

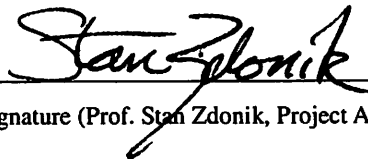
Priority-Based Bandwidth Allocation in Aurora*

Alexander Rasin

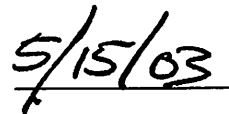
Department of Computer Science

Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the
Department of Computer Science at Brown University



Signature (Prof. Stan Zdonik, Project Advisor)



Priority-based bandwidth allocation in Aurora*

Ugur Centintemel, Alexander Rasin, Stan Zdonik

21st March 2003

Abstract

There are many application areas in which real-time data stream processing is needed. Aurora is a data stream manager that addresses those needs. Aurora* is a distributed Aurora query network. It scales significantly better since it is able to utilize hosts (or nodes) in parallel. Unfortunately, by distributing Aurora query network we introduce a new limited resource – inter-node bandwidth. In addition to allocating CPU time for processing tuples at the node, we must also allocate available bandwidth for transferring tuples from node to node. Based on the application classes that are more sensitive to bandwidth constraints than to CPU constraints, we would like to concentrate on the problem of bandwidth allocation for the tuples. The problem is similar to real time scheduling of CPU tasks or delivering multimedia streams, but most of the existing algorithms are not suited for our needs. We would like to consider several algorithms and present experimental evaluation of algorithm performance using a simulation of Aurora*.

1 Introduction

Many modern applications require processing infinite streams of data tuples in real time. Examples include monitoring of an army in the battlefield or analyzing stock data in real time. Unlike in conventional non-streaming databases, the inputs and outputs are usually push based and processing of a query is potentially infinite. Aurora [4] is a data stream manager that is designed to handle the concurrent processing of multiple continuous queries. Each query consists of a directed acyclic graph of SQL-like operators (or boxes). A query processes some continuous tuple streams and produces results for one (or more) stream-based client applications. Applications specify their Quality-of-Service (QoS) requirements to announce their relative importance and guide resource allocation in Aurora query network. Since the query network must scale well for arbitrary size, we introduce Aurora* [1], the distributed version of Aurora. In this paper, we modify the Aurora QoS model slightly by assigning the QoS requirements to tuple streams rather than output applications.

An example application that we would like to run is the MITRE application. The purpose of their application is to monitor the battle arena that contains

both friendly and hostile units. Several data sources periodically report positions and heading directions of the units present on battlefield. Data valuation specifications determine the delivery requirements for each object type. The data valuation specifications have several parameters, as shown in the table:

Class	Value	Initial Delivery	Update	Friend/Foe	Ground/Air
Class a	High	Fast	Frequent	Foe	Air
Class b	Medium	Slow	Frequent	Friend	Air
Class c	Low	Slow	Medium	Foe	Ground

The first three parameters determine the performance expected from a particular object class. Value determines the relative importance of an object. In general, if we must drop one of the objects, we drop the object with lowest value. Initial delivery determines the allowable delay between the object creation and the time object is delivered to an output application. Finally the update value determines the frequency with which the object stream must be refreshed.

The operations performed on the objects are simple, thus little CPU time is needed for processing. However, the bandwidth between different command and analysis centers that process data is limited. Therefore the bottlenecks of the MITRE query network are the limited bandwidth links between the network components. The goal of the network is to satisfy the requirements of as many highest value object classes as possible (i.e. deliver the tuples within the allowed delay and deliver as many as the update field requires). A secondary goal is to deliver as many tuples as possible from the lower value streams that whose goals were not completely met.

Let us define the environment and our assumptions. We would like to keep our environment simple and ignore CPU scheduling issues to focus our attention on bandwidth allocation. Our query network is a directed acyclic graph that is distributed over one or more physical nodes. The nodes are connected by network links with a specified constant bandwidth. The client applications are located on some other node(s) and are also connected to Aurora* outputs via their node network links. Aurora* has some inputs that receive continuous tuple streams (produced by some data source, for example, a temperature sensor). Data input sources are producing input tuples at a uniform rate but we make no strong assumptions about the distribution of inter-arrival times (in our simulation we use exponential distribution). The tuples are processed by the network and are passed to the client applications as they reach the outputs. Since we are concentrating on bandwidth allocation, we assume that the throughput capacity of node's CPU is higher than the throughput capacity of the incoming network link. Therefore queues never form on intra-node arcs and the processing time of a tuple within a node is considered negligible.

