Operator Scheduling in a Data Stream Manager

Don Carney[†], Uğur Çetintemel[†], Alex Rasin[†], Stan Zdonik[†]
Mitch Cherniack[§], Mike Stonebraker[±]

[†]Department of Computer Science, Brown University
[§]Department of Computer Science, Brandeis University
[±]Laboratory for Computer Science & Department of EECS, Massachusetts Institute of Technology

Abstract

Many stream-based applications have sophisticated data processing requirements and real-time performance expectations that need to be met under asynchronous, time-varying data streams. In order to address these challenges, we propose novel operator scheduling approaches that specify (1) which operators to schedule (2) in which order to schedule the operators, and (3) how many tuples to process at each execution; and study them in the context of the Aurora data stream manager.

We argue and provide experimental evidence that a fine-grained scheduling approach in combination with various scheduling techniques (such as batching of operators and tuples) can significantly improve the efficiency by reducing various system overheads. We also discuss application-aware extensions that address Quality of Service (QoS) issues by making scheduling decisions according to tuple processing delays and per-application QoS specifications. Finally, we present prototype-based experimental results that characterize the efficiency and effectiveness of our approaches under various stream workloads and processing scenarios.

1 Introduction

Applications that deal with potentially unbounded, continuous streams of data are becoming increasingly popular due to a confluence of advances in real-time, wide-area data dissemination technologies and the emergence of small-scale computing devices (such as GPSs and micro-sensors) that continually emit data obtained from their physical environment. Example applications include sensor networks, position tracking, fabrication line management, network management, and financial portfolio management. All these applications require timely processing of large volumes of continuous, potentially rapid and asynchronous data streams. Hereafter, we refer to such applications as *stream-based* applications.

We have designed a system called Aurora [6], a data stream manager that addresses the performance and processing requirements of stream-based applications. Aurora supports multiple concurrent continuous queries, each of which produces results to one or more stream-based applications. Each continuous query consists of a directed acyclic graph of a well-defined set of operators (or *boxes* in Aurora terminology). Applications define their service expectations using Quality-of-Service (QoS) specifications, which guide Aurora's resource allocation decisions. We provide an overview of Aurora in Section 2.

A key component of Aurora, or any data stream management system for that matter, is the scheduler that controls processor allocation. The scheduler is responsible for multiplexing the processor usage to multiple continuous queries according to application-level performance or fairness goals. Simple processor allocation can be achieved by assigning a thread per operator or per query. This technique does not scale since no system that we are aware of can adequately deal with a very large number of threads. More importantly, for stream processing purposes, any such approach would abdicate the details of scheduling to the operating system. This paper shows that having finer-grained control of processor allocation can make a significant difference to overall system performance by cutting down various system overheads associated with continuous query execution.

Figure 1 depicts the cost components of a continuous query execution (where the individual operators are scheduled using a random and a round-robin scheduling policy). The query used for this experiment models an intelligent data routing application (provided by MITRE Corp.), where data gathered by a next generation reconnaissance aircraft are routed to appropriate ground stations. The query basically consists of 40 stream-based operators, most of which are simple filter, project, and union operators (for our purposes the exact form of the query is not important). The figure reveals that the actual time spent for processing is smaller than 5% of the overall execution time in both cases. The remainder consists of three basic overheads incurred by the

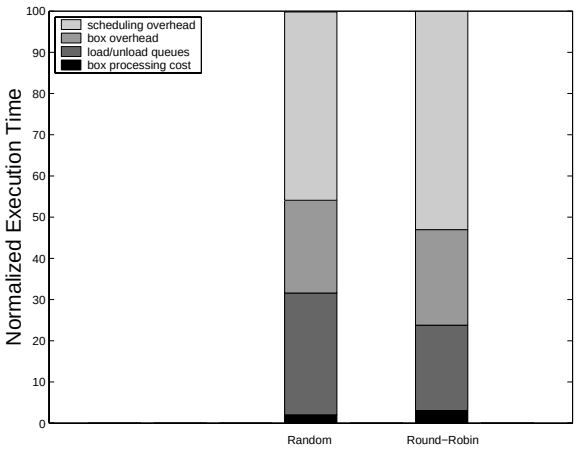

Figure 1: Where the time goes in query execution

scheduler, the buffer manager (i.e., loading/unloading tuple queues), and the worker thread that executes the operator (we discuss these components in more detail in the remainder of the paper). Note that this particular query consisted mostly of low cost operators; in general one might expect that queries include operators with higher processing costs (such as aggregates, joins, and user-defined functions). In such cases, processing costs will clearly become more pronounced. On the other hand, we believe that the result is representative in that overheads will always be non-negligible and will frequently dominate the overall execution time.

Motivated by this key observation, this paper studies operator scheduling for stream processing systems with the goal of reducing system overheads. We propose a set of novel scheduling techniques that reduce various system overheads by batching (of operators and tuples), incremental state tracking, approximation, and pre-computation.

In particular, we describe the design and implementation of the Aurora scheduler, which performs the following tasks:

1. *Dynamic scheduling-plan construction*: The scheduler develops a scheduling plan that specifies, at each scheduling point, (1) which boxes to schedule, (2) in which order to schedule the boxes, and (3) how many tuples to process at each box execution.
2. *Latency-based priority assignment*: The Aurora scheduler strives to maximize the overall QoS delivered to the client applications. At a high level, our scheduling decisions are based on a novel box priority assignment technique that uses the latencies of queued tuples and application-specific QoS information. For improved scalability, we also use an approximation technique, based on bucketing and pre-computation, which trades scheduling quality and scheduling overhead.

We also evaluate and experimentally compare these algorithms on our Aurora prototype under various stream processing and workload scenarios. Through the implementation of our techniques on the prototype rather than a simulator, we were better able to understand the actual costs associated with system overhead.

The rest of the paper is organized as follows: Section 2 provides an overview of the Aurora data stream manager. Section 3 describes the state-based execution model used by Aurora. Section 4 discusses in detail Aurora's scheduling algorithms. Section 5 discusses our prototype-based experimental study that provides quantitative evidence regarding the efficiency and effectiveness of Aurora's scheduling algorithms. Section 6 extends our basic approaches to address QoS, describing queue-based priority assignment and an approximation technique for improving the scalability of the system. Section 7 describes related work, and Section 8 concludes the paper.

