation, and network and data locality. Each of the query sub-networks is a separate module providing a different functionality. Each sub-network is in turn implemented using a set of processing elements. The processing elements within a query sub-network are connected in *series* to perform data aggregation and exploit data locality wherever applicable. Further, PEs (e.g., the AccDetect PE described later in Section 5.2.2) leverage SPC's flexible programming environment to implement custom query functions.

We describe each of the three query sub-networks and their associated PE modules in detail next.

5.2.1 Segment-Traffic Statistics

The segment-traffic statistics query sub-network aggregates vehicle position reports for maintaining the traffic statistics for every segment of every expressway with different time granularities. This functionality is implemented using three PE modules: (a) the VehSpeed PE calculates the running average speed of a vehicle according to all position reports it emits during a one minute interval and emits the updated average values downstream, (b) the SegRecord PE receives running averages of vehicle speed from the VehSpeed PE, maintaining the average speed and the number of vehicles for every segment of every expressway with 1 minute granularity, and (c) the SegStats PE computes the rolling average of the SegRecord PE output over the last 5 minutes and writes the statistics to a table, which is used in toll calculation.

5.2.2 Accident Detection and Recording

The accident detection query sub-network processes position reports for detecting and recording when vehicles are stopped, and checks whether the stopped vehicles are involved in an accident. This query sub-network comprises of the following processing elements: (a) the VehPos PE, which records and sends an update whenever a vehicle emits the same position in two consecutive position reports indicating *potential* accident occurrence, (b) the AccDetect PE, which uses a custom aggregate function to detect accidents by examining the last four position readings for each vehicle update received from the VehPos PE, and (c) the Accidents PE, which maintains a table for insertion (*accident detected*), deletion (*accident over*), and query (*accident exists?*) of accident events. For accident over notifications, the VehPos PE sends a notification whenever a vehicle involved in an accident emits an update with a new position different from the accident location. In our implementation, we also add a simple optimization of a feedback loop from the AccDetect PE to the VehPos PE, informing about vehicles in current accident locations to cull near-future updates of potential accident notifications for the same location.

5.2.3 Toll Calculation and Notification

The toll calculation query sub-network is responsible for calculating and emitting tolls for each vehicle as well as for charging tolls when appropriate. Note that this query sub-network is also the critical path since the benchmark performance is measured by the maximum load supported without violating the real-time query response requirements. The two PEs implementing toll functionality are: (a) the VehRecord PE processing each vehicle report to identify vehicles that have crossed into a new segment, and (b) the Tolls PE issuing a query to both the SegStats PE and the Accidents PE for every update sent by the VehRecord PE. If the query response from the Accidents PE indicates that the vehicle has entered a segment within 5 segments upstream of a recent accident then no toll is charged and the vehicle is notified of the accident location. Otherwise, if the query response from the SegStats PE indicates traffic congestion[1] on the segment in which the vehicle is traveling, a toll amount for the current segment is notified to the vehicle and recorded. At the same time, the toll reported for the segment being exited is assessed to the vehicle's account. If the vehicle exits at the exit ramp of a segment, no toll is charged. The continuous query results (toll notifications and accident results) as well as the historical query responses (account balance) are emitted to an output stream.

Note that in our implementation, the tables are stored as persistent data structures in memory. For permanent storage, they could be stored in a relational database to support off-line queries.

5.3 Discussion

Our implementation of the Linear Road application is fully distributed in contrast to the single-node Aurora implementation. We would like to highlight the following aspects of the implementation:

- In a distributed environment, the streaming data packets can get dropped, reordered, and duplicated. This departs from Aurora's model of intra-node communication receiving all data packets in order. In the context of Linear Road, the unreliable nature of the network can result in missing position reports and queries that were either dropped or duplicated in the network, or arrived with timestamp earlier than the current time.

[1]Linear Road charges a toll to a vehicle v traveling on a segment s during time t if all of the following three conditions hold: (1) s has no accidents within 5 segments downstream, (2) the average vehicle speed is less than 40, and (3) the number of vehicles is greater than 50 in the last minute preceding t.

- Aurora's *box-at-a-time* scheduler is limited by the synchronization primitives to control the order of processing in their query network [9]. In an asynchronous network, however, no constrains on the PE scheduler can be enforced; the order in which streaming data is processed is only governed by the data graph model in our implementation.