Each tuple consists of three fields: the arrival (at input) time-stamp, data payload and priority class. The priority class designates the relative importance of a tuple. The size of a tuple is assumed to be constant for a tuple class and is used to calculate the time (cost) of tuple transfer over a network link ($cost = \frac{size}{bandwidth}$). A sequence of tuples with the same priority a tuple stream

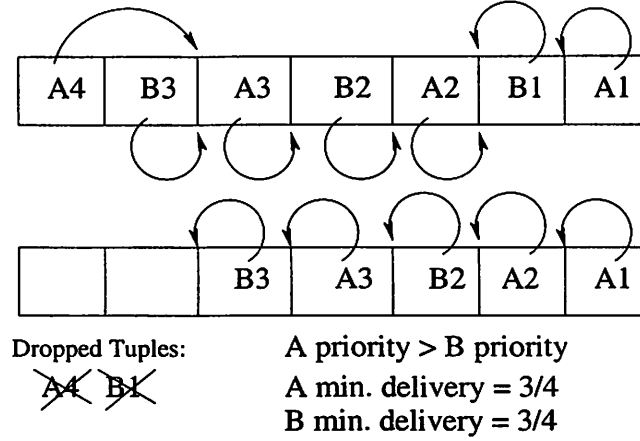


Figure 1: Priority schedule building under our model

of that priority. In this environment our goal is to give preference to more important data streams over less important ones. An example might be two sensors one of which reports the nuclear reactor core temperature and the other that reports coffee machine temperature. Here's (Figure 1) an example reordering that we might expect. Assuming that As are high priority tuple stream and Bs are low priority and that A's update frequency is 3 out of every 4 tuples we might expect the following schedule behavior:

In Figure 1 arrows refer to the tuple deadline (the latest point at which the tuple will be accepted by client application). Thus the first slot is taken by A1, second slot is taken by A2 (B1 is dropped since As are higher priority). B2 gets the third slot without competition and A3 gets the fourth slot. Note that in competition for the fifth slot B3 wins because A's minimum delivery requirement (3 out of 4 tuples) is already fulfilled.

Each tuple stream has Quality-of-Service (QoS) specifications that define the service expectation for the stream. The specification consists of a deadline and a minimum delivery rate. Deadline is the maximum allowed latency for each tuple ($= \text{Timestamp}_{\text{arrival}} - \text{Time}_{\text{output}}$). A tuple that misses its deadline provides no utility to the client application and does not need to be delivered at all. Minimum delivery rate is the percentage of tuples from the data stream that must be delivered to a receiving application. Minimum delivery rate is specified as a fraction $\frac{m}{n} \leq 1$ which means that out of a window of n tuples at least m must be delivered. The window is a tumble window, i.e. the stream is split into disjoint windows of size n . Delivery rate is not guaranteed for arbitrary sliding window (any continuous tuple subset of size n) because this would place very deterministic requirements on the tuples that must be delivered (or might be impossible to meet). Note that the $\frac{m}{n}$ fraction also specifies the maximum allowable number of undelivered consequent tuples ($2 * (n - m)$).

An example of such stream might be a news broadcast. The deadline is

the maximum allowable delay between the time the news are announced at the source and the time the recipient hears them. Minimum delivery rate corresponds to the number of packets that must be delivered from the source in order for the audio stream to be coherent.

2 Related work

2.1 Bandwidth allocation as a real-time scheduling problem

The Aurora* bandwidth allocation problem can be mapped into a real-time scheduling problem with some modifications. Real-time scheduling problem deals with scheduling a set of tasks with soft (can be missed) or hard (cannot be missed) deadlines. We assume that tuples can not miss their deadlines, corresponding to hard deadlines. Our limited resource is the bandwidth thus instead of allocating A CPU cycles for a task, we are allocating B K-bytes per second for tuple transfer. A tuple is a task that has a constant cost ($\frac{Size}{Bandwidth}$), a priority (priority class) and a deadline. We assume that tuple transfer is atomic which corresponds to non-preemptive tasks.

While these problems are similar, there are some properties that distinguish our problem from real-time scheduling. Our input does not have a periodic nature (neither can we guarantee a minimum inter-arrival rate at any time). We also require the delivery of some percentage of the tuples delivered on time to meet QoS requirements with flexibility allowed in dropping tuples. There is also another, more subtle problem that we might have to deal with. Class deadline is defined at the output. The scheduler at every link must be independent and make local decisions in order for the scheduler to scale at all. Therefore the intermediate deadlines at nodes that the tuple traverses on its way must be chosen by our algorithm.

2.2 Scheduling algorithms

Liu and Layland presented an optimal fixed priority scheduling algorithm, the *Rate Monotonic* (RM) algorithm as well as two dynamic priority scheduling algorithm, *Earliest Deadline First* (EDF) and *Minimum Laxity First* (MLF) [10]. These algorithms serve as a basis for real-time scheduling research and represent the two approaches to scheduling – static and dynamic. Static schedule requires the least overhead but provides worse utilization guarantees than the dynamic schedule. The priorities are computed *a priori* and do not change at the run time. Dynamic schedule requires a relatively high overhead but is able to guarantee higher CPU utilization and automatically adapts to changes in task arrival rates.