2 Aurora Overview

2.1 Basic Model

Aurora data is assumed to come from a variety of data sources such as computer programs that generate values (at regular or irregular intervals) or hardware *sensors*. We will use the term *data source* for either case. In addition, a data *stream* is the term we will use for the collection of data values that are presented by a data source. Each data source is assumed to have a unique source identifier and

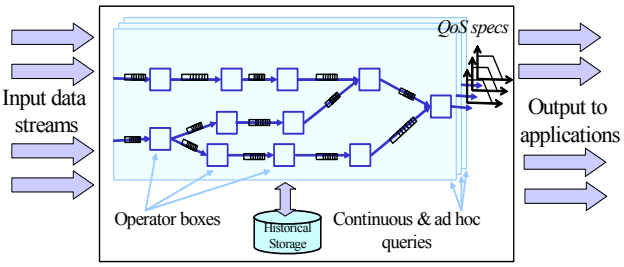

Figure 2: Aurora system model

Aurora timestamps every incoming tuple to monitor the QoS being provided.

The basic job of Aurora is to process incoming streams in the way defined by an *application administrator*. Aurora is fundamentally a data-flow system and uses the popular *boxes and arrows* paradigm found in most process flow and workflow systems. Hence, tuples flow through a loop-free, directed graph of processing operations (a.k.a. *boxes*). Ultimately, output streams are presented to *applications*, which must be programmed to deal with output tuples that are generated asynchronously. Aurora can also maintain historical storage, primarily in order to support ad-hoc queries.

Tuples generated by data sources arrive at the input and are queued for processing. The *scheduler* selects boxes with waiting tuples and executes them on one or more of their input tuples. The output tuples of a box are queued at the input of the next box in sequence. In this

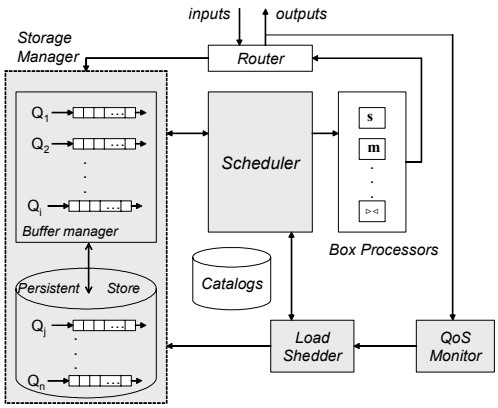

Figure 3: Aurora run-time engine

way, tuples make their way from the inputs to the outputs. Each output is associated with one or more QoS specifications, which define the utility of stale or imprecise results to the corresponding application. Figure 2 illustrates this high-level system model.

The primary performance-related QoS is based on the notion of the *latency* (i.e., delay) of output tuples—output tuples should be produced in a timely fashion, otherwise, QoS will degrade as latencies get longer. In this paper, we will only deal with latency-based QoS graphs; for a discussion of other types of QoS graphs and how they are utilized, please refer to [1, 6]. Aurora assumes that all QoS graphs are normalized, and are thus quantitatively comparable. Aurora further assumes that the QoS requirements are *feasible*; i.e., unless the system is overloaded, an idealized scheduler will be able to deliver maximum possible QoS for each individual output.

Aurora contains built-in support for eleven primitive operations for expressing its stream processing requirements. Some operators just transform individual items in the stream to other items, while other operators, such as the aggregate operators (e.g., moving average), apply a function across a window of values in a stream. A description of the operators is outside the scope of this paper and can be found in [1, 6].

2.2 Architecture

Figure 3 illustrates the architecture of the basic Aurora run-time engine. Here, inputs from data sources and outputs from boxes are fed to the router, which forwards them either to external applications or to the storage manager to be placed on the proper queues. The storage manager is responsible for maintaining the box queues and managing the buffer, properly making tuple queues available for read and write by operators. Conceptually, the scheduler picks a box for execution, ascertains what processing is required and *how many tuples to process* from the corresponding queue, and passes a pointer to the box description (together with a pointer to the box state) to the multi-threaded box processor. The box processor executes the appropriate operation and then forwards the output tuples to the router. The scheduler then ascertains the next processing step and the cycle repeats.

The QoS monitor continually monitors system performance and activates the load shedder when it detects an overload situation and poor system performance. The load shedder then sheds load till the performance of the system reaches an acceptable level. The catalog contains information regarding the network topology, inputs, outputs, QoS information, and relevant statistics (e.g., selectivity, average box processing costs), and is essentially used by all components.

3 Basic Execution Model

The traditional model for structuring database servers is *thread-based execution*, which is supported widely by traditional programming languages and environments. The basic approach is to assign a thread to each query or operator. The operating system (OS) is responsible for providing a virtual machine for each thread and overlapping computation and I/O by switching among the threads. The primary advantage of this model is that it is very easy to program, as OS does most of the job. On the other hand, especially when the number of threads is large, the thread-based execution model incurs significant overhead due to cache misses, lock contention, and switching. More importantly for our purposes, the OS handles the scheduling and does not allow the overlaying software to have fine-grained control over resource management.

Instead, Aurora uses a *state-based execution* model. In this model, there is a single scheduler thread that tracks system state and maintains the execution queue. The execution queue is shared among a small number of worker threads responsible for executing the queue entries (as we discuss below, each entry is a sequence of boxes). This state-based model avoids the mentioned limitations of the thread-based model, enabling fine-grained allocation of resources according to application-specific targets (such as QoS). Furthermore, this model also enables effective *batching* of operators and tuples, which we show has drastic effects on the performance of the system as it cuts down the scheduling and box execution overheads.

An important challenge with the state-based model is that of designing an intelligent but low-overhead scheduler. In this model, the scheduler becomes solely responsible for keeping track of system context and deciding when and for how long to execute each operator. In order to meet application-specific QoS requirements, the scheduler should carefully multiplex the processing of multiple continuous queries. At the same time, the scheduler should try to minimize the system overheads, time not spent doing "useful work" (i.e., processing), with no or acceptable degradation in its effectiveness.

4 Two-Level Scheduling

Aurora uses a two-level scheduling approach to address the execution of multiple simultaneous queries. The first-level decision involves determining *which* continuous (sub-)query to process. This is followed by a second-level decision that then decides *how* precisely the selected query should be processed. This former decision entails dynamically assigning priorities to operators at run-time, according to QoS specifications, whereas the latter decision entails choosing the order in which the component operators will be executed. The outcome of these decisions is a sequence of operators, referred to as a *scheduling plan*, to be executed one after another. The scheduling plan is inserted into the execution queue to be later picked up and executed by one of the worker threads.

In order to reduce the scheduling and operator overheads, Aurora heavily relies on batching (i.e., grouping) during scheduling. We developed and implemented algorithms that batch both operators and tuples. In both cases, we observed significant performance gains over the non-batching counterparts. We now describe in detail our batching approaches for constructing scheduling plans.

4.1 Operator Batching - Superbox Processing

A superbox is a sequence of boxes that is scheduled as an atomic group. Superboxes are useful for decreasing the overall execution costs and improving scalability as (1) they significantly reduce the scheduling overhead by scheduling multiple boxes as a single unit; (2) they eliminate the need to access the storage manager for each individual box execution by having the storage manager allocate memory for the entire superbox at once[1].

