CLASS-BASED CONTINUOUS QUERY SCHEDULING IN DATA STREAM MANAGEMENT SYSTEMS

by

Lory Al Moakar

Bachelor of Science, American University of Science and Technology, 2004

Submitted to the Graduate Faculty of
the Dietrich School of Arts and Sciences, Department of Computer Science in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

University of Pittsburgh

2013

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES, DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Lory Al Moakar

It was defended on

May 28, 2013

and approved by

Alexandros Labrinidis, Associate Professor, Department of Computer Science

Panos K. Chrysanthis, Professor, Department of Computer Science

Kirk Pruhs, Professor, Department of Computer Science

Mohamed Sharaf, Assistant Professor, University of Queensland

Dissertation Advisors: Alexandros Labrinidis, Associate Professor, Department of Computer Science,

Panos K. Chrysanthis, Professor, Department of Computer Science

CLASS-BASED CONTINUOUS QUERY SCHEDULING IN DATA STREAM MANAGEMENT SYSTEMS

Lory Al Moakar, PhD

University of Pittsburgh, 2013

The emergence of Data Stream Management Systems (DSMS) facilitates implementing many types of monitoring applications via continuous queries (CQs). However, different monitoring applications will have different quality-of-service (QoS) requirements for detecting events. For example, the CQs for detecting anomalous events (e.g., fire, flood) have stricter response time requirements over CQs which are for logging and keeping statistical information of interesting physical phenomena. Traditional DSMSs treat all the CQs as being equally important in the system and attempt to optimize their overall performance. In particular, they employ a CQ scheduler to decide the execution order of CQs to achieve a global performance goal and as such perform badly in an environment where CQs have different importance levels.

The hypothesis of this research is that there is a need for a suite of schedulers that optimizes the response time of important CQs while satisfying the requirements of the other, less important classes and taking into consideration the underlying processing environment. Toward this, we first develop the Continuous Query Class (CQC) scheduler for single-core / single-process systems which is assumed by many of the current DSMS prototypes, including our own, AQSIOS. Then, we propose the Adaptive Broadcast Disks scheduler (ABD) which is more suitable for dual-core environments. After that, we extend our work to multi-core environments to take advantage of modern machine architectures and their processing capabilities. We propose the Multi-core Broadcast Disk scheduler (MBD) which optimizes the response time of the critical CQs while maintaining acceptable performance for less-critical classes. In addition, it also utilizes the cores efficiently and provides better performance. We demonstrate the effectiveness of our schedulers through a

thorough experimental evaluation using new metrics under AQSIOS, our prototype DSMS, and SimAQSIOS, a simulator that closely mimics AQSIOS.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

PREFACE

I would like to thank my advisors, Alexandros Labrinidis and Panos K. Chrysanthis, for their support, advice and feedback in the last six years. I would also like to give special thanks to my committee members, Kirk Pruhs and Mohamed Sharaf for their feedback and for collaborating with me on a couple of exciting projects.

I am also grateful for my fellow PhD students at the ADMT lab: Roxana Gheorghiu, Shenoda Guirguis, Thao Pham and Panickos Neophytou for all the discussions and friendship during the last few years. I also received generous moral support from the other graduate students at the Computer Science Department. I would like to mention Michel Hanna, Michal Valko, Christine Chung, Weijia Li, Iyad Battal, and Rakan Maddah who were there for me with their awesome friendship. Very special thanks goes to my coffee buddies: Vinicius Petrucci and Alexandre Ferreira for all the coffee conversations and laughs we shared. I will never forget all the good advice and help I received from the staff at the department especially Kathleen O'Connor, Kathleen Allport, Karen Dicks and Keena Walker.

I would also like to express the deepest appreciation to Father Rodolph Wakim, Father Simon Elhajj and the families at Our Lady of Victory Maronite Church for blessing me with a home away from home. I would like to especially mention Therese Sadaka and the Alchoufetes: Bahige, Micheline, Therezia, Fadi and Tamara for being my second family. I owe a very important debt to my family and friends especially my parents: Jean and Salwa and my brother Alan. Their love, support and daily conversations helped me through rough times to get where I am today. Finally, I dedicate this thesis to my husband Peter whom I met on the first day of this journey. His love, support and encouragement were the crucial elements that kept me going when I was ready to give up.

1.0 INTRODUCTION

1.1 MOTIVATION

The growing need for monitoring applications such as the real-time detection of disease outbreaks, tracking the stock market, detecting credit card fraud, environmental monitoring via sensor networks, and personalized and customized Web alerts, has led to a paradigm shift in data processing, from Database Management Systems (DBMSs) to Data Stream Management Systems (DSMSs) (e.g., [1], [6], [16], [14] and [48]). In contrast to DBMSs in which the data is stored, in DSMSs, the monitoring applications register Continuous Queries (CQs) which continuously process unbounded data streams looking for events of interest to the end-user. There are already a number of commercial stand-alone DSMSs on the market, such as Streambase [52], IBM System S [54], Coral8 [18], Esper [21] and MS StreamInsight [36] aiming to support specific applications. These DSMSs treat all the CQs equally and attempt to optimize their overall performance.

The increasing demand for monitoring applications and Big Data analytics by a variety of users, will inevitably establish a need for DSMSs that recognize the relative importance of some CQs. In many applications, some of the CQs submitted to the DSMS are more critical than others. For example, a CQ that detects a fire inside a warehouse is very critical and requires short response time, whereas CQs that report average conditions are not as critical.

Traditional DSMSs employ a CQ scheduler to optimize the Quality of Service (QoS) provided by the system. The CQ scheduler is the DSMS component which decides the execution order of CQs to achieve a certain performance goal such as minimizing response time or maximizing fairness. Although there have been multiple scheduling policies proposed (e.g., Round Robin (RR), Chain [13], Highest Rate (HR) [47], Highest Normalized Rate (HNR) [47] with different QoS metrics), these are oblivious to the importance levels of different CQs and the existence of

multiple CQ classes with different QoS requirements.

To better understand the multi-class CQ scheduling problem, consider the following example where neglecting the importance of queries can lead to undesirable response times. Assume two continuous queries: CQ_1 which detects fire conditions in a forest (e.g., high temperature, low humidity, and strong wind) and as such has a high priority, and CQ_2 which records the average temperature and humidity measurements for scientific observation purposes and as such has a low priority. When considering appropriate schedulers for this example, it is a well known fact that RR is oblivious to the priorities and treats CQ_1 and CQ_2 equally. Similarly, schemes that optimize for the average response time, such as HR, or for the average slowdown time, such as HNR, are also oblivious to query importance. Even worse, they may make decisions that contradict the objective of optimizing the performance of the highly-important CQs. For instance, in our example, CQ_1 is highly selective because fire conditions are rare. Thus, CQ_1 has a low *output rate* which will lead HR to assign CQ_1 a lower scheduling priority than CQ_2. Under HR, CQ_2 has a better response time than CQ_1. The conflict between a CQ's importance and its scheduling priority is a problem which we refer to as *priority inversion*.

Priority inversion becomes a serious problem in a multi-tenant DSMS where the CQs belong to different users. In these DSMSs, the users would pay to have their CQ belong to a more important class. Thus, the DSMS in general and the scheduler in particular need to be aware of the CQ classes and their priorities. Clearly, priority inversion is undesirable in such a system. However, an efficient DSMS should not starve the lower priority classes while trying to satisfy the priorities of the more important classes.

1.2 PROBLEM STATEMENT

As inspired by the above real life examples, the CQs submitted to a DSMS could have varying levels (or classes) of importance. A traditional DSMS and in particular a traditional scheduler does not distinguish between these classes and would attempt to optimize the overall system performance without any regard to important CQs. Thus, a new scheduler is needed to schedule CQs in a DSMS when these CQs belong to different classes.

The hypothesis of this thesis is that *a CQ-class-aware scheduler can be developed to optimize the performance of important CQs while utilizing the system resources efficiently, preventing priority inversion and starvation, and adapting to the characteristics of the workload.* Such a scheduler needs to schedule CQs if the DSMS is running on a single-core, dual-core or multi-core machine.

1.3 APPROACH

This thesis proposes a family of schedulers that allow the user to specify the importance/priority of the CQs, and use these priorities to satisfy the QoS of highly important CQs, while providing reasonable QoS for low priority CQs and allocating CPU time efficiently. We assume a traditional DSMS in which overloading is avoided by the admission control and load shedding modules. In this thesis, the schedulers schedule CQs after the admission control module has executed to make sure that the system does not get overloaded. Our proposed schedulers use a new computing model that separates the source operators from the operational and output operators. In addition, these schedulers allocate CPU time in a single-core, dual-core, and multi-core environments while adhering to the goals of preventing starvation, respecting priorities, and utilizing the system resources efficiently under normal load conditions.