RM algorithm assigns priorities based on the frequency of task arrival. RM assumes periodic tasks for which deadline of the task is equal to the inter-arrival time of that task. The task with highest arrival frequency gets the highest

priority. The scheduler always executes the task with the highest priority. RM assignment is optimal in a sense that no better static assignment of priorities exists. The CPU utilization for which a set of n tasks is guaranteed to be schedulable is $W_n = n(2^{1/n} - 1)$. A CPU utilization of a single task P_i is defined as a ratio of its cost C_i to its period T_i .

EDF algorithm assigns priority according to task's deadline. The task that has the earliest deadline (i.e. the one that will be the first to miss its deadline) has the highest priority and gets precedence over other tasks. EDF guarantees 100% CPU utilization and will automatically adjust to any changes in the task set. EDF assignment is optimal, as it is able to find a feasible schedule if one exists. These advantages come at a price: 1) EDF overhead is higher than RM overhead 2) EDF does not perform well in case of *transient overload* as it has no sense of how *critical* a task is. When the processor is overloaded, EDF might keep running earliest deadline tasks even if they are going to miss their deadline. Minimum Laxity First (MLF) algorithm is very similar to EDF algorithm. The task with the smallest laxity (or slack) is assigned the highest priority. As a result the cost of the task is taken into consideration when choosing task's priority (but not the task's priority).

Jones et al. presented Rialto operating system [8] that supported CPU reservations using precomputed schedules. In Rialto the responsibility of load shedding is relegated to the thread. They also assumed a common base period for all scheduled tasks by proportionally scaling the task reservation request when necessary. McCanne et al. propose Receiver-driven Layered Multicast (RLM) in [12] where data transferred is encoded in multiple layers. The receiver is responsible for shedding higher-order layers when it detects a network congestion. In [7] Fall et al. developed an Early Discard Load Shedding (EDLS) algorithm which would detect and uniformly discard MPEG frames when CPU is overloaded. EDSL does not support meeting specific deadlines or specifying the amount to be shed per video stream. RLM and EDSL are used in tandem to meet streaming multimedia QoS requirements.

In [3] Waldspurger and Weihl describe a random Lottery Scheduling algorithm. Such approach does not work in our environment because the share of link utilization is determined by data arrival rate. The decisions that our scheduling algorithm must make consist of dropping and reordering tuples rather than deciding what bandwidth share should stream get. As some of tuple streams are (temporarily) exhausted, the priorities of remaining streams change unpredictably.

Dovrolis et al., in [6] presents a differentiating mechanism that allows controlling the average packet delay by service class. His mechanism supports a ratio based differentiation between the delays. Note that for our purposes average class delay is inappropriate, since only the tuples whose delay is smaller than deadline are useful. Chang et al in [5] studies the use of Weighted Fair Queuing and Priority Queuing. This paper demonstrates that such algorithms can be used provided admission control is used (as these algorithms share the bandwidth between the existing users). This paper also distributes the resources in some proportion rather than to meet a specific fixed goal.

3 Scheduling

Let us define what tuple slack is: *tuple slack* is the difference between the tuple's arrival deadline and the tuple's arrival time if it were sent now. Tuple's arrival deadline is the tuple timestamp plus the deadline of the tuple's priority class. Tuple's arrival time if sent now is equal to the current time plus the cost of transmitting the tuple. In other words, the tuple slack is the maximum amount of time one can delay the tuple and still expect it to arrive on time.

3.1 First In First Out

FIFO is the most straight forward baseline approach to scheduling. The tuples are processed as they arrive without any regard to their deadlines or priority classes. This approach requires zero overhead since there is no management performed at the physical link. Note that this algorithm is modified to keep it competitive with the dynamic algorithms. When tuple is about to be sent, we compute its slack and if the slack is negative, we drop the tuple instead of sending it. If we allow delivering tuples that we know to be late, we severely penalize this algorithm by wasting bandwidth. If capacity is exceeded and we are unable to shed tuples, the queues will keep growing and none of the tuples will be delivered on time.

3.2 Static Priority

This algorithm simply maps the priority class (or value) of a tuple into tuple's priority. The highest priority class gets the highest priority and lowest priority class gets lowest priority. This approach is similar to shedding of lower priority classes in case of congestion. It is not equivalent, because lower priority tuples will be sent if there are no higher priority tuples queued at the time. Note that this algorithm is modified in the same way as FIFO and will shed tuples with negative slack.

In order to avoid overhead we can maintain two first-in-first-out queues for each priority class (one queue for normal tuples and another one for best effort tuples). When selecting the next tuple, we select one from the highest priority non-empty queue (lowest priority normal queue supersedes highest priority best-effort queue).

3.3 Minimum Slack First

Minimum Slack First (MSF) algorithm uses the same approach to scheduling as MLF. Tuples are prioritized based on their slack. If the tuple's slack is negative the tuple is going to be dropped. MSF does not allow us to distinguish between priority classes but does consider the relative tuple cost. MLF is fairly straightforward to implement and is very predictable (we will demonstrate it in results section).