Conceptually, a superbox can be an arbitrary connected subset of the Aurora network. However, we do constrain the form of superboxes such that each is always a tree of boxes rooted at an *output box* (i.e., a box whose output tuples are forwarded to an external application). The reasons that underlie this constraint are twofold. First, only the tuples that are produced by an output box provide any utility value to the system. Second, even though allowing arbitrary superboxes will provide the most flexibility and increase opportunities for optimization, it will also make the search space for superbox selection intractable for large Aurora networks.

The following subsections discuss the two key issues to deal with when scheduling superboxes, namely *superbox selection* and *superbox traversal*.

[1] Another benefit of superbox scheduling, which we do not address in this paper, is that it improves effective buffer utilization by consuming as many tuples as possible once the tuples are in memory. This potentially reduces the number of times each tuple is swapped between memory and disk.

4.1.1 Superbox Selection

The first-level scheduling issue involves determining superboxes to schedule. Fundamentally, there are two different approaches to superbox selection: *static* and *dynamic*. Static approaches identify potential superboxes statically before run-time, whereas the dynamic approaches identify useful superboxes at run-time. We implemented two representative superbox selection algorithms in Aurora.

- *Application-spanner* (*AS*). This approach statically defines one superbox for each query tree. As a result, the number of superboxes is always equal to the number of continuous queries (or applications) in the Aurora network. Figure 4 illustrates a simple query tree that consists of six boxes (the tree is rooted at box b_1).
- *Top-k-spanner* (*TKS*). This algorithm identifies, at run-time, the tree that is rooted at an output box and that spans the *top k* highest priority boxes for a given application. The priorities are assigned to boxes based on the latencies of tuples on each box's input queues and on application-specific QoS specifications (Section 5). A high priority box's input tuples need to be processed as soon as possible if the system were to gather any utility. Consider again the query tree in Figure 4. Assuming that b_2 and b_6 are the top two highest priority boxes, the top-2-spanner of the query tree includes the shaded boxes. Note that TKS also includes all the intermediate boxes that lie on the path between any of the top k boxes and the root box. TKS is equivalent to an application tree when k is equivalent to the number of boxes in the application tree.

4.1.2 Superbox Traversal

Once it is determined which boxes need to be executed, a second-level decision process needs to specify the ordering of these boxes in the scheduling plan. This is accomplished by *traversing* the superbox. The goal of superbox traversal is to process all the tuples that are queued within the superbox (i.e., those tuples that reside on the input queues of all boxes that constitute the superbox).

We describe three traversal algorithms that primarily differ in the performance-related metric for which they strive to optimize: *throughput*, *latency*, and *memory requirements*.

Min-Cost (MC). The first traversal technique attempts to optimize per-output-tuple processing costs (or *average throughput*) by minimizing the *number of box calls per output tuple*. This is accomplished by traversing the superbox in *post order*, where a box is scheduled for execution only after all the boxes in its sub-tree are scheduled. Notice that a superbox execution based on an MC traversal consumes all tuples while executing each box only once.

Consider the query tree shown in Figure 4 and assume for illustration purposes that a superbox that covers the

entire tree is defined. Assume that each box has a processing cost per tuple of p, a box call overhead of o, and a selectivity equal to one. Furthermore, assume that each box has an input queue that consists of a single tuple. An *MC traversal* of the superbox consists of executing each box only once:

$$b_4 \rightarrow b_5 \rightarrow b_3 \rightarrow b_2 \rightarrow b_6 \rightarrow b_1$$

This traversal consists of six box calls. A simple back-of-the-envelope calculation tells us that the total execution cost of the superbox (i.e., the time it takes to produce all the output tuples) is $15p + 6o$ and the average output tuple latency is $12.5p + o$.

Min-Latency (ML). Average latency of the output tuples can be reduced by producing initial output tuples as fast as possible. In order to accomplish this, we define a cost metric for each box b, referred to as the output cost of b, *output_cost*(b). This value is an estimate of the latency incurred in producing one output tuple using the tuples at b's queue and processing them downstream all the way to the corresponding output.

This value can be computed using the following formulas:

$$o\_sel(b) = \prod_{k \in D(b)} sel(k)$$

$$output\_cost(b) = \sum_{k \in D(b)} cost(k) / o\_sel(k)$$

where $D(b)$ is, as before, the set of boxes *downstream* from b including b, and $sel(b)$ is the estimated selectivity of b. In Figure 4, $D(b_3)$ is $b_3 \rightarrow b_2 \rightarrow b_1$, and $D(b_1)$ is b_1. The output selectivity of a box b, $o\_sel(b)$, estimates how many tuples should be processed from b's queue to produce one tuple at the output.

To come up with the traversal order, the boxes are first sorted in increasing order of their output costs. Starting from an empty traversal sequence and box b with the smallest such value, we can then construct the sequence by appending $D(b)$ to the existing sequence.

An *ML traversal* of the superbox of Figure 4 described above is:

$b_1 \rightarrow b_2 \rightarrow b_1 \rightarrow b_6 \rightarrow b_1 \rightarrow b_4 \rightarrow b_2 \rightarrow b_1 \rightarrow b_3 \rightarrow b_2 \rightarrow b_1 \rightarrow b_5 \rightarrow b_3 \rightarrow b_2 \rightarrow b_1$

The ML traversal incurs nine extra box calls over an MC traversal (which only incurs six box calls). In this case, the total execution cost is $15p + 15o$, and the average latency is $7.17p + 7.17o$.

Notice that MC always achieves a lower total execution time than ML (in this case by $6o$). This is an important improvement especially when the system is under CPU stress, as it effectively increases the throughput of the system. ML may achieve lower latency depending on the ratio of box processing costs to box overheads. In this example, ML yields lower latency if $p / o \geq 1.16$.

Min-Memory (MM). This traversal is used to maximize the consumption of data per unit time. In other words, we schedule boxes in an order that yields the maximum increase in available memory (per unit time).

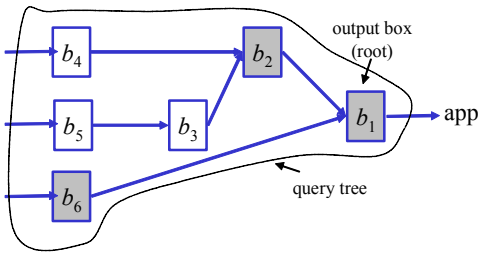

Figure 4: Sample query tree

$$mem\_rr(b) = \frac{tsize(b) \times (1 - selectivity(b))}{cost(b)}$$

The above formula is the expected memory reduction rate for a box b (*tsize*(b) is the size of a tuple that reside on b's input queue). Once the expected memory reduction rates are computed for each box, the traversal order is computed as in the case of ML.