- **Single-core environment:** In the traditional system model where the DSMS runs in one thread/process, we designed the *Continuous Query Class scheduler (CQC)* [5] to schedule CQs of different importance by grouping them into CQ classes, where each CQ class has a user-specified priority value that indicates its relative importance. CQC is a two-level scheduler that uses a weighted RR scheduler on the top level with multiple HR schedulers on the lower level. In a single core environment, the CQC scheduler optimizes the performance of important classes while maintaining acceptable performance for classes that are not important.
- **Dual-core environment:** In order to efficiently utilize both cores in this architecture, we divide the execution into two parts: One core is used to listen on the network for any incoming tuples and put them in the input queues of the operators. The second core is used to run the CQs registered by the user. The separation between these two modules allows the system to process data tuples as they arrive without waiting for all the older tuples in the system to

finish their execution. Also, this model is better suited at our target application because it removes the dependency between important CQs and non-important CQs at the input level. To allocate resources under this system model, we designed the *Adaptive Broadcast Disks scheduler (ABD)* [4], a novel scheduler that optimizes the weighted average response time of the query classes and respects user priorities by preventing priority inversion. The ABD scheduler is designed to run with the expectation that an operator can receive tuples at any time during the scheduling cycle. It is a two-level scheduler with a Broadcast Disks-inspired scheduler on the top level with multiple per-class RR schedulers on the lower level.

- **Multi-core environment:** To better utilize modern computer architectures, we propose a new system model where the system can take advantage of all of the cores available in a multi-core environment. Under this model, one or more cores are used to listen for incoming tuples. The other cores are used for the operational and output operators. Toward this, we designed the *Multi-core Broadcast Disk scheduler (MBD)* which is a CQ-level scheduler that translates the priorities of the classes into CQ-priorities. The MBD scheduler is a two-level scheduler like the ABD scheduler. It uses the CQ-priorities to generate an MBD schedule using the same algorithm as the ABD schedule. This schedule is then used at the top level to determine the order of the execution of the CQs. Unlike the ABD scheduler which has only one scheduling point, the MBD scheduler has multiple scheduling points, equal to the number of cores dedicated to the operational and output operators. Each of these cores has a scheduler that retrieves from the MBD schedule the next CQ to execute. The MBD scheduler also has a priority inversion detection module that continuously monitors the response times of the classes and detects whether priority inversion has occurred or not. Once detected, it adjusts the priorities of the classes in a way that honors the original priorities and rebuilds the MBD schedule. In addition because the MBD scheduler is at the CQ-level, it detects if the user priorities or the selectivities change during runtime and rebuilds the MBD schedule when necessary.
- **Evaluation Metrics:** In order to evaluate the new schedulers, we designed a suite of metrics reflecting desirable properties. In our work, we use the weighted average response time in addition to the weighted response time percentiles to capture the efficiency of the schedulers and how they adhere to the priorities specified by the users. In order to detect priority inversion,

we collect and compare the average response times per class and capture the amount of priority inversion in the form of a ratio. To detect starvation, we monitor the trend of the response time for each class. If a class's response time is non-monotonically increasing for a period of time, we conclude that the class is starving.

- **Evaluation Environment:** To evaluate the schedulers in a single core environment, we use the AQSIOS which is our DSMS prototype developed at the Advanced Data Management Technologies (ADMT) laboratory at the University of Pittsburgh. For the dual-core and multi-core models, we developed a simulator called SimAQSIOS that can simulate the AQSIOS DSMS running on more than one core.

1.4 CONTRIBUTIONS

This dissertation makes four key contributions:

- **The CQC scheduler:** allocates CPU time in a single core environment for CQs belonging to multiple classes of importance without starving less important CQ classes.

- **The ABD scheduler:** schedules CQs belonging to different classes in a dual-core environment, while maintaining the priority semantics of the classes.

- **The MBD scheduler:** schedules CQs in a multi-core environment while satisfying the priorities of the classes, preventing starvation and priority inversion, and utilizing the CPU time efficiently.

- **Evaluation:** we evaluate the proposed schedulers using novel metrics via real implementation and simulation. Our results confirm our hypothesis making it possible to develop DSMSs capable of meeting the requirements of different CQs and supporting differentiated levels of service in multi-tenant environments.

1.5 ROADMAP

We discuss our system model in Chapter 2. Chapter 3 reviews the related work. Chapter 4 describes and evaluates the CQC scheduler for single-core environments. Chapter 5 describes and evaluates the ABD scheduler for dual-core environments. Chapter 6 describes and evaluates the MBD scheduler and its modules for multi-core environments. Finally, we conclude in Chapter 7

2.0 SYSTEM MODEL

2.1 AQSIOS DATA STREAM MANAGEMENT SYSTEM

A Data Stream Management System (DSMS) is at the heart of every monitoring application. AQSIOS is a DSMS prototype built on top of the STREAM prototype. It accepts data streams as an input from data sources such as sensor networks, internet sources, and social websites. These data streams arrive over the network and are of infinite nature and bursty arrival rates. The users register continuous queries (CQs) to run over the incoming data streams for a long period of time. CQs process the data tuples as they arrive at the system and output the results that are sent back to the user. AQSIOS is unique in that it allows the users to specify the importance class their CQ belongs to. All the CQs in the system have to belong to one of the different CQ classes. Each class has a numerical priority that indicates the importance of the results of the CQs in this class relative to the other CQs in the system. This priority is specified by the system administrator and gets translated by the system into the amount of resources available for the CQs in the class.

Inside AQSIOS, once the user submits a CQ, the admission control module (e.g., [37]) receives the CQ and decides whether to admit it or not depending on the current load in the system. Then if admitted, the query optimizer translates the CQ into an efficient query plan composed of operators connected via queues. AQSIOS supports three types of operators namely source, operational, and output operators (Figure 1). The source operators read and translate the data streams coming from the data sources to an internal representation format in the form of data tuples. They also record the time the tuple arrived at the DSMS as a timestamp for the data tuple. The operational operators process the data according to the query specified by the user. The Join, Select, and Project operators are examples of operational operators. Once the operational operators have finished processing the tuples, the results of the processing are put in the input queues of the output operators. The output

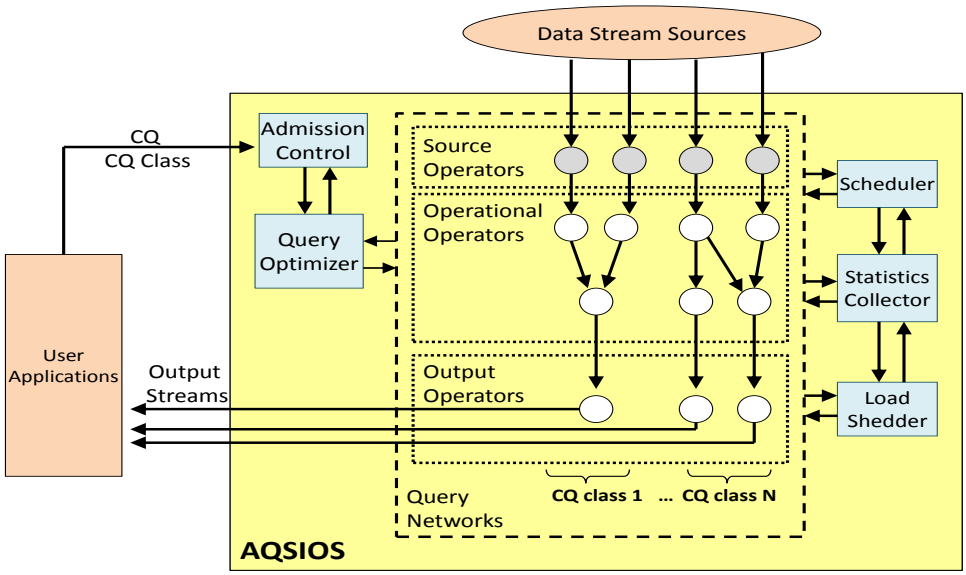

Figure 1: Overview of AQSIOS

operators then disseminate the results to the end user. They also calculate the response time of the tuple by subtracting the timestamp from the time the tuple was sent to the user.