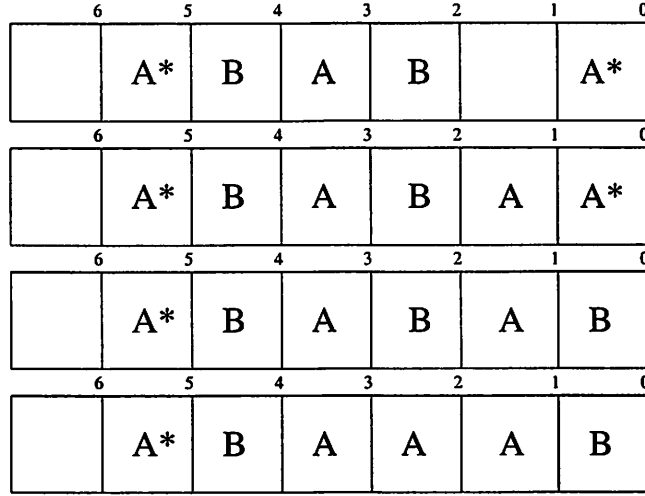


Figure 2: A demonstration of several insertion steps for Conveyor Belt

3.4 Conveyor Belt Scheduling

We would like to define a dynamic algorithm similar to MLF. We approach it in the following way: we maintain a schedule of slots which are filled with tuples as the tuples arrive. We assume the tuples to be of the same size but this model can be easily extended to accommodate tuples of different size. Any time a tuple arrives, we compute its slack and find the latest slot at which the tuple could be scheduled. We attempt to insert the tuple starting from that slot and moving towards the beginning of the schedule. This is done in up to three different passes.

Note that some of the tuples are statically marked as non-mandatory according to minimum delivery requirements. For example if the minimum delivery requirement is $\frac{3}{4}$, one in every four tuples is marked for non-mandatory delivery. This approach is simpler than the dynamic approach and based on the simulation results it performs the same as dynamic approach. Figure 2 demonstrates several steps of the scheduler. Non-mandatory tuples are denoted with a *. Class A priority is higher than Class B priority. The top schedule is the initial schedule that we have. A tuple A arrives and it has a slack equal to 3. Starting from the first slot we do a pass looking for empty slots. We find an empty one at position 2 and place the newly arrived tuple there (second schedule reflects that). Starting from the second schedule, we receive a tuple B with slack of 4. A pass through last four slots find no empty place for the tuple. Therefore we fall over to a second pass that detects non-mandatory tuples in the schedule. At slot 1 we find A* and replace it by B (reflected in the third schedule). The tuple is discarded rather than delayed, since all the slots it could have used are taken by higher priority tuples (any mandatory tuple has a higher priority than a non-mandatory one). Finally, an A tuple arrives with a slack

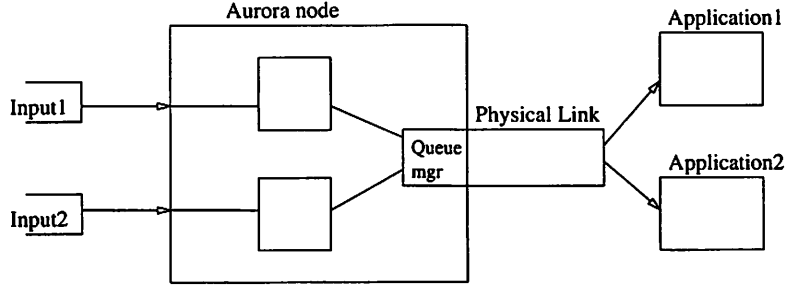


Figure 3: Single node simulation setup

of 3. We perform a scan for free schedule slots and, failing that, we scan for non-mandatory tuples. Since that attempt to find a slot also fails, we resort to the third pass that looks for lower priority tuples. A tuple in slot 2 has a lower priority since priority A is higher than priority of B. Therefore the tuple B in slot 2 is replaced by A in the last, fourth schedule.

This algorithm has the highest overhead compared to other algorithms. However, the length of the queue is bounded by the longest slack a tuple could have. The longest slack is in turn bounded by tuple deadlines, thus the overhead is constant per tuple.

4 Simulation results

Our simulation is built in C++, using the CSIM18 library written by Mesquite Software [11]. CSIM provides all the routines necessary for running a simulation using a virtual time counter.

The initial network consists of one Aurora* node (see Figure 3). The network is fed by two tuple sources that produce tuples of priority class 1 (highest) and 2. The inputs are fed to a box in Aurora node and the box output is sent to a queue manager that makes tuple scheduling decisions before they are sent via the physical link. Applications count only the tuples that have been delivered on time. The default inter-arrival rates are 200ms and 300ms (exponentially distributed) for Input1 and Input2 respectively. Default tuple size is 500 bytes and default minimum delivery requirements are $\frac{6}{10}$ for both priority classes. The default deadlines are 1400ms and 1950ms for Class1 and Class2 respectively. We measure and average the data over a run time of 800 seconds.