Let's now consider the *MM traversal* of the superbox in Figure 4, this time with the following box selectivities and costs: b_1 = (0.9, 2), b_2 = (0.4, 2), b_3 = (0.4, 3), b_4 = (1.0, 2), b_5 = (0.4, 3), b_6 = (0.6, 1). Assuming that all tuples are of size one, *mem_rr* for all the boxes, b_1 through b_6 respectively, are computed as follows: 0.5, 0.3, 0.5, 0, 0.2, 0.4. Therefore, the MM traversal is:

$b_3 \rightarrow b_6 \rightarrow b_2 \rightarrow b_5 \rightarrow b_3 \rightarrow b_2 \rightarrow b_1 \rightarrow b_4 \rightarrow b_2 \rightarrow b_1$

Note that this traversal might be shorter at run time: for example, if b_5 consumes all of its input tuples and produces none on the output, the execution of b_3 after b_5 will clearly be unnecessary. In this example, the average memory requirements for MM, MC, and ML turn out to be approximately 36, 39, and 40 tuples, respectively (memory requirements are computed after the execution of each box and averaged by the number of box executions).

It is clear that different traversal approaches are effective at optimizing for the metrics that they address.

4.2 Tuple Batching - Train Processing

A *tuple train* (or simply a *train*) is a sequence of tuples executed as a batch within a single box call. The goal of tuple train processing is to reduce overall tuple processing costs. This happens due to several reasons: First, given a fixed number of tuples to process, train processing decreases the total number of box executions required to process those tuples, thereby cutting down low-level overheads such as scheduling overhead (including maintenance of the execution queue and memory management), calls to the box code, and context switch. Second, train processing has the effect of improving memory utilization by reducing the number of times a tuple gets shuttled back and forth between memory and disk throughout its lifetime. This affect becomes apparent if system operates under memory stress. A third reason, which we do not directly explore in this paper, is that some operators may optimize their

execution better with larger number of tuples available in their queues. For instance, a box can materialize intermediate results and reuse them in the case of windowed operations, or use merge-join instead of nested loops in the case of joins.

The Aurora scheduler implements train processing by telling each box when to execute and how many queued tuples to process (unlike traditional blocking operators that wake up and process new input tuples as they arrive). This approach somewhat complicates the implementation and increases the load of the scheduler, but is necessary for creating and processing trains, which significantly decrease overall execution costs.

Aurora allows an arbitrary number of tuples to be contained within a train. And, presently, train sizes in Aurora are fixed at a percentage of the queue size (usually 100%). In general, the size of a train can be decided by constraining a specific attribute such as the number of tuples, variance in latencies, total expected processing cost, and total memory footprint. We leave the investigation of more sophisticated train construction policies to future work.

5 Experimental Evaluation

5.1 Experimental Testbed

We will use the Aurora prototype system to study our operator scheduling techniques. The reference run-time architecture is defined in Section 2.2.

The prototype is implemented on top of Debian GNU/Linux using C++. In the experiments, we used a dedicated Linux workstation with dual 1.5Ghz Pentium IV processors and 1GB of RAM. The machine is isolated from the network to avoid external interference.

Due to the fact that the domain of stream-based applications is still emerging and that there are no established benchmarks, we decided to artificially generated data streams and continuous queries to characterize the performance of our algorithms, as described below.

We generated an artificial Aurora network as a collection of continuous queries, each feeding output tuples to individual applications. We modeled a continuous query as a tree of boxes rooted at an output box (i.e., a box whose outputs are fed to one or more applications). We refer to such a query tree as an *application tree*. Each query is then specified by two parameters: *depth* and *fan-in*. Depth of a query specifies the number of levels in the application tree and fan-in specifies the average number of children for each box.

For ease of experimentation, we implemented a generic, *universal* box whose per-tuple processing cost and selectivity can be set. Using this box, we can model a variety of stateless stream-based operators such as filter, map, and union. For purposes of this paper, we chose not to model stateful operators as their behavior is highly-dependent on the semantics they implement, which would introduce another dimension to our performance evaluation. This would complicate the understanding of the results without making a substantial contribution to the understanding of the relative merits of the algorithms.

An Aurora network consists of a given number of query trees. All queries are then associated with *latency-based* QoS graphs, each of which is specified by three points: (1) maximum utility at time zero, (2) the latest latency value where this maximal utility can be achieved, and (3) the deadline latency point after which output tuples provides zero utility.

To meaningfully compare different queries with different shapes and costs, we use an abstract *capacity* parameter that specifies the overall load as an estimated fraction of the *ideal* capacity of the system. For example, a capacity value of .9 implies that 90% of all system cycles are required for processing the input tuples. Once the target capacity value is set, the corresponding input rates are determined using a straightforward open-loop computation. Because of various system overheads, the CPU will saturate much below a capacity of one.

The graphs presented in the rest of the paper provide average figures of six independent runs, each producing 10000 output tuples.

5.2 Operator Batching – Superbox Scheduling

We investigate the benefits of superbox scheduling by looking at the performance of several approaches: the random (RANDOM), round-robin (RR), and the p-tuple (P-TUPLE) algorithms run in the default box-at-a-time (BAAT) mode, and the ML and MC traversal algorithms applied to superboxes that correspond to entire applications (i.e., application-at-a-time or AAAT). Figure 5 shows the average tuple latencies of these approaches as a function of the input rate (as defined relative to the capacity of the system). Also shown is an intermediate approach, the top-5-spanner, which uses ML as the traversal scheme. As the arrival rate increases, the queues eventually saturate and latency increases arbitrarily. The interesting feature of the graphs in the figure is the location of the inflection point. RANDOM-BAAT and RR-BAAT do particularly badly. In these cases, the scheduling overhead of both of the box-at-a-time approaches is very evident. This overhead effectively steals processing capability from the normal network processing, causing saturation at much earlier points. The curve for P-TUPLE-BAAT illustrates the use of the slope-slack technique. Notice that it does better than the other BAAT algorithms at low input rates, as it takes into account the tuple latencies. As the input rates increase, it saturates early just as the other BAAT algorithms do; however, when saturation occurs, it manages to maintain a reasonable latency for value over a much broader range. On the other hand, both the ML_AAAT and the MC_AAAT algorithms perform quite well in the sense that they are very resistant to high load. The AAAT techniques experience fewer scheduler calls and, thus,