During runtime, a scheduler is needed to allocate resources to the operators. At every scheduling point, the scheduler selects the next operator to execute and allows it to process its tuples for a specified amount of time/tuples. Depending on the scheduling scheme, the scheduler might need some statistical information about the operators. Thus a statistics collection module is employed. However, the scheduler also needs to allocate resources to the statistics collection module and refresh the priorities of the operators. Consequently, to simplify the scheduling, the statistics collection module is implemented in a distributed fashion where each operator collects statistics about its execution. The scheduler and other modules in the system can retrieve these statistics from the operators at any scheduling point. The HR scheduler for example pulls the statistics collection module once per cycle. The cycle is defined in terms of the number of tuples inputted into the system. The cycle length typically is 200 tuples so that the operators have processed enough tuples in order to update the statistics. The HR scheduler uses these statistics to set the scheduling priority of the operators. The system ensures that the statistics are valid by requiring that the statistics

collector is in steady state when HR scheduling starts. The statistics collector reaches steady state when all the operators have processed tuples and updated their statistics. These statistics are also used by the load shedding module which monitors the load in the system and decides whether to drop tuples according. In this dissertation, however, we do not enable the load shedding module (e.g., [40]) because of its effects on the performance in the system. Thus, we assume that admission control on the level of the CQs has already taken place and that the load shedder is disabled, so as to evaluate the behavior of the schedulers.

2.2 CLASS-BASED SCHEDULING IN AQSIOS

The focus of this thesis is on using the priorities of the classes under AQSIOS as part of scheduling. As such, the scheduler needs to allocate CPU time to the CQs in a way that optimizes the response time of the high-priority classes without starving the lower-priority classes and without priority inversion. Under the view of the scheduler, the source operators are not part of the query network and are scheduled separately from the operational and output operators (see Figure 1). This is done because if the source operators are scheduled in the same way as the other operators, this can result in an unnecessary overhead when a source operator is selected to execute and given the CPU, if it has no incoming tuples to read, while some intermediate tuples are still waiting inside the system to be processed. This will result in larger delays and increased waiting time for the tuples in the system.

2.2.1 AQSIOS in a single-core environment

The current version of AQSIOS runs in a single thread. Thus the scheduler is responsible for allocating resources to all of the operators: source, operational and output operators. Consequently, the scheduler has control over when new tuples arrive at the input queues of operational operators. In order to reduce the overhead of scheduling source operators that have no tuples, the source operators are scheduled to execute only when there are no tuples waiting in the input queues of the operational and the output operators. This enforces longer train execution which means that

once an operator receives the CPU, it has more than one tuple in its input queue. As a result, the overhead costs of the operator are depreciated over multiple tuples and the CPU time is utilized more efficiently.

2.2.2 AQSIOS in a dual-core environment

The next version of AQSIOS that we are developing would run on two processing cores to better utilize more modern machines. This also applies for a machine that has 2 processing threads. As mentioned in Section 2.2.1, our current version of AQSIOS does not utilize multiple cores. In fact, the system would leave one of the cores idle while delaying the processing of the heavy source operators until needed. Also under the point of view of the scheduler in the dual core environment, the source operators are not part of the query network and are scheduled separately from the operational and output operators (see Figure 1). Thus, the source operators are scheduled and run on a separate core than the operational and output operators. A simple round robin scheduler is used to schedule the sources to ensure fairness at the source operator level. This reduces the effect of scheduling the sources and makes our scheduling schemes applicable to systems that have one tuple routing module instead of individual source operators. However, another scheduler is needed to allocate resources to the operators. At every scheduling point, the scheduler selects the next operator to execute and allows it to process its tuples for a specified amount of time/tuples.

2.2.3 AQSIOS in a multi-core environment

Many modern machines are equipped with a single chip multiprocessor. This trend promises to continue into the future with machines with 16 or more cores. Thus, AQSIOS needs to be parallelized even further than two cores/threads so that it can run more efficiently and provide the users with faster response times. This also allows AQSIOS to handle heavier loads, more CQs and more users. The source operators are scheduled on their own one or more cores just as in Section 2.2.2 and the operational and output operators need to be scheduled on multiple cores. In order to do that, the scheduler needs to be aware of the number of cores allocated to the DSMS so that it can schedule operators/CQs on the cores in a way that satisfies the goals for this thesis namely preventing starvation and respecting priorities while utilizing the underlying hardware in the best

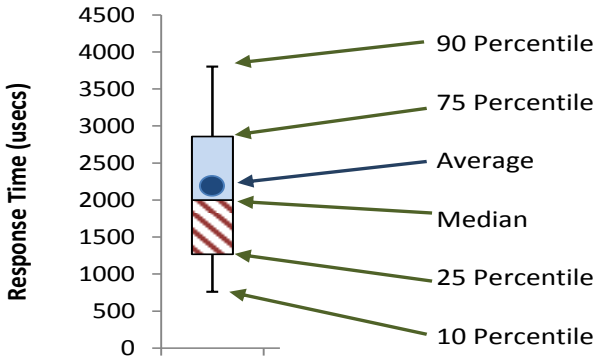

Figure 2: Box-and-Whisker chart explanation

way possible. In cases when one core is not enough to meet the load of the source operators, the scheduler can allocate more than one core for the source operators by calculating the expected load per tuple for the source operators and the operational and output operators. Then, it divides the cores among them.

2.3 METRICS FOR SCHEDULER EVALUATION

In order to evaluate our schedulers, we need to devise metrics that capture the different dimensions of performance that are of interest. Beyond the traditional response time, we need to be able to quantify whether a scheduler respects class priorities and whether a scheduler exhibits priority inversion or starvation. Capturing these dimensions in a single objective function or metric is very hard because of the variety of dimensions. For example, the weighted average response time captures partially respecting priorities and not starving lower priority classes but cannot capture priority inversion. Thus, we follow a best effort model in this thesis and compare the schedulers using a variety of metrics that capture the different performance dimensions.

2.3.1 Respecting Priorities

A CQ-class aware scheduler needs to respect the priorities of the CQ classes and its performance needs to reflect the priority order. This does not mean that a class whose priority is half of the highest priority class has double the response time. On the contrary, it indicates that the highest priority class has a better response time. Thus, to compare multiple schedulers, we utilize the overall weighted average response time instead of the average response time. In instances when the input rate changes drastically during runtime, especially in a multi-core environment, the weighted average might be overwhelmed by outliers. In those cases, other weighted percentiles are more useful such as the median (i.e., 50-percentile), 75-percentile, 90-percentile or 95-percentile.

In addition, a CQ-class-aware scheduler needs to provide improved performance to critical CQ classes as compared to a traditional scheduler. In order to capture this, we look at the response times of each class. We compare the average response time per class if no fluctuations in the input are present. If the input changes during runtime, we use multiple percentiles: a 10-percentile, 25-percentile, median, 75-percentile and 90-percentile to have a clearer picture of the performance of each class. We show these percentiles in addition to the average response time per class for our multi-core workloads using the Box-and-Whisker charts. Figure 2 shows a sample chart with pointers to these percentiles. These charts allow us to also consider the variance in the performance of the different schedulers, instead of just looking at the average value.

2.3.2 Preventing or reducing priority inversion

Priority inversion happens when a higher priority class has a higher average response time than a lower priority class. If the administrator of the DSMS desires, the definition of priority inversion in this thesis could be extended so that priority inversion also happens when a higher priority class has the same response time as a lower priority class. In the rest of this thesis, we assume that priority inversion refers a higher priority class having a higher response time that a lower priority class. Thus, if the response time of a high-priority class is lower than that of a low priority class, then priority inversion is zero. In cases when all schedulers have some priority inversion, we need to quantify the amount of priority inversion to compare and contrast. Toward this, we use the priority inversion ratio at different percentile levels. The priority inversion ratio of class i is calculated

according to the following formula:

$PrIR_i = \frac{Pr_{class(i)}}{Prclass(i+1)} \times |min\{0, 1 - \frac{RT(i)}{RT(i+1)}|\}$

Where $RT(i)$ and $RT(i+1)$ could be the average response times or any percentile response times of classes i and $i+1$ respectively. $Pr_{class(i)}$ and $Pr_{class(i+1)}$ are the priorities of classes i and $i+1$ respectively.

$PrIR_i$ is also useful over time when the input rates or the selectivities change during runtime because it measures the adaptivity of the schedulers to these changes. We also use the cumulative $PrIR_i$ to compare the overall performance of the schedulers with respect to priority inversion.