4.1 Static scheduling algorithms

First we would like to compare the performance of static algorithms. Naturally, we expect the SP priority algorithm to do better as it is able to shed load when link capacity is insufficient. In Figure 4 we measure the tuples delivered vs link bandwidth in Kbytes/sec. The number of tuples delivered is normalized with respect to the minimum delivery requirements. We compute the expected total

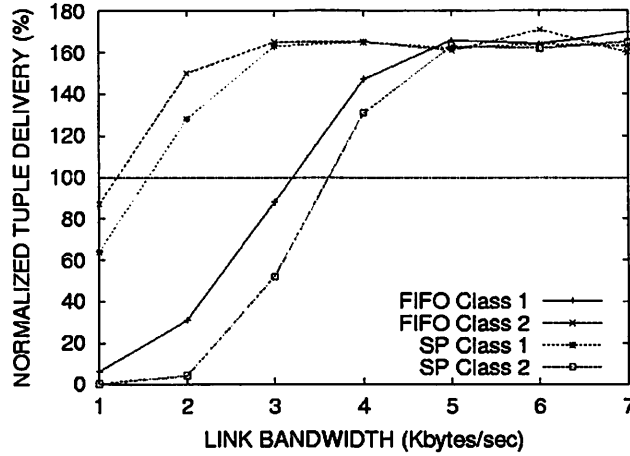


Figure 4: Static Priority and First In First Out: tuples delivered

number of tuples using the input rate of a class. In Figure 4 the y-value of 0 means no tuples were delivered, 60 means that 60% of the expected minimum number of tuples have been delivered. So the value of 100% corresponds to a point where particular class requirements are met. For example SP Class 2 (denoted by square) achieves the value of 100 and thus meets its requirements at around 3.6 Kbytes/sec.

The point of 100%-intersection is an important measurement. The intersection represents the amount of bandwidth that is required to meet the minimum delivery requirements of the current class (and all priority classes that were already met). Each curve represents a combination of class and scheduling algorithm. In the rest of our figures, we are going to measure that number to represent the performance of a scheduling algorithm. Clearly, the less bandwidth a priority class requires, the better.

In Figure 5 we plot the minimum bandwidth required by FIFO versus changes in the deadline of Class 1. The absolute value of the deadline is not as significant as the relationship between the Class 1 and Class 2 deadlines. Therefore we normalize the Class 1 deadline on the x-axis with respect to the (fixed) value of Class 2 deadline. The x-axis values range from $\frac{1}{5}$ to about $\frac{9}{5}$. Points of intersection in a graph such as the one in Figure 4 corresponds to a point on Figure 4. In fact, the two values at x equals to $\frac{4}{5}$ are taken from the intersections we see in Figure 4. Important thing to note is that the minimum bandwidth required for satisfying both priority classes is the max of two bandwidth values for two classes. Therefore the smaller value is the bandwidth required for satisfying the one class, but the larger value is the bandwidth needed for satisfying

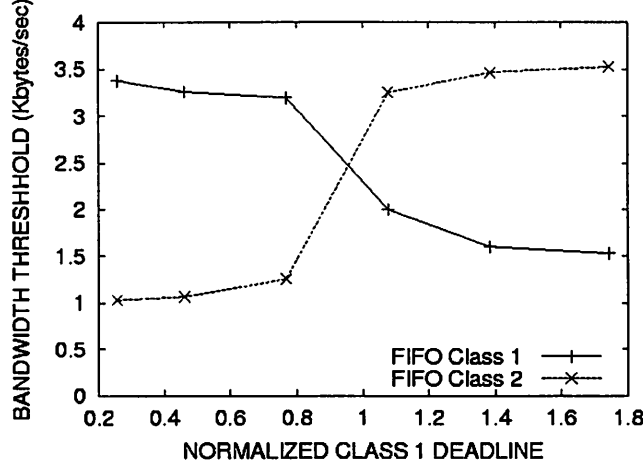


Figure 5: FIFO performance

both classes.

Note the behavior of FIFO in Figure 5. The priority class with the longer deadline has a clear advantage as a bigger fraction of tuples from that class arrive on time. The minimum bandwidth curves of priority classes under FIFO intersect at x equals to 1. At the point where both class deadlines are equal, the bandwidth requirements of the two priority classes matches. It is obvious that FIFO is very unstable and is not well suited for meeting our goals. SP in Figure 6 behaves better than FIFO. It is able to meet the requirements of class 1 at a very low bandwidth, though that comes at a price of needing a lot of bandwidth to satisfy the requirements for both classes. The behavior of SP is close to that of an algorithm that simply sheds all class 2 tuples. A good property of SP is its ability to meet the requirements of class 1 at very low bandwidth. Unfortunately, that means that to meet the requirements of both classes, we need a significant amount of bandwidth.