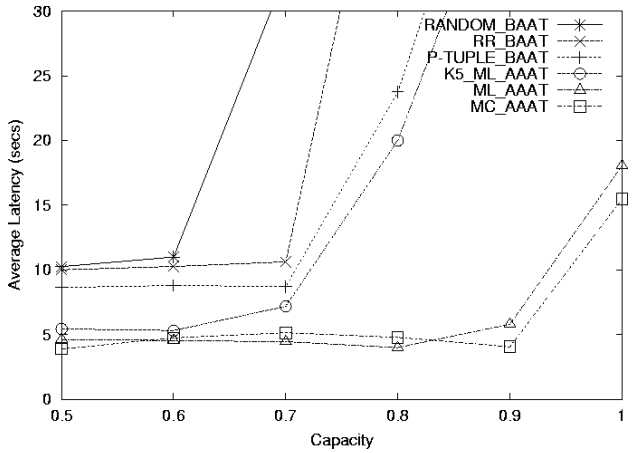

Figure 5: Application vs. Box Scheduling

have more processing capacity. These algorithms are able to *hang on* at input rates of over 90% of the theoretical capacity. Recall that the top-5-spanner algorithm picks a spanning tree that includes the top 5 highest priority boxes. While it is categorized as an AAAT algorithm, it will in general only traverse a subset of the entire application. As k is increased, the top-k-spanner algorithm approaches application at-a-time scheduling. Thus, top-k-spanner is an intermediate approach that can be tuned to behave somewhere between the BAAT and the AAAT approaches. The graph in Figure 5 bears this out. The curve for top-5-spanner starts out with an average latency that tracks the AAAT algorithms and then quickly deteriorates to track the BAAT cases.

5.3 Superbox Traversal

We first investigate the performance characteristics of the Min-Cost (ML) and Min-Latency (MC) superbox traversal algorithms. In this experiment, we use a single application tree with a box selectivity of one, a fan-in of 1.2, a depth of ten, and a CPU utilization of 0.9.

Figure 7 shows the average output tuple latency as a function of per-tuple box processing cost. As expected, both approaches perform worse with increasing processing demands. For most of the cost value range shown, ML not surprisingly performs better than MC as it is designed to optimize for output latency. Interestingly, we also observe that MC performs better than ML for relatively small processing cost values. The reason is due to the relationship between the box processing cost and box call overhead, which is the operational cost of making a box call. The box call overhead is a measure of how much time is spent outside the box versus inside the box (processing tuples and doing real work). As we decrease the box processing costs, box call overheads become non-negligible and, in fact, they start to dominate the overall costs incurred by the algorithms. As we explained in Section 4.1.2, an MC traversal always requires less number of box calls than ML does. We thus see a cross-over effect: for smaller box processing costs, box call overheads dominate overall costs and MC wins.

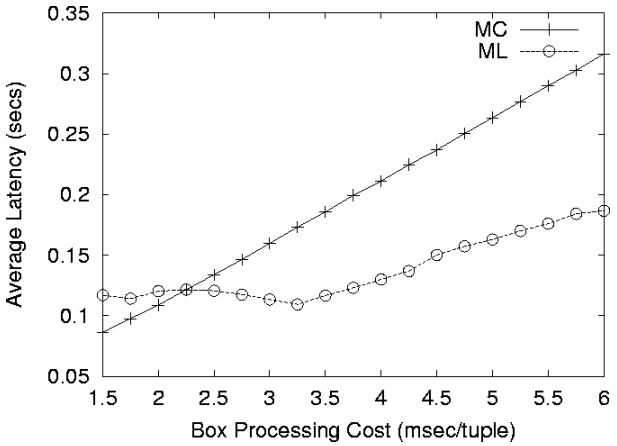

Figure 7: Processing Costs vs. Average Latency

For larger processing costs, ML wins as it optimizes the traversal for minimizing output latency.

Figure 6 presents a complementary result by plotting the overall box call overheads for different application tree depths for MC and ML. As argued before, MC incurs less overall box overhead as it minimizes the number of box calls. The difference increases as the applications become deeper and increase in the number of boxes. In fact, the overhead difference between the two traversals is proportional to the depth of the traversed tree. Consider a complete application tree with a fan-out of f and a depth of d. Then the additional number of box calls needed to be made when the depth of the tree is incremented is roughly:

$$O(df^{d+1}) \text{ and } O(f^{d+1})$$

for ML and MC, respectively.

This result can be utilized statically and/or dynamically for improving scheduling and overall system performance. It is possible to statically examine an Aurora network, obtain box-processing costs, and then compare them to the (more or less fixed) box processing overheads. Based on the comparison and using the above result, we can then statically determine which traversal algorithm to use for improved QoS. A similar finer-grained approach can be taken dynamically. Using a

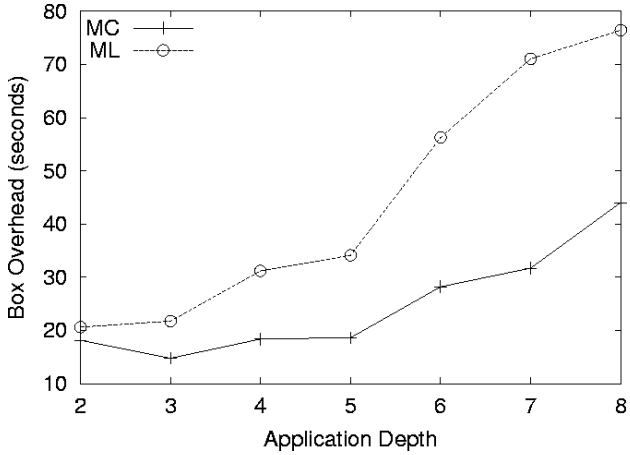

Figure 6: Box overheads for ML and MC

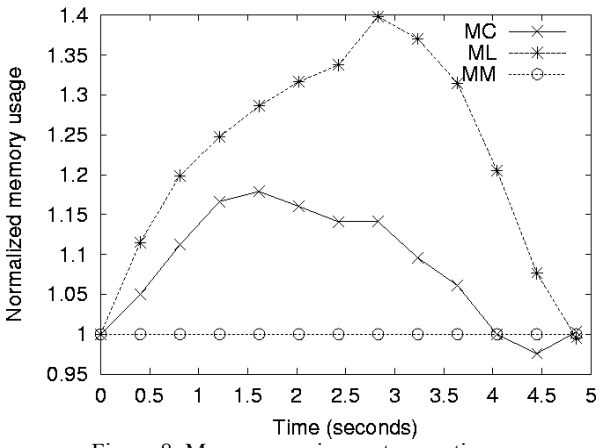

Figure 8: Memory requirements over time

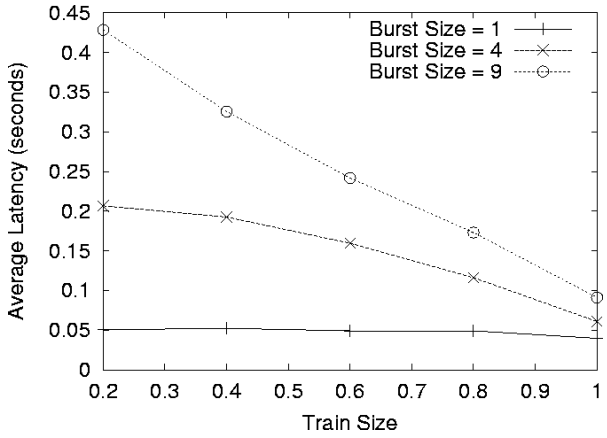

Figure 9: Train size vs. latency

simple cost model, it is straightforward to compute which traversal algorithm should do better given a particular superbox.