2.3.3 Preventing starvation

The goal of preventing starvation is very important because starvation impairs the performance of the system. A monotonically increasing response time over a period of time is an indication of starvation if that increase does not correspond to an increase in the input rate. In a DSMS, the response time of a CQ_i increases when its input rate increases or when it is starving. Thus, we utilize plots that show the response time of a class over time as an indication of the presence or absence of starvation.

2.3.4 Utilizing system resources efficiently

This applies primarily in a multi-core environment because an idle core is wasting resources. In a single-core environment, using the weighted average response time is enough to compare how different schedulers utilize their core. However in a multi-core environment, the weighted average response time is not enough. A multi-core scheduler needs to scale well as the number of CQs or input rate increases proportionally to the number of cores.

3.0 RELATED WORK

In this chapter, we survey the related work in the areas of scheduling in computing systems, scheduling in a DBMS, scheduling in a DSMS, multi-core scheduling and priority inversion.

3.1 SCHEDULING IN COMPUTING SYSTEMS

Scheduling has been studied extensively in the areas of operating systems, networking and real-time systems. In most cases, the cost of a task is known or can be estimated a priori. Also, deadlines are specified as a target for execution. Thus, algorithms such as Earliest Deadline First (EDF) and Shortest Remaining Processing Time (SRPT) have been proposed and used in various settings. However, in our system these algorithms cannot be applied because the cost of a CQ is highly dependent on the input rate of its input streams, the selectivity of its operators, and the number of tuples in its intermediate and input queues at scheduling time. These factors are dependent on the scheduling scheme used and incur overhead to collect and use at each scheduling point. Furthermore, CQs do not have deadlines in our system because of their continuous nature.

Multiple scheduling strategies have taken into consideration the priority or weight of a task while scheduling. Weighted Round Robin (WRR) scheduling has been proposed in [38] to schedule tasks with weights/priorities, but it assumes preemptive execution. Our schedulers do not preempt the execution of a tuple, but they utilize ideas and concepts from WRR scheduling. The work in [12] extends round robin scheduling to allocate time slices to groups of tasks with similar weights. However, the weights used are not the priorities of the tasks, but rather, the demand of the tasks. The work in [56] proposes *Lottery Scheduling* as a solution for this problem. However, this does not work very well for DSMSs because it is highly randomized and has no limits on the wait

time that a tuple can endure in the system as shown in 6.2. The work in [55] places such boundaries on the wait time by specifying a value for the stride which is the amount of time a task needs to wait before its next execution. This is similar to our proposed ABD/MBD schedule which is inspired by the work on broadcast disks [2]. Under these schedulers, each task executes an amount of time proportional to its priority. However, stride scheduling is dependent on the value of the stride and incurs more overhead than the ABD scheduler when selecting the next task to execute. Other works such as [15] and [20] monitor the amount of time each task executes for and make scheduling decisions depending on these statistics and the task weights. This is in contrast to CQs where scheduling affects the amount of time a CQ executes for due to the effects of batch processing. Monitoring the batch sizes at each scheduling point incurs a lot of unnecessary overhead. In general, these approaches provide insights into our class-based CQ scheduling problem but they do not offer complete solutions.

3.2 SCHEDULING IN A DBMS

The work in [45] considers scheduling multi-class workloads in the context of e-commerce OLTP transactions in traditional database management systems. Specifically, it divides transactions to classes of different QoS targets and uses those class-based targets to schedule transactions. The scheduler is an external module which non-preemptively dispatches a small set of transactions to the database system for execution, where the size of that set is a system parameter. However, under a non-preemptive dispatcher, a highly important transaction might be blocked waiting for a less important one to finish execution first, resulting in priority inversion.

The work in [39] considers scheduling queries with different priorities in a parallel DBMS. The authors propose a mechanism to balance the resource allocation for queries based on their priorities. The objective is to allocate to each query a portion of the CPU at least proportional to its priority while maximizing resource utilization. Other schedulers such as [49] and [24] attempt to minimize the tardiness of a query. However, our work is in the context of one-time queries not CQs that needs to be rescheduled and run for a long time.

In a similar manner to the CQC scheduler, our group has used the approach of two-level

scheduling in the context of web-databases [42]. In particular, the work in [31] deals with the problem of scheduling queries and updates in a web-database system in the presence of Quality Contracts, which specify user-preferences for Quality of Service and Quality of Data for each query. The proposed scheduling algorithm (QUTS) involves two separate queues, one for queries and one for updates, and dynamically assigns CPU time to each according to the expected "profit" in the system through the Quality Contracts framework.

3.3 SCHEDULING IN A DSMS

Efficient CQ scheduling has been one of the main techniques for improving the performance of a DSMS. This motivated the proposal of several policies for scheduling the execution of CQs in a DSMS with the objective of optimizing certain performance goals such as minimizing latency [13, 47, 48] or minimizing memory requirements [8, 9] or both [53]. Other work focuses only on certain type of CQs such as shared join CQs in [25]. In our work, we also focus on the scheduling of CQs in a DSMS, however, our objective is to optimize the DSMS performance in the presence of a multi-class workload where CQs belong to different classes according to their importance.

Related to our work on multi-class CQ scheduling is the work on the Aurora project [1, 13] which considers a set of Quality of Service (QoS) functions including a latency-based function. In particular, under such a model, each CQ is associated with a QoS function and the perceived quality of service degrades when the output delay is beyond some threshold δ. However, the objective is to improve the overall DSMS performance, whereas in this thesis, we focus on mainly improving the performance of critical CQs while still providing acceptable performance to the other less-important CQs in the DSMS.

The work on the RTSTREAM system [60] considers scheduling classes of CQs based on deadlines which are used as priorities for CQ instances during scheduling. However, when a query instance is foreseen to miss its deadline, it will be removed from the system and its input data will be discarded. Hence, the RTSTREAM approach aims at increasing the DSMS success rate by satisfying as many deadlines as possible. Other work such as [61], [33],[34],[30] & [59] also consider deadlines of tuples or CQs as a criteria of meeting QoS. This is in contrast to our approach where

all CQs are executed to completion without discarding any input data under the goal of minimizing the latency of CQs according to their importance.

The work in [26] and [58] considers both priority and deadlines where each CQ is assigned a priority and a deadline. The authors utilize a query-based scheduler that optimizes the Hit-Value Rate of critical CQs and the Deadline Missed Rate of the rest. The scheduler first attempts to meet all the deadlines specified by the user. If that is not possible, then it schedules according to both the priority and the deadline of the query. This is different from our scheduler, in which there are no user-specified deadlines.

The work in MavHome [7, 14] uses a runtime optimizer that monitors how the system meets or violates the QoS specified by the user. When a QoS measure is violated, the runtime optimizer analyzes the available options for scheduling and/or load shedding. It considers the QoS per CQ. It utilizes a two-level scheduler inside the runtime optimizer where the top level is a dispatcher and the lower level contains different schedulers each aimed at optimizing a metric. The different QoS metrics such as memory and latency are assigned priority. This differs from our system where the CQs are assigned priorities and the goal is to optimize the response time of critical CQs. MavHome does not consider the priority of the CQ, but rather uses QoS graphs similar to the Aurora model.

Along these lines, in DILoS [40], we have explored the synergy between the CQ scheduler and the load shedder where the priorities are also considered for both modules. However, that work is outside the scope of this thesis.

3.4 MULTI-CORE SCHEDULING

Multi-core scheduling has been a subject of study in the field of operating systems and real-time systems for a long time. Works such as [32] and [11] attempt to allocate cores or threads to tasks in a way that is fair and enables load balancing. Other schemes utilize a global queue in a similar fashion to the MBD scheduler however with the objectives of optimizing the overall system performance such as [28].

Several papers have looked into scheduling in a multi-core environment in the database community. There have been multiple papers that focused on optimizing different types of queries to

run in multi-core environments. The work in [44] evaluated streaming aggregates over different multi-core architectures. With regard to joins, the work in [22] looked at executing stream joins over a cell processor. Also for joins, the work in [10] focuses on designing new algorithms for in-memory join operators in multi-core environments. Besides optimizing specific operations, some papers focused on parallelizing the query optimizer [27]. Also in the field of query optimization in multi-core environments, the work in [3] describes a new operator called ASYNC that allows portions of the query plan to be parallelized. The work in [19] proposes a cooperative locking paradigm for parallelization of frequency counting over data streams. The idea is to delegate work on a shared resource to the thread currently holding the lock on that resource. In the field of Business Intelligence Databases, the work in [41] utilizes the idea of cache affinity to improve the performance of business intelligence queries in a multi-core environment. The authors in [23] propose a scheduling policy that improves utilization and lowers execution time for metric space queries. However, none of the mentioned papers considers class-aware schedulers where each class has an assigned numerical priority. Rather they devise schedulers that optimize one or more types of queries to utilize the system resources efficiently.