- Aurora uses a query-network model where multiple processing elements read and write to *shared* tables. In SPC implementation, the PE modules emit stream-data updates to perform read and write operations to information hosted at different PEs through a well-defined query interface. Specifically in Linear Road, the Tolls PE sends a query each to the Accidents PE and the SegStats PE, and performs toll calculation based on received responses.

- Aurora's streaming model provides *annotated streams* i.e., multiple stream types by annotating each data packet with attributes corresponding to that stream. In the SPC implementation, we provide support for annotated streams as well as for sending a raw packet type with stream attributes set in the payload. We provide an empirical comparison of these two approaches in Section 6.

- Aurora uses a workflow-like boxes-and-arrows model where application writer hard-wires the connection between different boxes. In SPC, each processing element *publishes* its output ports descriptions to the network with stream types so that other PEs can subscribe to the appropriate output port. Similarly, it specifies a flow specification expression, which describes the the particular stream characteristics it expects as input. Given these output port descriptions and flow specifications, the middleware takes care of automatically establishing the stream connections between producers and consumers.

- Finally, in a centralized system like Aurora, performance can be enhanced by allocating more resources to bottleneck PEs but at the risk of other PEs becoming resource-constrained. In contrast, a distributed stream processing environment such as SPC facilitates leveraging available network resources for maximum performance gains.

6. EXPERIMENTAL EVALUATION

In this section, we present the experimental results from running the SPC implementation of the Linear Road benchmark. Although Linear Road proposed L-rating[2] as the sole measure of scalability of a stream processing system [9], we perform a detailed evaluation study taking into consideration various factors that significantly impact overall system performance, for example: the number of expressways, the physical allocation of processing elements to cluster nodes, the buffer capacities, and the packet format for transporting streamed data. Further, we examine and present approaches to alleviate the performance bottlenecks we observed in the system.

[2]The maximum number of expressways, L, for which a system can respond to the specified set of continuous and historical queries while meeting their response time and accuracy requirements.

6.1 Methodology

We have implemented a fully functional prototype of the distributed Linear Road benchmark over SPC. Each of the PEs described in Section 5.2 is implemented as a stand-alone C++ module, which is compiled and linked with the SPC runtime library. The experimental testbed is an 85-node Linux cluster each with a dual-core hyper-threaded 3 GHz Xeon processor with 2 GB RAM running Linux kernel 2.6.5-SMP connected using a Gigabit switched Ethernet network.

The MITSIM traffic simulator is employed to generate L flat files each containing 3 hours of streaming input data from a single expressway in the benchmark. The data file contains tuples marked with timestamps reflecting the times of their generation as described in Section 4.1.2. A historical traffic generator produces a separate data file consisting of 10 weeks worth of historical data corresponding to the traffic simulator output. This historical data is read off-line by the HistTolls PE. The Generator PE delivers the streaming data input in real-time to the Linear Road query network as described in Section 5.2. The system response is recorded in an output data file along with the timestamp at which it was generated. The response time measurements were made at the application-level and include the queuing delay, the transmission latency, and the SPC processing overheads. At the end of the 3-hour data playback, the validation tool from the benchmark is used to check if the system output meets the correctness and query response time requirements. During the course of our study, we encountered some bugs in the traffic simulator and the validator which we would be submitting along with their fixes to the Linear Road project [2].

6.2 Input Data Distribution

A 3-hour, single expressway worth of input data consists of about 12 million position reports, about 67000 account balance, and 14000 daily expenditure query requests. As the simulation time progresses, the input data exhibits a monotonically increasing distribution with time, ranging from 15 records per second to 1700 records per second. The aim is to stress-test the SDMS by increasing input data rate with time to determine the load threshold until which SDMS can sustain meeting query requirements. Figure 4 shows the input data distribution for one expressway generated by the traffic simulator.

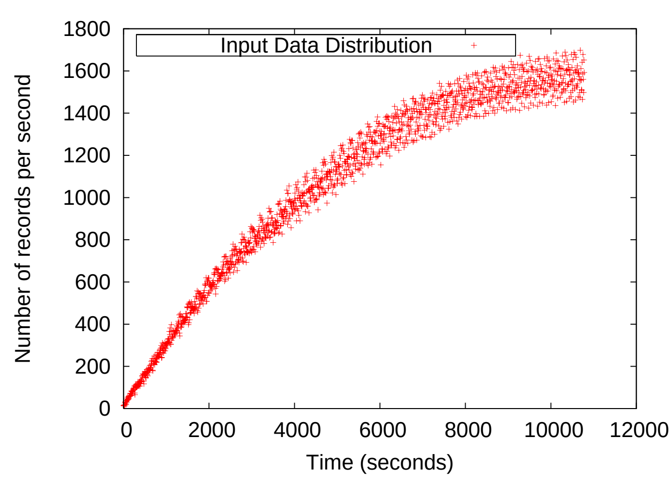

Figure 4: The input data distribution.