Figure 8 demonstrates the amount of memory used over the time of superbox run. The curves are normalized with respect to MM values. ML is most inefficient in its use of memory with MC performing second. MC minimizes the amount of box overhead. As a result MC discards more tuples per unit of time than ML.

MM loses its advantage towards the end since all three traversals are executed on a common query network. Even though each chooses a different execution sequence and incurs different overhead, all of them push the same tuples through the same sequence of boxes. The crossover towards the end of the time period is a consequence of the fact that different traversals take different times to finish. In general, MC has the smallest total execution time–the reason why it catches up with MM at 4 seconds.

5.4 Tuple Batching - Train Scheduling

Train scheduling is only relevant in cases in which multiple tuples are waiting at the inputs to boxes. This does not happen when the system is very lightly loaded. In order to see how train scheduling affects performance, we needed to create queues without saturating the system. We achieved this by creating a bursty (or clustered) workload that simply gathers tuples in our previously studied workloads and delivers them as a group. In other words, if our original workload delivered n tuples evenly spaced in a given time interval *T*, the bursty version of this delivers *n* tuples as a group and then delivers nothing more for the next *T* time units. Thus, the bursty workload is the same in terms of average number of tuples delivered, but the spacing is different. The graph in Figure 9 shows how the train scheduling algorithm behaves for several bursty workloads. The train size (x-axis) is given as a percentage of the queue size. As we move to the right, the trains *bite off* a larger and larger portion of the queues. With burst size of one, all tuples are evenly spaced. This is equivalent to the normal workload. Notice that the curve for this workload is flat. If there are no bursts, train scheduling has no effect. For the other two curves, however, as the burst size increases, the effect gets more pronounced. With small a train size of 0.2, the effect on latency of increasing the burst size is substantial. For a burst size of 4, we quadruple the average latency. Now as we increase the train size, we markedly reduce the average latency for the bursty cases. In fact, when the train size is equal to one (the whole queue), the average latency approaches the latency for the non-bursty case. Trains improve the situation because tuples do not wait at the inputs while other tuples are being pushed through the network. It is interesting to note that the bursty loads do not completely converge to the non-bursty case even when the train size is one (i.e., the whole queue). This is because the tuples still need to be processed in order. Since the bursty workload generation delivers *n*-1 of the tuples early, their latency clock is ticking while the tuples in front of them are being processed. In the non-bursty case, the tuples arrive spaced out in time, and a fair amount of processing can be done on queued tuples before more tuples arrive.

5.5 Overhead Distribution

If we turn our attention to Figure 11, we will see a comparison of the execution times and how they are distributed for BAAT, ML, and MC for three different Aurora networks consisting of 10, 20, and 30 applications (i.e., continuous queries). The y-axis is total time execution time spent when processing these applications. Each bar is divided into the four fundamental cost components.

The first thing to notice is that BAAT is significantly worse than the other two methods, and the difference increases with increasing number of applications. This again underscores our conclusion that train and superbox scheduling are important techniques for minimizing scheduler overheads.

Additionally, this graph shows clearly that scheduler overhead and box call overhead dominate the effects of

loading and unloading queues, as well as the processing costs. The difference between MC and ML is due to the fact that ML typically incurs more box calls and that MC achieves higher tuple batching. As a result, MC achieves smaller total execution times and reduced total scheduling and box overheads.

As suggested in the introduction, we were able to significantly cut down system overheads using a combination of operator and tuple batching, as exemplified by MC and ML.

6 QoS-Driven Priority Assignment

We first discuss how we compute box priorities and, at a coarser level, output priorities using application-specific QoS information and tuple latencies. We first describe our basic approach and then propose an approximation technique, based on bucketing and pre-computation, which is used to improve scalability by trading off scheduling overhead with scheduling quality. The latter is our main contribution in this section.

6.1 Computing Priorities

The basic approach is to keep track of the latency of tuples that reside at the queues and pick for processing the tuples whose execution will provide the most expected increase in overall QoS. Taking this approach per tuple is not scalable. We therefore maintain latency information at the granularity of individual boxes and define the *latency* of a box as the averaged latencies of the tuples in its queue.

Our priority assignment approach is to order the boxes in terms of their *utility* and *urgency*. We define the importance of a box b in terms of its *expected slope value*, *slope*(b), and define its urgency in terms of its *expected slack time*, *slack*(b).

Utility computation: We compute the utility of b as follows:

$$utility(b) = gradient(eol(b))$$

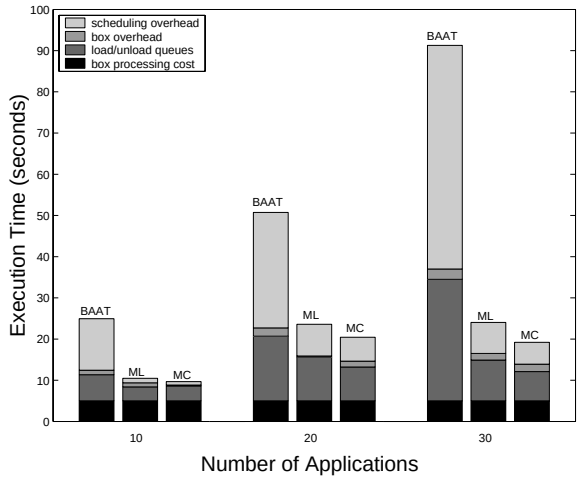

Figure 11: Distribution of overheads

This value is the gradient of the QoS-latency curve for b's corresponding output at the latency value *eol*(b), where *eol*(b) is the *expected output latency* of b. This value is an estimation of where b's tuples currently are on the QoS-latency curve at the corresponding output. In other words, this value provides a lower bound on the expected latency of the corresponding tuples at the output (assuming that the tuples are pushed all the way to the output without further delay). The value *eol*(b) is computed by adding the current latency value to the expected computation time for a given output as follows:

$$eol(b) = latency(b) + cost(D(b))$$

$$cost(D(b)) = \sum_{k \in D(b)} cost(k)$$

where $D(b)$ is the set of boxes downstream from and b (i.e., $D(b)$ is the sequence of boxes that lie on the path that start at b and end at the root box) and *sel*(b) is the selectivity of box b.