4.2 Dynamic Scheduling Algorithms

Figure 7 demonstrates performance of our dynamic algorithms, MSF and CB. MSF demonstrates some interesting properties. While it does not consider the priority of tuples when scheduling, it is able to achieve the requirements of both tuple classes at a reasonably low bandwidth ($\sim 2.5k$). However, MSF will fail to satisfy either of the tuple classes if less than 2.5Kbytes/sec is available. The algorithm orders tuples by their slack and, since the arrival rates are randomly distributed, the tuples are dropped uniformly and equally from both tuple

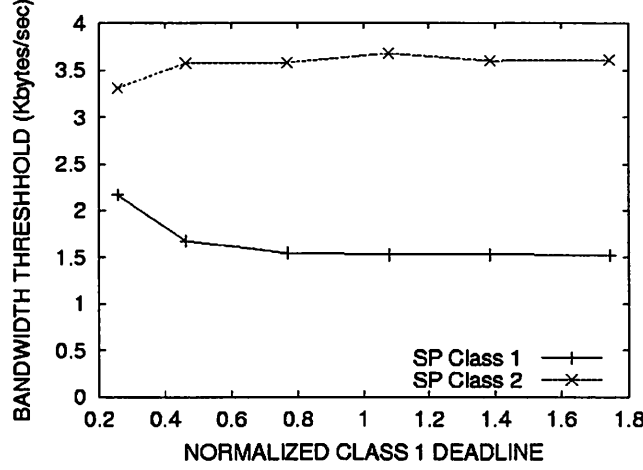


Figure 6: SP performance

streams. As a result, the delivery requirements of both classes are achieved at the same time. Note that while the required bandwidth for a class is always a nearly straight line (i.e. independent of the relationship between class 1 and class 2 deadlines), they are not necessarily at the same level as in Figure 7 (see next section, and Figure 8).

CB algorithm differs from MSF. CB considers the tuple priorities and tries to guarantee that the needs of class 1 are met first. It is fairly restricted at very small class 1 deadlines (i.e. left side of x-axis) since the slack of arriving class 1 tuples is very small. However as we increase the deadline of class 1, CB is able to decrease the bandwidth required for satisfying class 1 significantly. The performance of Class 2 is not decreased, however. CB is able to improve its performance by producing more efficient schedules. A longer deadline increases the average tuple slack and thus it also increase the size of the schedule that CB maintains. As the schedule size increases, CB becomes more flexible in its decisions. While CB requires relatively little bandwidth to satisfy class 1, it requires more bandwidth to satisfy both class 1 and class 2.

Figure 7 clearly demonstrates that depending on the bandwidth that we have available, we will have to select a different algorithm. None of the algorithms are optimal under all circumstances, thus we need to develop a hybrid algorithm.

4.3 Hybrid Scheduling Algorithm

As pointed out in previous section, we need to design the hybrid algorithm that will perform well using any available bandwidth. Both MSF and CB have

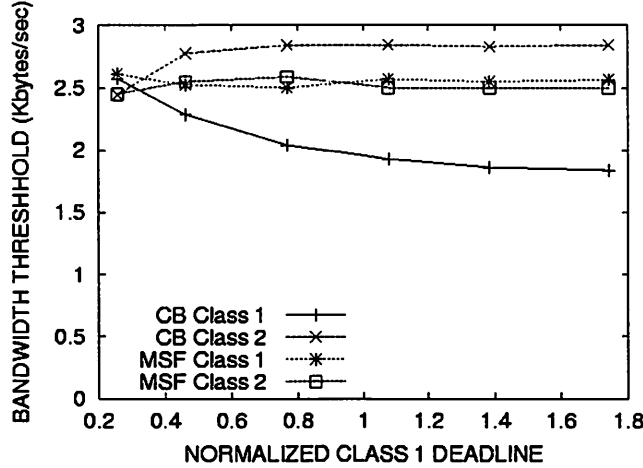


Figure 7: MSF and CB performance

their advantages and weaknesses. Even SP might be preferable under some circumstances.

The advantage of SP is obvious. Due to the heavy shedding that it employs, it is able to satisfy class 1 requirements at slightly lower bandwidths than CB can. CB can satisfy class 1 requirements at a slightly higher bandwidth, but at the same time it allows class 2 tuples to pass. What distinguishes CB is its ability to reorder tuple to maximize the number of tuples that are delivered on time. CB can satisfy Class 1 at low bandwidth while still delivering as many Class 2 tuples as it can. CB is particularly effective in the case when the deadline of Class 1 is large. Thus until the bandwidth is sufficient for MSF to meet the requirements of both classes, CB is our best choice.

The advantage of MSF is in the uniform way it sheds extra load when a link is congested. As a result, MSF is able to satisfy both priority classes at lower bandwidths than other algorithms. CB loses to MSF in that sense because it must make sure that Class 1 requirements are satisfied before it can give bandwidth to Class 2 (i.e. it is greedy when reserving space for Class 1 tuples and thus its efficiency is reduced). As a result CB has less flexibility than MSF in shedding tuples. Once the bandwidth is sufficient for MSF, it does better than other algorithms, since no other algorithm can meet the requirements of both priority classes at such low bandwidths.