The corresponding stream data input received by the Tolls PE over the 3-hour run, is around 2.2 million tuples including the query responses received from the SegStats PE and the Accidents PE. In terms of the input load from the Generator PE, the toll calculation and notification query sub-network, comprising of the VehRecord PE and the Tolls PE (Figure 3), processes all the position report tuples as well, with a peak load of about 1700 SDOs per second per expressway.

We next report the *L*-rating for the benchmark running on our system.

6.3 Scale Factor

The SPC system achieves an *L*-rating of 2.5 for the Linear Road benchmark running on a single machine. Aurora also reported an *L*-factor of 2.5 on their system when run on a single node [9]. The Aurora source distribution available from their web site [3], however, does not provide the implementation of the Linear Road benchmark and we had to implement the application processing elements ourselves. Therefore, we do not report Linear Road performance of Aurora on our experimental testbed for a head-to-head comparison.

Here, we make one important distinction from Aurora in the evaluation of the query response time: instead of choosing an arbitrary fixed response time threshold, e.g., 5 seconds for toll notifications, we focus on the end-to-end query response latency as a function of the load to study the scalability characteristics of SPC. We still, however, present the *L*-rating of SPC for each case wherever it is applicable.

In all the graphs presented in this section, we plot the end-to-end query response latency (y1-axis) with increasing *load* (y2-axis) during the 3 hour period (x-axis). The load is measured in terms of the total number of tuples received at the Tolls PE and each point in the graph denotes the average query response time sampled every 100,000 tuples received. For all our experiments, we perform the measurements at the Tolls PE, since this PE assimilates results from all three query sub-networks to determine the toll and accident notifications, which are the responses to queries placed on the system. The Tolls PE also turns out to be a contention point on the critical path as we show in Section 6.4.

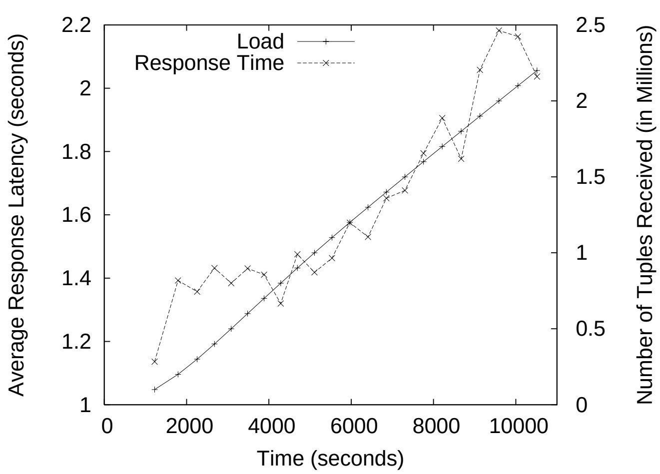

Figure 5: Latency-load measurement for 1 expressway running all the PEs on a single node.

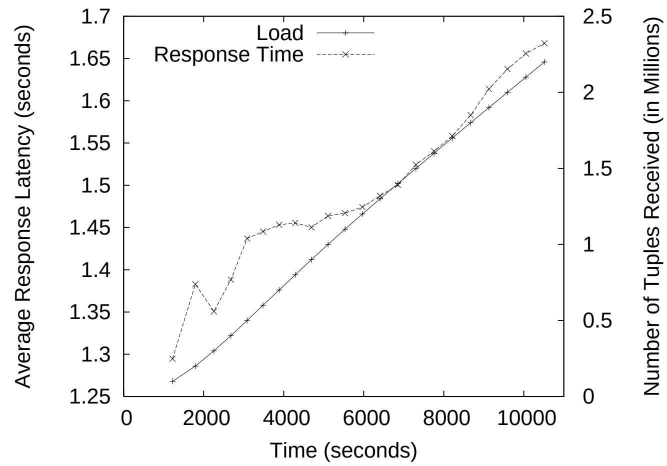

Figure 6: Latency-load measurement for 1 expressway running the Generator PE on one node and infrastructure PEs on another node.