In [29], the authors examine the problem of sharing among queries if a database system is running in a multi-core environment. While this is very relevant to our work, it is only one of its subproblems. The paper does not consider queries with different priorities and the main focus is on DBMS queries. In this thesis, we handle the sharing of operators among CQs within the goals of providing CQ-class-aware performance.

3.5 PRIORITY INVERSION

In the field of real time systems [46], priority inversion happens when a higher priority task is waiting for a lower priority task to finish executing. The work in [43] considers the problem when a higher priority job is blocked waiting for a resource held by a lower priority job. The authors propose using priority inheritance for critical sections of lower priority tasks so as they would release their locks on the shared resources faster. The Windows scheduler uses a similar approach to solve priority inversion by boosting the priority of a lock-holding thread [35]. The work in

[57] also resolves priority inversion in spin-locks by using priority inheritance. Later works [17] extend the priority inheritance model in real time systems when there are more complex resource access policies. In other real time systems such as the work in [51], a priority ceiling is used to determine if a task is allowed to lock a resource or not based on its priority and the priorities of other lock-holding tasks.

In this thesis, priority inheritance is used to promote a whole CQ when it shares operators with a higher priority class in a similar fashion to real time systems. However, priority inversion here refers to a lower priority CQ having a lower/better response time than a higher priority CQ.

3.6 SUMMARY OF RELATED WORK

In this chapter, we surveyed existing scheduling policies in computing systems, DBMSs, DSMSs and multi-core environments. We discussed how these schedulers do not meet the requirements set forth in our thesis statement. We also surveyed work on priority inversion in real time systems and distinguished it from priority inversion under our system model. In the next three chapters, we design and evaluate our proposed schedulers for single-core, dual-core, and multi-core DSMSs respectively.

4.0 SCHEDULING FOR A SINGLE-CORE AQSIOS

In this section, we describe the Continuous Query Class (CQC) scheduler for a DSMS running in a single-core environment. We show the evaluation of our scheduler in Section 4.2.

4.1 CONTINUOUS QUERY CLASS (CQC) SCHEDULER

In a single-core environment, more specifically in a single thread system, the scheduler allocates resources to the source operators as well as the operational and the output operators. To schedule CQs that belong to different CQ-classes, we propose a two-level scheduler, namely, Continuous Query Class (CQC) scheduler (Figure 3), that combines the Weighted Round Robin (WRR) scheduler (level 1) and the HR scheduler (level 2).

On level 1, a WRR scheduler (Algorithm 1) allocates to each query class i a quota (c_i) equal to a time slice of T_i time units. T_i is the product of the priority of query class i (P_i) and a configurable time period k divided by the sum of all class priorities ($\sum_{j=0}^{n} P_j$). For example, in a system with three query classes $\{H, C, N\}$ with priorities of $\{6, 3, 1\}$ and $k = 20$, WRR assigns weighted quotas of $\{12, 6, 2\}$ to the query classes $\{H, C, N\}$, respectively.

Level 2 consists of a set of HR schedulers. Each HR scheduler is responsible for a set of operators that belong to a specific class. In AQSIOS under the single-core system model (Section 2.2.1), each HR class scheduler distinguishes between source and operational operators and schedules them independently. It utilizes a separate queue for the source operators called Source Queue (SQ) as shown in Figure 3. For the operational and output operators, it uses the Operational Queue (OQ) also shown in Figure 3. Recall that HR [47] is designed to minimize the average response time. In particular, HR sets the priority of an operational operator x of a CQ to be $\frac{S_x}{C_x}$, where S_x is

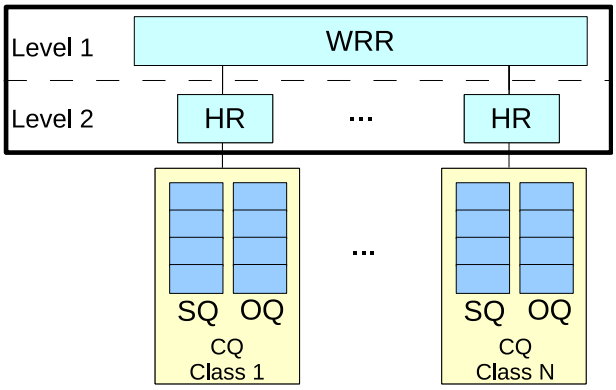

Figure 3: CQC Scheduler components in a single core environment

the expected selectivity and C_x is the expected processing cost of the query fragment downstream from the operator x. The priority of an output operator is set to the priority of its input operator. Thus, the OQ is implemented as a priority queue sorted on these HR priorities. At each scheduling point, an HR class scheduler first checks the OQ for any ready-to-execute operational or output operators. It switches to schedule the source operators in the SQ only when all the operators in the OQ have empty input queues and thus are not ready to execute.

In AQSIOS, the WRR scheduler does not preempt the HR class schedulers but uses a negative credit system (Steps 2 and 3 in the execution phase of Algorithm 1). If any HR class scheduler i exceeds its quota (c_i), the WRR scheduler deducts the excess amount from its future quotas. This, for example, reduces the effect of heavy loaded low-priority class N CQs on high-priority class H CQs. However, unused time quotas do not carry on. If an HR class scheduler is waiting for input tuples, it returns control to the WRR scheduler after polling its SQ twice.

The CQC scheduler isolates the operators in different CQ classes. This guarantees a time-slice $k \cdot P_i / \sum_{j=0} P_j$ for operators in class i during each round. Hence, the isolation of operators among classes reduces the probability of starvation. In our example, high-priority operators in classes H and C, cannot starve low-priority operators in class N. The latter compete only with high-priority operators in their own class. Priority class isolation also guarantees a high probability of execution for all CQs in class H because their operators do not compete with high-priority operators in class

Algorithm 1 CQC Scheduling Algorithm: Level 1

INPUT: a set of HR schedulers where each scheduler is responsible for a set of all operators belonging to a specific class.

Setup Phase

1. Sort the scheduler set in decreasing order of class priorities.
2. For each scheduler i, calculate its ideal time slice ($T_i = P_i \times k \,/\, \sum_{j=0} P_j$) and its quota ($c_i = T_i$).

Execution Phase

1. Select the next scheduler i from the scheduler set.
2. If i has nonpositive quota c_i, then Step a, else Step b.
   a. Add T_i to c_i. Go to Step 1.
   b. Schedule i to run for c_i time units.
3. After i returns control, determine the number of time units (tu) it executed for. If $tu \leq c_i$, go to Step a, else Step b.
   a. Reset c_i to T_i. Go to Step 1.
   b. Set $c_i = T_i - (tu - T_i)$. Go to Step 1.

C or N CQs. That is, the CQC scheduler ensures no operator class priority inversion and is suitable for our target applications.

4.2 SINGLE-CORE EXPERIMENTAL EVALUATION

In this section, we evaluate the CQC scheduler under AQSIOS, our DSMS prototype, running on a single core.

Table 1: Experimental Setup for CQC Evaluation

System Parameters	
k (time period)	1000 micro second
Time unit	1 micro second
Query Load Specifications	
Types of CQs	Select, Aggr, 2-way Joins
Window Size	10 time units
Selectivity of Selections	$[0.25 - 1]$
Data Stream Specifications	
Humidity (int)	Uniform $[0 - 100]$
Temperature (int)	Uniform $[0 - 40]$
Location (12 chars)	20 locations
Number of input streams	27
Tuple arrival rate/stream	1500 tuples/second
Number of tuples/stream	10,000

4.2.1 Experimental Setup

To evaluate the CQC scheduler, we implemented the HR and CQC schedulers in AQSIOS. AQSIOS is based on the STREAM source code, but is extended to include scheduling policies, a statistics collection module and a load shedding module (which is disabled in this work in order to isolate and study the effects of scheduling). We compared the performance of the CQC scheduler to both the HR and the STREAM DSMS Round Robin (RR) schedulers. Under the RR and HR schedulers, each operator processes all the tuples in its input queue before returning control back to the scheduler. Under HR scheduling, the operators are scheduled in non-increasing order of output rate. For the CQC scheduler, we chose the value of the scheduling period k to be 1000 micro seconds so that the lowest priority class is not overloaded. Tables 1 shows the configuration parameters of the system.