The intuition behind this utility function is that it measures the expected QoS (per unit time) that will be *lost* if the box is *not* chosen for execution.

Urgency computation: The *expected slack time*, *est*(b), is an indication of how close a box is to a *critical point*; i.e., a point where QoS sharply changes. Urgency can be trivially computed by subtracting the expected output latency from the latency value that corresponds to the critical point. If there are multiple critical points, *est*(b) always corresponds to the distance to the closest critical point.

These concepts are illustrated in Figure 10, where the QoS is specified as a piece-wise linear function of latency with three critical points.

Combining utility and urgency: At each scheduling point in time, we can order the boxes with respect to their *priority tuple*, or *p-tuple*:

$$priority(b) = (utility(b), -est(b))$$

In other words, we first choose for execution those boxes that have the highest utility, and then choose from among those that have the same utility, the ones that have the minimum (i.e., least) slack time.

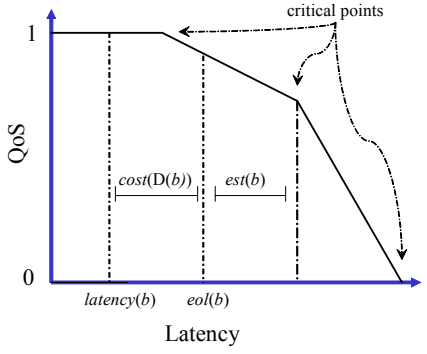

Figure 10: Critical points and expected output delay

Figure 12 shows a comparison of Aurora's QoS-aware scheduling approach and a simple round-robin scheduling policy. Both algorithms perform BAAT scheduling. The graph reveals a significant difference between the average QoS values achieved by the algorithms. The difference is pretty much stable up to a capacity value of 0.7, after which the system becomes overloaded and the performances of both algorithms decrease drastically and will eventually drop to zero (note that they remain above zero due to the finite amount of time experiments were run).

A straightforward implementation of this approach requires, at each scheduling point, computing the p-tuple for each box and then sorting the boxes with respect to their p-tuples, which is an $O(n \times \log n)$ operation, where n is the number of boxes.

6.2 Approximation for Scalability

We improve upon the basic algorithm using a combination of (1) *approximation* (via bucketizing) and (2) *pre-computation*. Our approach is to partition the utility-urgency space into discrete buckets, and efficiently assign boxes to individual buckets based on their *p-tuple* values at run time. During scheduling, buckets can be traversed in the order of decreasing *p-tuples* (illustrated in Figure 13), and the corresponding boxes are placed in the execution queue. Given a *latency* value, our first goal is to compute the corresponding bucket assignment in $O(1)$. To do this, we make use of two auxiliary graphs, gradient- and slack-latency graphs.

Gradient buckets: We divide the range of the gradient (i.e., utility) values into g buckets (Figure 14 shows an example with four buckets). All gradient values in the same bucket are treated as the same. The width of each bucket, thus, defines a bound on the inaccuracy (or variance) that we are willing to tolerate in terms of the potential deviation from the highest possible gradient value. In other words, the width of a bucket is a measure of the bound on the quantitative deviation from the optimal (with respect to the heuristic) scheduling decision.

Slack buckets: Similarly, we divide the slack values into s buckets (Figure 15) and treat all the slack values

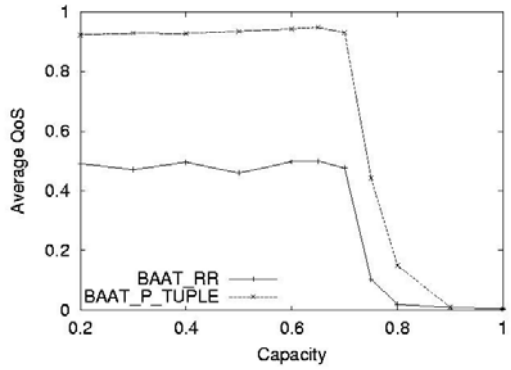

Figure 12: QoS-aware scheduling

within a single bucket as equal. Again, the width of a bucket is an indication of the level of approximation we make with regards to the slack values.

Given pre-computed gradient-latency graphs, it is possible to pre-compute the application-specific latency ranges that correspond to each bucket. For example, b_1 will be in $bucket_2$ beyond *latency* = 5 and in $bucket_3$ beyond *latency* = 15; whereas b_3 will be in $bucket_1$ till latency = 12 and in $bucket_4$ afterwards. Slack-latency graphs can be interpreted in a similar fashion as illustrated in the figure: b_1 falls in $bucket_2$ when latency is between 5 and 10, and in $bucket_1$ for other latency values.

When the execution queue is about to become empty, the scheduler performs *bucket assignment* by going through the boxes and assigning them into their current buckets. A straightforward implementation of bucket assignment takes $O(n)$ time by going through all the boxes, computing the bucket for each box in $O(1)$. This approach has the potential drawback of redundantly reassigning buckets for each box, even if the box's bucket has not been changed since the last assignment. In particular, we want the bucket assignment overhead to be proportional to the number of boxes that made a transition to another bucket. In order to accomplish this, we use a *calendar queue* [5], which is a multi-list priority queue that exhibits $O(1)$ amortized time complexity for the relevant operations (*insertion*, *deletion*, and *extract-*

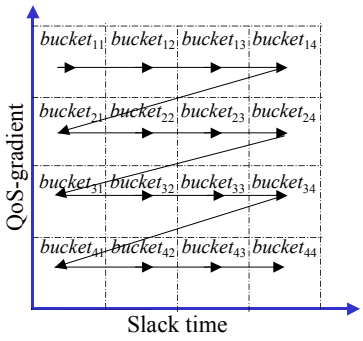

Figure 13: Bucket traversal

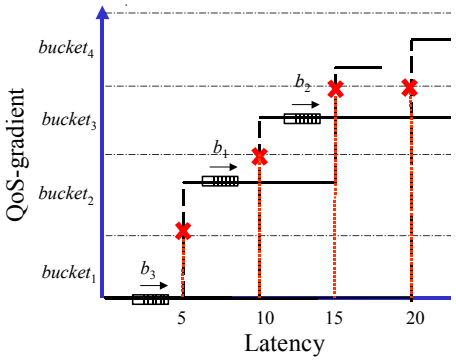

Figure 14: Gradient-latency graph

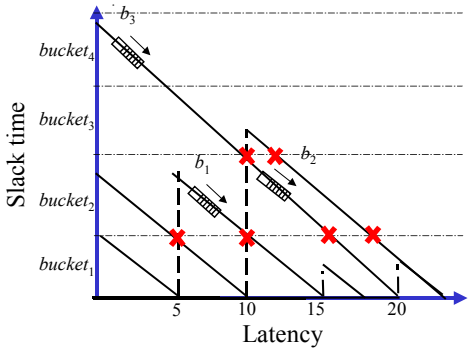

Figure 15: Slack-latency graph

min) under popular event distributions. As a result, we can implement all phases of bucket assignment in constant amortized time.