6.4 Effect of the Number of Expressways

We first describe the performance of Linear Road over SPC with varying number of expressways. We initially ran the benchmark as a stand-alone application on a *single* node hosting all the PEs, as in Aurora. Figure 5 depicts the query response latency as a function of load for one expressway. We observe average and worst-case response times of 2.18 and 2.23 seconds respectively. These are well within the Linear Road latency bound of 5 seconds, indicating that SPC scales to one expressway.

Since the Generator PE does a lot of file I/O reading the raw simulation data and expends CPU cycles converting the raw data into an SDO representation, we decided to isolate it by running it on its own, exclusive node. All other *infrastructure* PEs are assigned to a different node. Figure 6 shows the corresponding plot. As expected, the benchmark performance improves as the average and maximum average response time drops to 1.67 seconds and 1.79 seconds, respectively.

Figure 7 depicts the latency and load graph for two expressways. We run two instances of the Generator PEs, each assigned to a different node, and all the infrastructure PEs on a third machine. The experiment shows that the query response time increases linearly until a load of 2.5 million tuples corresponding to roughly 6000 seconds (1.67 hours) of elapsed time in the input data. After that, the response time flattens out demonstrating that SPC scales to two expressways.

We next increased the number of expressways to 2.5 by selecting the position reports for the west-bound direction for one expressway. This configuration shows a response time of roughly 6 seconds for 2.5 expressways. To understand the system bottlenecks in achieving higher scalability for this scenario, we performed detailed system analysis, which revealed that: (a) the Tolls PE, responsible for generating query responses, was overloaded and became a contention point, and (b) the induced load always kept the network buffer (which as a default has 1024 slots) full at the node hosting the infrastructure PEs. Therefore, the tuples were queued at the Generator PEs resulting in each tuple suffering a large latency. Increasing the buffer size to 10000 (Fig-

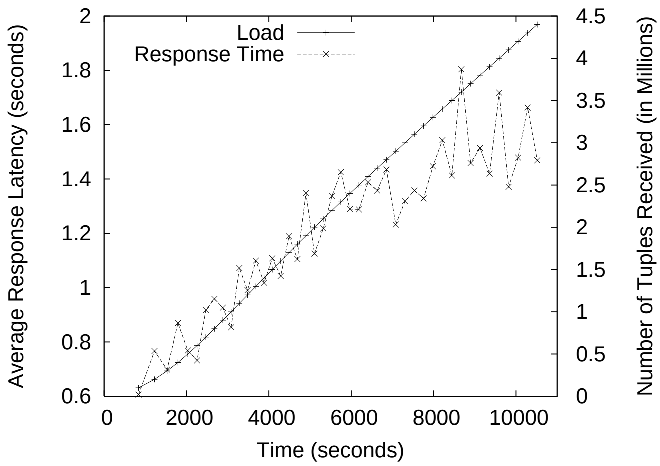

Figure 7: Latency-load measurement for 2 expressways running 2 Generator PEs on two different nodes and infrastructure PEs on another node.

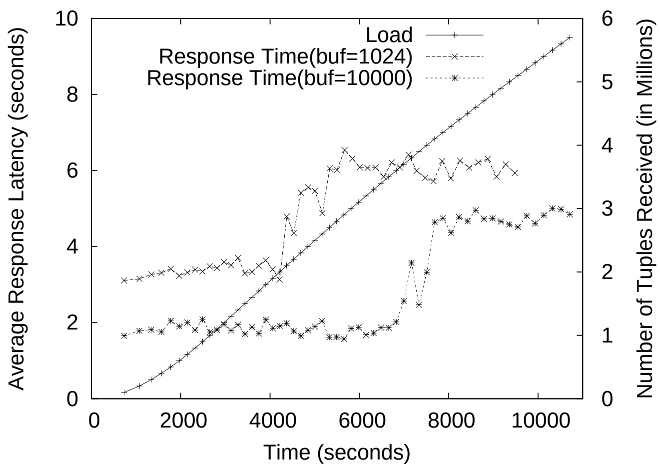

Figure 8: Latency-load measurement for 2.5 expressways running 3 Generator PEs on three different nodes and infrastructure PEs on another node.

ure 8), however, masks both the network and the remote queuing latency, and burstiness until a higher load. Further since more buffered data is locally available for processing in this configuration, the system is able to handle the load for 2.5 expressways with average and worst-case response time of 4.71 seconds and 4.87 seconds, respectively.