Workloads: We evaluated the CQC scheduler using three workloads: A, B and C shown in Table 2. Each workload has three classes: Class 1, Class 2 and Class 3. We also ran the HR scheduler with three additional workloads A_H, B_H and C_H that only contain Class 1 CQs out of workloads A, B and C, respectively. All workloads consist of nine aggregate CQs, six 2-way join CQs and six selection CQs. We have experimented with various query sets and we chose this one because it brings the system to a highly loaded state without overloading it. Table 2 shows the number of CQs and the priorities of each CQ class under each workload. The CQs in workload A are identical in each class while workloads B and C have more CQs in classes 2 and 3. Workloads B and C have an identical query load and differ only in the priority of the CQ classes. We chose the CQs in workloads B and C to satisfy the assumptions made concerning the applications we are considering (i.e., critical monitoring and scientific exploration).

Data Input: We assume a wireless sensor network bandwidth of 250Kbps [62]. Our tuples consist of 3 attributes: location (12 characters), humidity (int) and temperature (int) measurements. The network was simulated by injecting the system with 10,000 tuples per input stream as read from a file, at the highest assumed network bandwidth (1500 tuples/second). We simulate no operator sharing across CQ classes by duplicating shared input streams and source operators.

Metrics: We measure the average response time for each workload query class and scheduler.

Platform: We ran AQSIOS on a Dell Inspiron E1405 laptop with Intel Core Duo processor T5500 @ 1.66 GHz with 1 GB of RAM running Fedora 10 with the 2.6.27 Linux kernel. AQSIOS executes on only one core of the CPU. We measure response times using the PROCESS_CPUTIME clock, which measures only the process active execution time.

4.2.2 Experimental Results (Figures 4-7)

Figure 4 compares the average response times of Class 1 CQs for all schedulers on workloads A, B, and C (Table 2) and for the HR scheduler on workloads A_H, B_H, and C_H. In the latter case, Class 1 CQs are running by themselves, i.e., there are no CQs from Class 2 or 3. We introduce these workloads to compare with the ideal case. When executing the CQs of all the classes in the workload, the CQC scheduler improved the average response time of Class 1 queries as compared to the HR scheduler by a factor of at least 9.4 (the y-axis in Figure 4 is in logarithmic scale). Class 1

Table 2: Workloads for CQC Evaluation

Workload A			
	Class 1	**Class 2**	**Class 3**
Number of CQs	7	7	7
Types of CQs	all	all	all
Priorities	6	3	1
Input rate (tup/sec)	1500	1500	1500
Workload B			
	Class 1	**Class 2**	**Class 3**
Number of CQs	2	8	11
Types of CQs	join	all	all
Priorities	3	2	1
Input rate (tup/sec)	1500	1500	1500
Workload C			
	Class 1	**Class 2**	**Class 3**
Number of CQs	2	8	11
Types of CQs	join	all	all
Priorities	6	3	1
Input rate (tup/sec)	1500	1500	1500

CQs running by themselves have an even lower response time because the system is under-loaded.

Figure 5 shows the average response times of CQ classes under all the schedulers for workload A. As expected, the CQC scheduler improves the average response time of Class 1 CQs, but impairs the response time of Class 3 CQs. Class 2 CQs have a higher average response time for this workload because they constitute one third of the system CQs and they have a high output rate. As a result, when their HR scheduler executes, Class 1 CQs always have tuples to process and do not return before their quota T_1 ends.

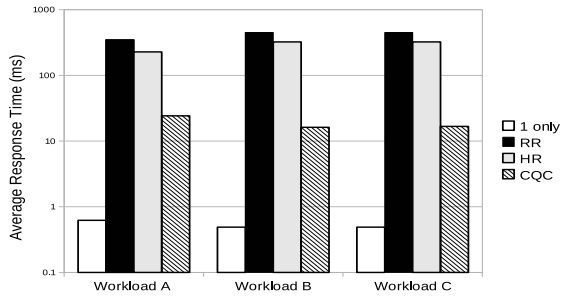

Figure 4: Average response time for Class 1 queries (y-axis is in log-scale). "1 only" are the CQs of Class 1 by themselves under HR.

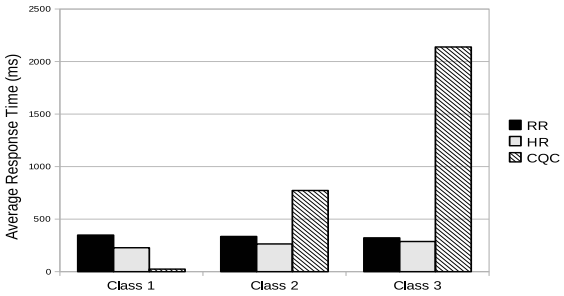

Figure 5: Average response time for Workload A: CQC provides 9.4 times improvement on Class 1 v.s. HR.

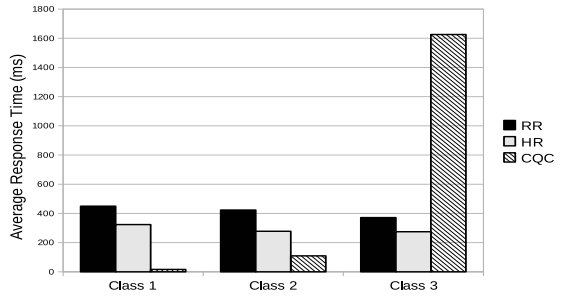

Figure 6: Average response time for Workload B: CQC provides 19.8 times improvement on Class 1 v.s. HR.

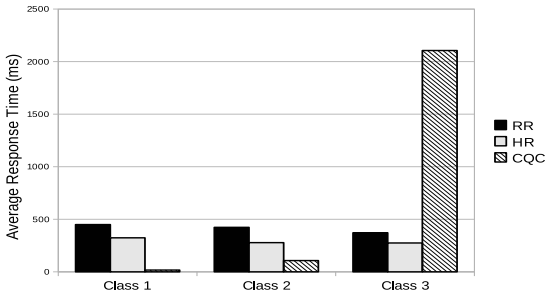

Figure 7: Average response time for Workload C: CQC provides 19.3 times improvement on Class 1 v.s. HR.

CQC improves the average response times of Class 1 and Class 2 CQs for both B and C workloads (Figures 6 and 7), but impacts negatively the performance of Class 3 CQs. In contrast with workload A, Class 1 CQs are fewer, so they do not affect the performance of Class 2 CQs in workloads B and C, which are a set of more realistic workloads for the motivating applications we are considering. The response time of Class 3 CQs under Workload B is lower than under Workload C because they are assigned a lower relative priority (1/10 vs. 1/6) under Workload C.

4.3 SUMMARY OF CQC SCHEDULING

In this chapter, we described the design of the CQC scheduler than we evaluated it under AQSIOS. Overall, our scheduling policy CQC does extremely well in single-core DSMSs. Our experimental evaluation shows that the CQC scheduler improves the response time of the most important CQs, and penalizes the data-gathering CQs (i.e., of low priority) whose results do not need to be handled immediately.

5.0 SCHEDULING FOR A DUAL-CORE AQSIOS

In this chapter, we first describe how the CQC scheduler is inadequate for a DSMS running in a dual-core environment in Section 5.1. Then, we describe the ABD scheduler which is designed to solve the limitations of the CQC scheduler in a dual-core environment in Section 5.2. We evaluate the ABD scheduler in Section 5.3.

5.1 CQC IN A DUAL-CORE ENVIRONMENT

CQC works very well when the system is running on one core. However, it has many side-effects under a dual-core architecture.

First, the HR scheduler is proven to optimize the average response time of a set of CQs when they have equal priority. However, once it is used in a dual-core environment on the lower level (level 2), it has higher chances of starving the operators at the end of the priority queue. *Starvation* happens when the size of the scheduling period is not enough to execute all of the operators in a class in one round. In this case, the semantics of HR's priority queue are violated because there is an interruption of execution when control is switched to another HR scheduler. During this interruption, given that the source operators are executing on another core, new tuples could arrive at the operators at the head of the priority queue. Thus even when the input rate is low, the operators at the end of the priority queue might not execute for many rounds.

Second, we may have *priority inversion* between the CQ classes where the operators at the head of the priority queues of all the classes could execute more often than the operators at the end of the priority queue of the most critical class.