The final question that we need to answer is whether it is possible to compute the threshold bandwidth value at which one should switch to MSF. Due to MSF's predictability it is possible. The computation is rather simple. Based on the parameters, using 200ms and 300ms for average inter-arrival rate, 500bytes

for sizes and $\frac{6}{10}$ for both priority classes we can compute the bandwidth that MSF should require. Since loss is distributed evenly, we must base our estimation on the priority class with highest requirements, so we select $\max(\frac{6}{10}, \frac{6}{10})$ as minimal delivery multiplier:

$$(\frac{1second}{200ms} * 500bytes + \frac{1second}{300ms} * 500bytes) * \frac{6}{10} = 2.5Kbytes/sec$$

Figure 8 demonstrates that the above computation works for any combination of minimum delivery requirements. The top graph shows the computed threshold values for MSF and the bottom graph shows the same values measured using our simulation.

5 Future Work

There are quite a few directions in which this work should be extended. Some are obvious extensions of the considerably simplified model, others are complementary techniques to improve performance of Aurora*

5.1 Multiple Priority Classes

Clearly we would like to be able to schedule more than two classes at the same time. The algorithms described require no modifications to work with any number of priority classes. However, it would be even more interesting to consider whether the problem of distributing bandwidth resources can be simplified by considering priority classes in pairs. It is possible to distribute the bandwidth between different priority class pairs and utilize most appropriate (possibly different) scheduling algorithms for each pair.

5.2 Multiple Aurora Nodes

Our simplified model works with a single Aurora node. Consider an Aurora* query network consisting of two sequential Aurora nodes. The problem in this network is that it is unclear what the class profile specifications are for the first node. The class parameters (such as deadline, minimum delivery rate, etc.) are defined at the output. The intermediate nodes do have such things defined. Thus the first question we need to resolve is whether we need to take any additional steps. Thus in Figure 9 we analyze the performance of our algorithms with no modifications. We assume that both links have same bandwidth.

Figure 9 demonstrates the performance of all 4 algorithms using default settings. Note that FIFO performs somewhat worse in the distributed case (due to the additional cost incurred by the second transfer). It does not deteriorate greatly, however, since most of its tuples have sufficient amount of slack to pay the additional transfer cost and still arrive on time. SP algorithm performs almost identically in the distributed case as in single node case. The reason for that is due to SP tuples having the highest slack as they arrive to the intermediate queue. SP always gives precedence to priority class 1 and as a result the tuples of class 1 have relatively high expected slack upon arrival. As

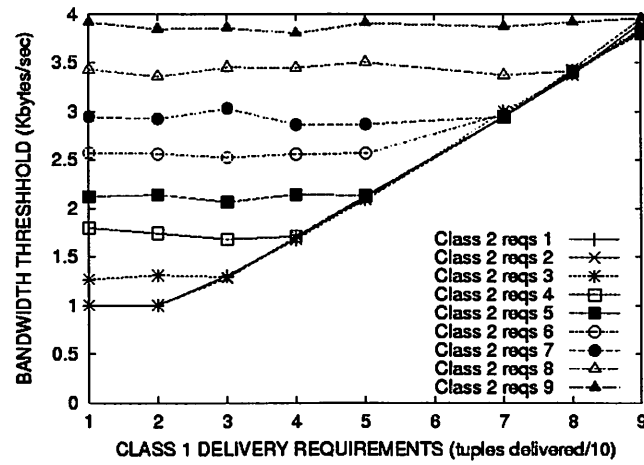
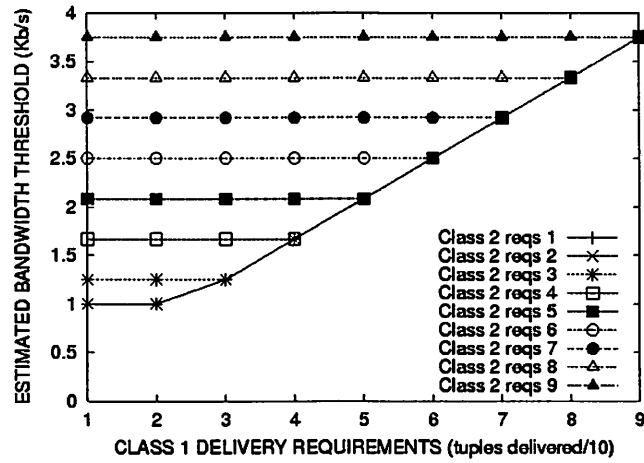


Figure 8: MSF switch threshold, measured vs estimated

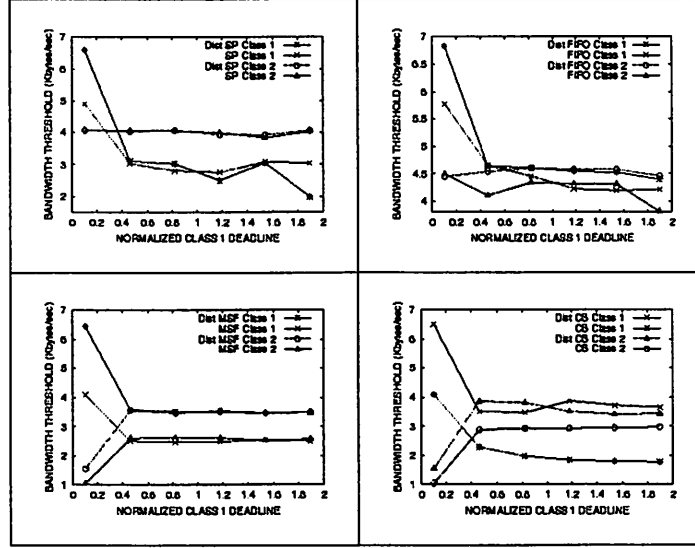


Figure 9: Unmodified algorithms on multiple nodes (Distributed) vs performance on single node

a result SP is barely affected by the extra cost of tuple transfer on the second Aurora node.