SPC, however, does not meet the benchmark requirements for 3 expressways running infrastructure PEs on a single node as shown in Figure 9. We hit a throughput bottleneck somewhere between 2 and 2.5 expressways since we observe the jump in the average response latency due to the queuing delay for 2.5 expressways but not for 2 expressways. This amounts to a processing load of roughly 4300 SDOs per second at the Tolls PE. From these experiments (Figures 8, 9), we observe a jump in the latency experienced by the tuples at different points depending on the buffer size. In particular, when the buffer size gets larger, the corresponding jump in the query response latency occurs at a higher load threshold. This is due to the increased queuing delay

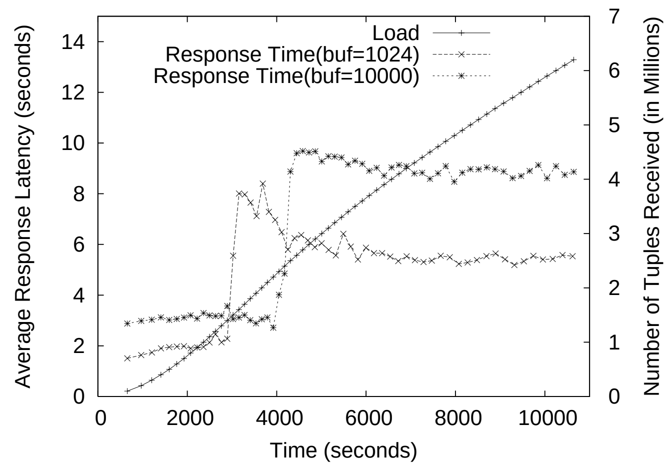

Figure 9: Latency-load measurement for 3 expressways running 3 Generator PEs on three different nodes and infrastructure PEs on another node.

experienced by the packets as the buffers start building up with increasing offered load. We analyze the effect of the buffer capacity on the benchmark performance in detail in Section 6.7.

6.5 Analyzing Bottleneck PEs

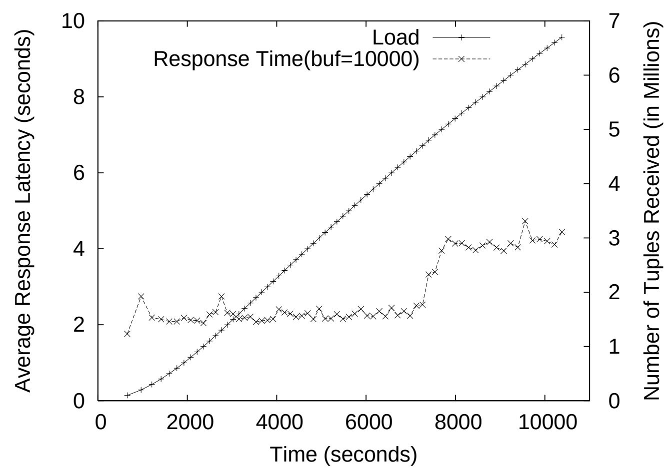

Figure 10: Latency-load measurement for 3 expressways running 3 Generator PEs and the Tolls PE on four different nodes and remaining infrastructure PEs on another node.

In the previous configuration, we identified the Tolls PE as one of the system bottlenecks. By manually allocating more resources to the Tolls PE by running it on a separate node, SPC is able to meet the query requirements for 3 expressways with average query response time of 4.44 seconds as seen in Figure 10. Further, 97% of the query response times are below 5 seconds threshold, 98.2% are under 6 seconds, and 99.3% are under 7 seconds.

During our experiments, we observed that when the CPU utilization reaches 70% for the infrastructure PEs, the SPC's Data Fabric daemon CPU utilization reaches 100%. This is due to the fact that the per-packet overhead in our system is high. Although Linear Road tuples are rather small, the offered load in terms of tuples-per-second is high and this manifests as a large per-packet CPU cost. The CPU cost includes overheads such as memory allocation, marshaling/demarshaling of each packet, TCP system-call and transmission overheads among others. Providing low latency in inter-PE communication is a key challenge and one of our main focus areas at present. We are exploring techniques such as batching of packets and using IP multicast to achieve better performance.

Another insight we got is that higher scalability can be achieved by assigning infrastructure PEs to nodes carefully based on their processing requirements. Unfortunately, that cannot be easily achieved by *hand-placing* PEs on nodes for all possible application configurations. The resource manager for SPC is currently under development that will perform automatic dynamic resource allocation depending on the current load distribution and query response requirements. In the next section, we analyze the effect of PE to node assignment policy.