Third, CQC relies heavily on the size of the scheduling period K which is a manual input to

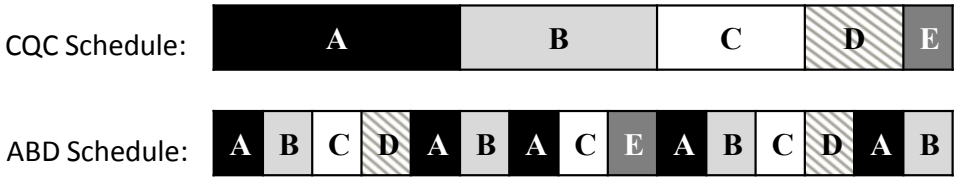

Figure 8: An example of 2 schedules under ABD and CQC

the scheduler. As mentioned previously, the size of the scheduling period allocated to a scheduler is very important. A small value for K could result in *starvation*. A large value of K could result in the scheduler degenerating into behaving the same as a RR scheduler. Finding the perfect value of K ahead of time is not trivial because it depends on the load in the system which can change during runtime.

Last, CQC increases the wait time and thus the response time of the tuples. All the HR schedulers including the critical class HR scheduler have to wait for all the other schedulers to use their allocated time in order to execute. This is especially bad when the load in the lower priority classes is considerably heavier than the load in the critical classes because the latter have to wait a long time in order to execute.

5.2 ADAPTIVE BROADCAST DISK (ABD) SCHEDULER

To solve all the above limitations with the CQC scheduler, we propose a novel scheduling policy called the ABD (Adaptive Broadcast Disks) scheduler. Like CQC, ABD is also a two-level scheduler with one scheduler per class on the lower level and one scheduler on the higher level that determines which class to run next. Similar to the CQC scheduler, the ABD scheduler attempts to allocate to each class an amount of time proportional to its priority. However, the ABD scheduler splits the amount of time allocated to a per-class scheduler into *quota slices*, i,e., equal time intervals. In order to reduce the waiting times, the ABD scheduler builds a schedule so that a per-class scheduler gets a number of quota slices distributed over the entire *scheduling period*.

Algorithm 2 ABD Schedule Generating Algorithm

INPUT: a set of priorities where each priority is associated with a CQ class.

OUTPUT: a schedule S where each class has a number of timeslices proportional to its priority.

1. Find the minimum power of two mp that represents all the priorities.
2. Generate all the powers of two from 0 until 2^{mp}.
3. Generate all the inverses of the powers in binary and store them as a masklist.
4. Find out which combinations of powers of two each mask is formed of.
5. Associate each class with the corresponding masks.
6. Iterate from 0 to $mp - 1$.
   - If iterator matches the mask generated then add to schedule S all the classes that are associated with the mask
7. Return S.

The ABD schedule is built using binary manipulation where the priority is converted to a bitmap as shown in Algorithm 2. The bitmap is broken down into individual bits. Then, the bits are compared against pre-generated masks of powers of two to determine when to place a class in the schedule. Figure 8 shows an example schedule under the CQC and ABD schedulers for a workload with five classes: A, B, C, D and E with priorities 5, 4, 3, 2 and 1 respectively. The CQC scheduler allocates all of the quota slices assigned to Class A at once. The ABD scheduler assigns Class A the same amount of time, but reduces its wait time by having it wait for a maximum of four quota slices before executing versus the equivalent of ten quota slices under the CQC scheduling.

To avoid starving operators within a class, the ABD scheduler uses a round robin scheduler (RR) on the lower level instead of an HR scheduler. Each RR scheduler is responsible for scheduling operators in its assigned class. It avoids going over the quota slice by as little as possible. Thus, before it schedules an operator, it estimates the amount of time it takes to execute all the tuples in its queue. If the time left until the end of the quota slice is not enough, then the RR scheduler determines how many tuples the operator can execute without exceeding the allocated quota slice. In cases when the time left is not enough to process any tuples, then the RR scheduler records which operator it stopped at and returns control back to the ABD scheduler. When the RR scheduler is

Algorithm 3 ABD Scheduling Algorithm: Level 1 during runtime

INPUT: A set of RR schedulers SRR and their schedule S

```
Get next sched_index
if one round completed then
    if quota_increased is FALSE and quotaslice > last_quota_increase then
        decrease quotaslice by last_quota_increase and assign new quota to all schedulers
    end if
    for all schedulers i in SRR do
        if i's average response time > i + 1's average response time then
            if i + 1's priority > 1 then
                decrease it by 1 and set pr_changed = True
            else
                increase i's priority by 1 and set pr_changed = True
            end if
        end if
    end for
    if pr_changed is TRUE then
        generate a new schedule
    end if
end if
Schedule the RR scheduler whose sched_index is selected
if quotaslice was not enough to execute all the tuples in the last operator's queue then
    calculate time_needed to execute those tuples and increase the quotaslice by amount needed
    set quota_increased = TRUE and last_quota_increase = time_needed
end if
```

scheduled next, it starts executing at this operator. This ensures fairness to all the operators in the class and avoids priority inversion between operators at different positions among classes. At the end of its execution, each RR scheduler records the amount of time it needed to finish executing all the tuples in the last operator's queue. This amount of time is utilized by the ABD scheduler as

shown in Algorithm 3 to adjust the size of the timeslice dynamically during runtime.

5.2.1 Dynamic Quota Slice

After running experiments with the CQC scheduler, we realized that its performance is highly dependent on the size of K. Measuring the workload during runtime is a costly process to determine the correct value. In order to keep the scheduling overhead small, the ABD scheduler monitors the amount of time each RR scheduler runs for. If a per-class RR scheduler goes over its quota slice, the ABD scheduler uses this as an indication that the quota slice is too small and needs to be increased. The quota slice is then incremented by the amount over the quota slice plus the amount of time the RR scheduler needs to finish all the tuples at the last operator. This new quota slice is then allocated to all the RR schedulers.

However, the workload could fluctuate during runtime and the system could end up with a quota slice that is too big such that the schedule is not used anymore and it is essentially running round robin also on the top level which would be unaware of class priorities. To avoid this situation, the ABD scheduler attempts to decrease the size of the quota slice if it was not increased in the last schedule. It decreases gradually by the same amount it was last increased with. This insures that the quota slice would adjust in small steps both in increasing and decreasing directions.

5.2.2 Response Time Monitoring

It is important to respect the priorities throughout execution even in the presence of fluctuations in the workload. Toward this, the ABD scheduler monitors the average response time of each class to detect any priority inversion. Whenever priority inversion is detected, the priorities are adjusted in order to correct it. Priority inversion happens when the priority of Class A is larger than that of Class B but the response time of Class A is higher than that of Class B. In this case, the ABD scheduler would decrease the priority of B if it is greater than 1. If B's priority is equal to 1, then the ABD scheduler would increment the priority of A by 1. The ABD scheduler then generates a new schedule. The process of generating a new schedule and detecting priority inversion is costly. Thus, this is done only as needed at the end of the schedule and average response times are used to guard against sudden fluctuations and frequent schedule rebuilding as shown in Algorithm 3.

5.3 EXPERIMENTS EVALUATING THE ABD SCHEDULER

The newest version of AQSIOS which runs in a dual core environment is still under development. Thus, to evaluate the ABD scheduler, we developed a simulator called SimAQSIOS using the SimPy Simulation Package [50]. SimAQSIOS models all the operators that are supported by AQSIOS.

The selectivities of most of the operators are implemented probabilistically. Some of the operators such as the selection, union and except operators are implemented in order to mimic their behavior as close as possible. The cost of the operators, i.e., the amount of time it takes to execute an operator and its tuples as well as the scheduling and the statistics collection overhead is modeled after careful profiling of AQSIOS. The cost of executing an operator takes into account the cost to call the operator's run function and set it up, which is the cost the system pays regardless of how many tuples it executes. It also considers the cost of executing each tuple inside the operator. We added fluctuations of +/- 5% on average to all the costs to model real system fluctuations.

We implemented under SimAQSIOS: the ABD, CQC, and Weighted HR schedulers. The Weighted HR scheduler (WeHR) assigns to each operator a priority equal to the product of its output rate and the priority of the class it belongs to.

5.3.1 Experimental Setup

Tables 3, 4, and 5 show the configuration parameters for SimAQSIOS, the schedulers and the workloads, respectively.

Scheduler-specific parameters: For the CQC scheduler, we choose the value for K to be 30,000 so that none of the classes is starving or overloaded. For the ABD scheduler, we choose an initial quota slice size of 50. We show later in this section that this initial value has no effect on the performance of this scheduler.