7 Related Work

There has been extensive research on scheduling tasks under real-time performance expectations both in operating systems [11, 13, 15, 18] and database systems [2, 8, 9, 16, 17]. To the best of our knowledge, Aurora's scheduling approach that combines priority assignment and dynamic scheduling plan construction is the first comprehensive proposal for scheduling continuous queries over real-time data streams and QoS expectations. Our solutions no doubt borrow a lot from the myriad of existing work on scheduling. Due to lack of space, however, we only discuss related work that is particularly relevant to our work and highlight the primary differences.

Scheduling proposals for real-time systems commonly considered the issue of priority assignment and consequent task scheduling based on static (table- or priority-driven) approaches or dynamic (planning or best-effort) approaches [17]. Static approaches are inherently ill suited for the potentially unpredictable, aperiodic workloads we assume, as they assume a static set of highly periodic tasks. Dynamic planning approaches perform feasibility analysis at run-time to determine the set of tasks that can meet their deadlines, and rejecting the others that cannot [11]. Our approach is to accept all new tasks (i.e., incoming tuples) but provide no guarantees that they will meet their deadlines (or in our QoS model their topmost critical points). This decision is based on two key observations: First, our priority assignment algorithm is based on a variation of Earliest-Deadline-First (EDF) algorithm [13], which is well known to have optimal behavior as long as no overloads occur. Second, Aurora employs a *load shedding* mechanism (not described in this paper but can be found in [6]) that is initiated when an overload situation is detected and that selectively sheds load to get rid of excess load in a way that least impacts the QoS. This allows our scheduling algorithm to focus only on underload situations. We note here that Haritsa *et al*. [9] proposed an extension of EDF that is designed to handle overloads through adaptive admission control.

Real-time database systems [2, 8, 9, 12, 16, 17] attempt to satisfy deadlines associates with each incoming transaction, with the goal of minimizing the number of transactions that miss their deadlines. These systems commonly support short-running, independent transactions, whereas Aurora deals with long-running continuous queries over streaming data and therefore has to deal with fine-grained operator-level scheduling (i.e., superbox scheduling) and tuple-level processing (i.e., train processing). Leaving aside these differences, of particular relevance to Aurora scheduling is the work of Haritsa *et al*. [8] that studied a model where transactions have non-uniform *values* (or utilities) that drop to zero immediately after their deadlines. They studied different priority assignment algorithms that combine deadline and value information in various ways, one of which is a *bucketing* technique. This technique is similar to ours in that it assigns schedulable processing units into buckets based on their utility. The differences are that (1) we use bucketing to trade off scheduling quality for scheduling overhead and, consequently, for scalability; and (2) we also use bucketing for keeping track of slack values.

Also related to Aurora scheduling is the work on adaptive query processing and scheduling techniques [3, 10, 19]. These techniques address efficient query execution in unpredictable and dynamic environments by revising the query execution plan as the characteristics of incoming data changes. Eddies [3] tuple-at-a-time scheduling provides extreme adaptability but has limited scalability for the types of applications and workloads we address. Urhan's work [19] on rate-based pipeline scheduling prioritizes and schedules the flow of data between pipelined operators so that the result output rate is maximized. This work does not address multiple query plans (i.e., multiple outputs) or deal with and support the notion of QoS issues (and neither does Eddies).

Related work on continuous queries by Viglas and Naughton [20] discusses rate-based query optimization for streaming wide-area information sources in the context of NiagaraCQ [7]. Similar to Aurora, the STREAM project [4] also attempts to provide comprehensive data stream management and processing functionality. Their initial scheduling goal involves minimization of the intermediate queue sizes [14], an issue that we do not directly address in this paper. Neither NiagaraCQ nor STREAM has the notion of QoS.

8 Conclusions

This paper has experimentally investigated scheduling algorithms for stream data management systems. It has demonstrated that the effect of system overheads (e.g., number of scheduler calls) can have a profound impact on real system performance. We have run our experiments on the Aurora prototype since simulators do not reveal the intricacies of system implementation penalties.

Processor allocation in a stream processor like Aurora could be achieved by assigning a thread per box. This technique does not scale since no system that we are aware of can adequately deal with many thousands of threads. More importantly, any such approach would abdicate the details of scheduling to the operating system. This paper shows that a more application-aware approach to scheduling can make a significant difference to overall system performance.

We have further shown that our approaches of train scheduling and superbox scheduling help a lot to reduce system overheads. We have also discussed exactly how these overheads are affected in a running stream data

manager. In particular, these algorithms require tuning parameters like train size and superbox traversal methods.

We also addressed QoS issues and extended our basic algorithms to address application-specific QoS expectations. Furthermore, we described an approximation technique that trades off scheduling quality with scheduling overhead.

Choices in these areas need to be made carefully based on knowledge of the workloads and the applications. We have provided some interesting results in this direction, and we intend to extend these studies as a guide to effective scheduler deployment.

References

[1] D. Abbadi, 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. Brown Computer Science CS-02-10, August 2002.

[2] R. J. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *ACM Transactions on Database Systems (TODS)*, 17(3):513-560., 1992.

[3] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, 2000.

[4] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109-120, 2001.

[5] R. Brown. Calendar Queues: A Fast O(1) Priority Queue Implementation of the Simulation Event Set Problem. *Communications of the ACM*, 31(10):1220-1227, 1988.

[6] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, 2002.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, 2000.

[8] J. R. Haritsa, M. J. Carey, and M. Livny. Value-Based Scheduling in Real-Time Database Systems. *VLDB Journal: Very Large Data Bases*, 2(2):117-152, 1993.

[9] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest Deadline Scheduling for Real-Time Database Systems. In *IEEE Real-Time Systems Symposium*, 1991.

[10] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2):7-18, 2000.

[11] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Symposium on Operating Systems Principles*, 1997.

[12] B. Kao and H. Garcia-Molina, "An Overview of Realtime Database Systems," in *Real Time Computing*, W. A. Halang and A. D. Stoyenko, Eds.: Springer-Verlag, 1994.

[13] C. D. Locke. Best-Effort Decision Making for Real-time Scheduling.

[14] R. Motwani., J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. Stanford University TR 2002-41, August 2002.

[15] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proc. 16th ACM Symposium on OS Principles*, 1997.

[16] G. Ozsoyoglu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(4):513-532, 1995.

[17] K. Ramamritham. Real-Time Databases. *Distributed and Parallel Databases*, 1(2):199-226, 1993.

[18] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55-67, 1994.

[19] T. Urhan and M. J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, Rome, Italy, 2001.

[20] S. Viglas and J. F. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, 2002.