Both MSF and CB exhibit significant performance deterioration in the two node network. This is happening because by their nature both of these algorithms try to optimize performance and maximize the number of tuples arriving on time by reordering the tuples. Therefore tuples are often reordered so that they are transferred at their last feasible slot, i.e. a point in time where their slack is very close to 0. As a result most of the tuples do not have the extra slack that can compensate the cost of the transfer via the second Aurora node. Not surprisingly at the bandwidth that's sufficient for a single node, most of these tuple arrive late in the two node case. So we see that the static algorithms are not severely affected by the lack of knowledge of the second node, yet the dynamic algorithms are severely handicapped by the same lack of knowledge.

This brings us to conclusion that, at least for dynamic algorithms, we must take additional steps and propagate the class profile specifications to intermediate nodes. We are not going to present these results here, but an even deadline distribution (first node sets the requirements for class deadlines at $\frac{1}{2}$ of their final output deadline) shows little performance deterioration in the distributed case.

5.3 Migrating boxes to reduce link congestion

If a physical link between two nodes is congested, we might be able to reduce its load by migrating Aurora boxes. Each Aurora nodes in Aurora* contains some queries with some number of operators (or boxes). A subset of a query may migrate to another Aurora node as long as the integrity of the query network is maintained. This ability is generally used to balance CPU load between the nodes. However, if we detect a link saturation and the upstream box has a low average selectivity (i.e. if it discards a lot of the tuples it receives), we may want to migrate the box. As a results many more tuples will be discarded before they are pushed through the link.

6 Conclusion

We have analyzed adaptations of existing real-time scheduling algorithms and proposed our own algorithm. In general we found no fully optimal algorithm. In some case we might want to use SP or MSF algorithm. However, our proposed algorithm (CB) closely approximates the performance of both of these algorithm based on the bandwidth changes. At low bandwidth the performance in highest priority class comes close to that of SP. When bandwidth is sufficient, CB performs similarly to MSF, though slightly worse. In an environment where some of the parameters fluctuate (bandwidth or input rates) CB would be optimal as it guarantees performance of highest priority stream while passing the remaining resources to a lower priority class.

References

- [1] 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. Brown Computer Science CS-02-10, August 2002.
- [2] R. Braden, L. Zhang, S. Berson, S. Herzog and S. Jamin, "Resource ReSeR-Vation Protocol (RSVP) Version 1, Functional Specification," RFC 2205, IETF, Sept. 1997
- [3] Carl A. Waldspurger and William scheduling: Flexible proportional-share man-agement. In First Symposium on Operating De-sign and Implementation (OSDI), pages California, November 14-17 1994.
- [4] 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.

- [5] L.F. Chang, Z. Jiang, N.K. Shankaranarayanan, Providing Multiple Service Classes for Bursty Data Traffic in Cellular Networks, Proc. 19 th Conf. on Computer Communications (IEEE Infocom), Tel-Aviv, Israel, 2000.
- [6] Dovrolis C., Stiliadis D. and Ramanathan P., Proportional Differentiated Services: Delay Differentiation And Packet Scheduling. Proceedings of SIGCOMM '02
- [7] Fall, Kevin and Pasquale, Joseph and McCanne, Steven, Workstation Video Playback Performance with Competitive Process Load , Proc. NOSSDAV '95, Apr 1995, p. 179-182
- [8] M. B. Jones and P. J. Leach. Modular real-time resource management in the rialto operating system. Technical Report MSR-TR- 95-16, Microsoft Research, Advanced Technology Division, May 1995.
- [9] J. P. Lehoczky and S. Ramos-Thuel, An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems, in 13th Real-Time Systems Symposium, pp. 110 123, Dec. 1992.
- [10] Liu, C. L., and J. W. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment, Journal of the Association for Computing Machinery, v.20, n.1, January 1973, pp. 44-61.
- [11] Mesquite Software, home page: <http://www.mesquite.com/>
- [12] McCanne, Steven and Jacobson, Van and Vetterli, Martin, Receiver-driven Layered Multi-cast , Proceedings of SIGCOMM '96, Aug 1996, p.117-130
- [13] Salama, H.F., Reeves, D.S. and Viniootis, Y., A Distributed Algorithm for Delay-Constrained Unicast Routing, IEEE INFOCOM'97, pp.84-91, 1997