6.6 PE Placement Policy

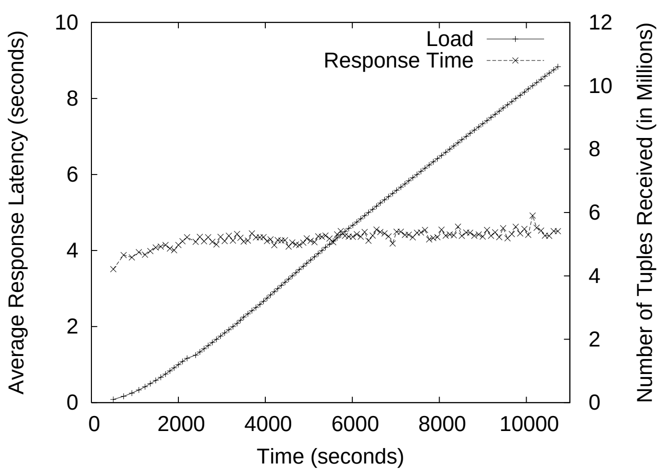

Figure 11: Latency-load measurement for 5 expressways running each PE on a different node.

We previously described one simple policy of allocating all the infrastructure PEs on a single node in Section 6.4. We now examine the effect of employing the other *extreme* of PE-to-node allocation where we distribute one infrastructure PE per node. Figure 11 shows that SPC successfully scales up to 5 expressways with a worst-case query response time of 4.85 seconds. Correspondingly, 99.3% of the query response times are below 5 seconds threshold, 99.6% are under 6 seconds, and 99.8% are under 7 seconds. We also observed that SPC's query response latency increases to 6.9 seconds for 6 expressways under this allocation policy.

In a resource-limited environment, however, the one-node-per-PE allocation policy might not be feasible for all application scenarios. Therefore, we further emphasize the need for a resource manager to do optimal allocation of (possibly limited) resources while still meeting the application requirements.

6.7 Tuning Buffer Capacity

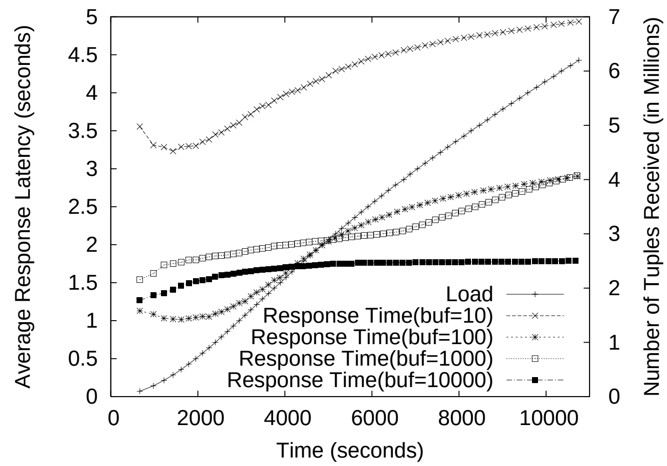

Figure 12: Effect of the network buffer size on query response time for 3 expressways running each PE on a different node.

As discussed in Section 6.4, the network buffer capacity has a significant impact on the query performance in Linear Road. Figure 12 shows the effect of the network buffer for four different sizes (10, 100, 1000, 10000) on query response time for 3 expressways with each PE running on a different node. The experiment illustrates that as the buffer size increases from 10 to 10000, the average query response latency decreases from roughly 4.8 seconds to 2 seconds.

Therefore, a large buffer size can effectively mask the network latency and the burstiness in input streaming traffic[3]. Note, however, that the query response latency in the case of buffer of size 100 is initially less than that for buffer of size 10000 up to about 4000 seconds into the benchmark run. This is because the receiver-side queuing delay dominates network latency when the volume of streaming traffic is small during the initial time period. But as the time progresses, the offered load from multiple generators exceeds the capacity of the Tolls PE thereby causing the buffers to fill up and the sender-side queuing latency to dominate.

Although the above analysis was done in the context of the Linear Road application, we show in an extended technical report [12] the mathematical analysis of how the network buffer capacities and the relative execution speeds of producer and consumer processes, affect the end-to-end response latency in a general streaming data application.

6.8 (De)Marshaling Costs

SPC employs a mechanism for manipulating stream data objects attributes, which relies on a global type system, and provides a convenient and general purpose mechanism for PE writers to annotate objects with typed, structured attributes in addition to an opaque payload. This allows for streams to be shared across applications. The Data Fabric uses this structured representation to validate if the attributes conform to what the PE declared for its ports and to perform the SDO-filter matching. The Data Fabric also

[3]The Generator PE tries to send all tuples with the same timestamp every second at the maximum possible transmission rate to meet real-time requirements.

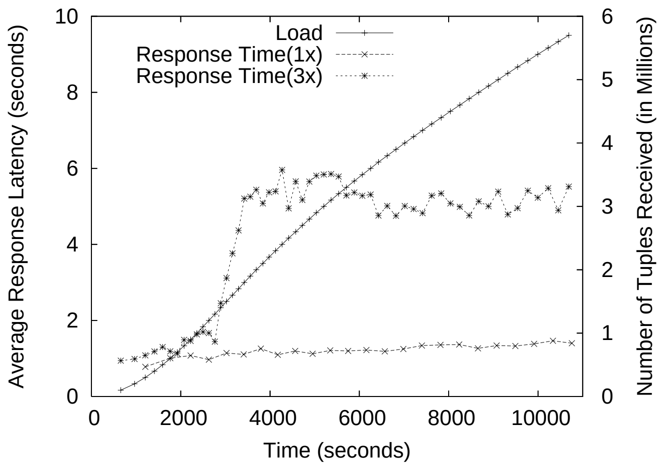

Figure 13: Effect of the raw stream data format on query response time varying the number of expressways running all infrastructure PEs on the same node.

provides transparent marshaling and de-marshaling of these structured attributes to and from an internal representation as stream data objects are transferred from one processing element to the next. In Section 6.5, we hypothesized that one of the factors in the high CPU cost was this marshaling and demarshaling cost associated with each packet and tested that in this experiment.

We wanted to isolate this (de)marshaling cost from the SDO transmission cost. We therefore experimented with an alternative approach where SDOs carried the tuple as an opaque payload. The Data Fabric transmits the opaque payload and we left the interpretation (marshaling/demarshaling) of the payload to the PEs themselves. This approach puts an additional burden on PE writers and prevents reuse of streams across applications. Nevertheless, as seen in Figure 13, we observe that sending raw SDOs where PEs are in charge of managing the data directly has a lower overhead compared to the default approach, where the Data Fabric is responsible for those services. For one expressway, the raw stream data format has an average query response time of less than 1 second compared to 1.67 seconds for annotated streams (Figure 6). The corresponding mean query response time for 3 expressways is 5.52 seconds for raw stream data format compared to roughly 6 seconds for annotated streams (Figure 9). We are currently working on optimizing the SDO representation to reduce these costs.

6.9 Lessons Learned

As described earlier, a large-scale distributed stream processing system such as the SPC has to host processing elements of widely varying characteristics – I/O bound, CPU-bound, and memory-bound. To address these challenges, and based on lessons learned from the experiments, we are adding new features to SPC that will expose the different knobs and allow for the appropriate resource allocation decisions to be made for maximizing the application performance. Some of the key parameters that affect the application performance came out clearly from scaling the Linear Road benchmark—end-to-end latency which includes queuing delay in the system as well as the PE-to-PE transmission latency, data-loss due to PE-processing rate mismatches, attribute processing overhead, and buffer sizes in the infrastructure to address burstiness in streams. We are currently working on various schemes to address each of the issues, some of which are indicated below.

- **Explicit rate feedback**: Stream processing systems need to keep data constantly moving through the system since data sources keep pushing "live" data. Due to the varying processing times on packets in different stages of the processing pipeline, rate mismatches and hence congestion may occur. Although the resource manager addresses long-term rate mismatches, the system has to handle local, short-term rate mismatches. This is achieved by sending explicit rate update messages upstream such that the Data Fabric component and eventually the producer PE can perform the appropriate load shedding operations. See [7] for details on the control algorithm used.

- **Co-location of PEs**: PEs can be built as dynamically loadable modules which allows multiple PEs to be co-located in the same execution container in threads, thereby providing resource savings.

- **Multiple transport options**: When PEs are loaded in the same process space, they can transport data using pointer passing; when in different process spaces, they can transport data using shared memory and when in different boxes, use sockets. For all experiments in this paper, the last scheme was adopted.

- **Data distribution using flow specifications**: The SPC allows PEs to have flow specifications which can describe the characteristics of the packets that a PE wishes to receive, using a pub-sub model. By using the flow specifications appropriately, applications can easily split the data that belongs to a stream based on the values of attributes, thereby providing scalability.

  In the context of implementing a scalable Linear Road application, the Tolls PE can be replicated to multiple nodes by splitting input data streams based on either vehicles' VIDs or expressways.

- **Tunable buffer sizes**: We realized that buffer sizes play an important role in determining application performance and that it is useful to expose this parameter to the PE. Hence, PEs can request to set the size of their input buffer. The system allows or disallows it based on the allowed resource budget for the PEs.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented the performance evaluation of the SPC stream processing system using an implementation of the Linear Road benchmark. We described the design and implementation of the Linear Road benchmark using the programming model provided by the SPC infrastructure, and reported the results from performing various experiments on a distributed prototype running on a Linux cluster. Most of the performance results on stream processing systems presented till date have been either on a single node or on very small configurations. This paper deviates from the others in presenting an analysis of performance bottlenecks and characteristics of a large-scale distributed stream processing system.

Using the Linear Road application, we have demonstrated some of the overheads in SPC and have also identified the features in the architecture that benefit applications the most. System factors most affecting the application performance are the inter-PE data transfer latency for which we are building various zero-copy data transfer schemes; attribute processing overhead in small packets for which again, we are exploring schemes which can adapt to the expected load characteristics of streams; resource allocation and PE placement decisions – although the current prototype has a rudimentary resource-allocation scheme of determining placement based on application-specified hints, we are working on a more sophisticated resource manager for dynamic resource allocation decisions. Some of the features that aided in the application design and performance were the ease with which the Linear Road infrastructure could be scaled using the dynamic flow specification feature, the ability to change the buffer size at the PE's port helped address the bursty nature of the streams.

Finally, through extensive experimental evaluation, we examined how SPC can be used and tuned to enable a very efficient implementation of the Linear Road application in a distributed environment.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments that helped us improve the presentation of this paper.

SPC is part of a larger project called System S, which is being developed as a collaborative effort among many groups within IBM Research. The authors wish to thank Nagui Halim, the principal investigator for System S, and other System S team members for many formative discussions.

9. REFERENCES

[1] http://mit.edu/its/mitsimlab.html.

[2] http://www.cs.brandeis.edu/~linearroad.

[3] http://www.cs.brown.edu/research/aurora/main.html.

[4] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), August 2003.

[5] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the 2005 Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, 2005.

[6] L. Amini, H. Andrade, F. Eskesen, R. King, Y. Park, P. Selo, and C. Venkatramani. The Stream Processing Core. Technical Report RSC 23798 (submitted for publication), IBM T. J. Watson Research Center, November 2005.

[7] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive Control of Extreme-Scale Stream Processing Systems. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS 2006)*, Lisboa, Portugal, July 2006.

[8] A. Arasu, B. Babcock, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford Stream Data Manager (Demonstration Description). In *Proceedings of the 2003 ACM International Conference on Management of Data (SIGMOD 2003)*, San Diego, CA, June 2003.

[9] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A stream data management benchmark. In *Proceedings of the 30th International Conference on Very Large Data Bases Conference (VLDB 2004)*, Toronto, Canada, 2004.

[10] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11), October 2001.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, , and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the 2003 Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, 2003.

[12] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. Technical Report TR-06-18, Department of Computer Sciences, University of Texas at Austin, March 2006.

[13] K. Kuo, R. Rabbah, and S. Amarasinghe. A productive programming environment for stream computing. In *Proceedings of the 2nd Second Workshop on Productivity and Performance in High-End Computing*, San Francisco, CA, February 2005.

[14] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM International Conference on Management of Data (SIGMOD 2002)*, Madison, WI, June 2002.

[15] C. Pu, K. Schwan, and J. Walpole. Infosphere project: System support for information flow applications. *ACM SIGMOD Record*, 30(1), March 2001.

[16] G. Swint, G. Jung, and C. Pu. Event-based QoS for a distributed continual query system. In *Proceedings of the 2005 IEEE International Conference on Information Reuse and Integration (IRI 2005)*, Las Vegas, NV, August 2005.

[17] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 2002 International Conference on Compiler Construction (ICCC 2002)*, Grenoble, France, April 2002.

[18] S. Zdonik, M. Stonebraker, M. Cherniak, U. Cetintemel, M. Balazinska, and H. Balakrishnan. The Aurora and Medusa projects. *Bulletin of the IEEE Technical Committee on Data Engineering*, March 2003.