Workloads: We evaluate ABD using four workloads shown in Table 4. Workloads D, E and F have three query classes. Workload E has the same identical CQs in each class. Workloads E and F have the same CQs, but the CQs for Class 1 under E are assigned to Class 3 under F. Workload 5G has five query classes; the heaviest class is Class 3. We choose the input rates for the streams so

Table 3: Experimental Setup for ABD Evaluation

CQ Load Specifications	
Types of CQs	Select, Aggr, 2-way Joins
Window Size	10 time units
Selectivity of Selections	$[0.25 - 1]$
Data Stream Specifications	
Humidity (int)	Uniform $[0 - 100]$
Temperature (int)	Uniform $[0 - 40]$
Location (12 chars)	20 locations
Number of tuples/stream	10,000
Scheduler Parameters	
ABD initial quota slice	50 micro seconds
CQC scheduling period k	30,000 micro seconds

that the system is loaded, but not overloaded. We choose the workloads so that they have a variety of priorities and load distributions among classes.

Metrics: We report the overall weighted average response time (Wavg) across all the classes and the average response time of the CQs in each class.

Overheads: All of the overheads including the context switches to schedulers and to operators are part of the response time calculation to better compare the schedulers.

Data Input: Our tuples consist of three attributes: location (12 characters), humidity (int) and temperature (int) measurements. The input streams were simulated by injecting the system with 10,000 tuples per input stream.

Table 4: Workloads D, E and F Specifications

	Workload D			**Workload E**			**Workload F**		
	Class1	**Class2**	**Class3**	**Class1**	**Class2**	**Class3**	**Class1**	**Class2**	**Class3**
Number of CQs	7	7	7	2	8	11	11	8	2
Types of CQs	all			join	all	all	all	all	join
Priorities	30	20	10	60	30	10	60	30	10
Input rate	1600 (tup/sec)			1500 (tup/sec)			1500 (tup/sec)		

Table 5: Workload 5G Specifications

Workload 5G					
	Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**
Number of CQs	3	5	7	6	4
Types of CQs	join & aggr	all	all	all	all
Priorities	50	40	30	20	10
Input rate	1200 (tup/sec)				

5.3.2 Experimental Results

Experiment 1 - Comparison of WeHR, CQC, and ABD (Figure 9): Figure 9 shows the performance of the schedulers while running Workload D. Notice that the y-axis is in log-scale. The ABD scheduler lowers the weighted average response time (Wavg) by 12.16% compared to the CQC scheduler and by 26 times compared to the WeHR scheduler. Even though the WeHR scheduler improves the response time of Class 1 by 10%, this comes at the expense of starving Class 3 making its response time over two seconds. The WeHR scheduler has similar performance for all

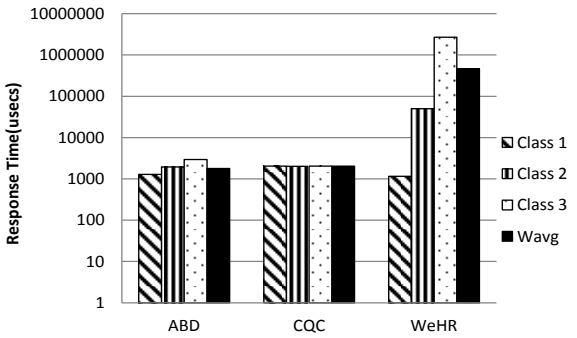

Figure 9: Workload D: Wavg under ABD is lower by 12.16% than under CQC. Class 1's response time is 36.6% lower under ABD.

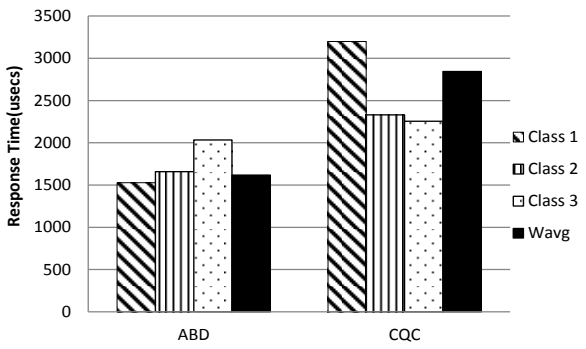

Figure 10: Workload E: Wavg under ABD is lower by 43.1% than under CQC. Class 1's response time is 52.2% lower under ABD.

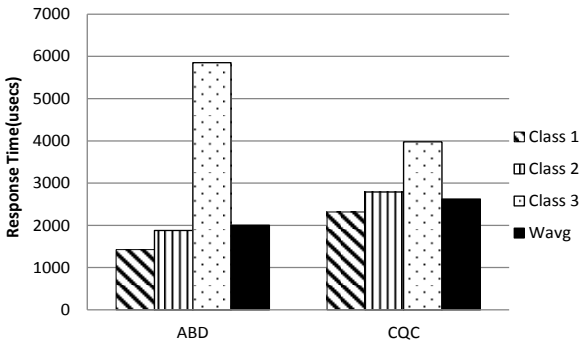

Figure 11: Workload F: Wavg under ABD is lower by 23.7% than under CQC. Class 1's response time is 38.6% lower under ABD.

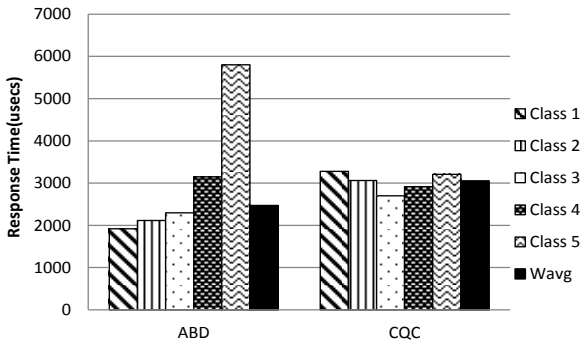

Figure 12: Workload 5G: Wavg under ABD is lower by 19.1% than under CQC. Class 1's response time is 41.5% lower under ABD.

of the workloads we ran, so we omit it from further graphs to improve their readability. The CQC scheduler for Workload D runs close to how round robin would run in that the response times of the three classes are very close. The ABD scheduler on the other hand, starting from a very low quota slice adapts to the workload and respects the priorities of the three classes without starving Class 3.

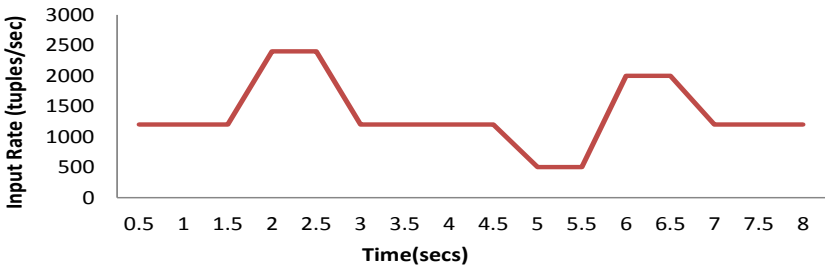

Figure 13: Adaptive Experiment: the input rate used for Class 1

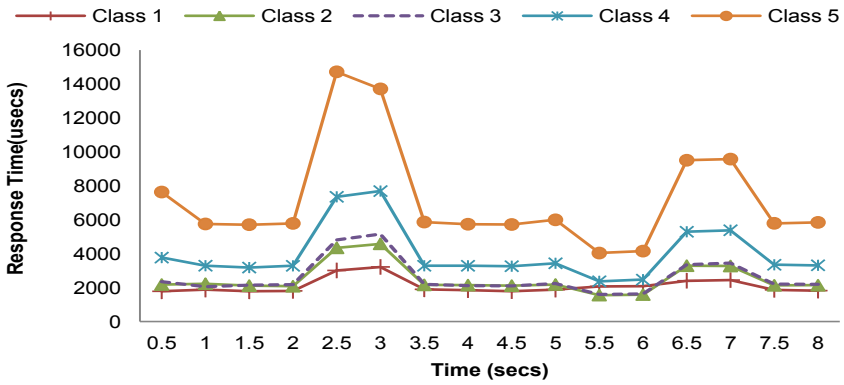

Figure 14: Adaptive Experiment: ABD adapts as the input rate of Class 1 changes

Experiment 2 - No Priority Inversion (Figures 10, 11, 12): Figures 10 and 11 show Workloads E and F under ABD and CQC scheduling. In both workloads, the ABD scheduler optimizes Wavg (43.1% for E and 23.7% for F) and improves the response time of Class 1 by 52.2% for E and 38.6% for F over the CQC scheduler. Even worse, the CQC scheduler results in priority inversion for workload E due to the accumulated wait time for Class 1. The ABD scheduler by slicing the quota reduces the wait time for this class and maintains the relative priorities of the classes.

Figure 12 shows the performance of the CQC and ABD schedulers while running Workload 5G. The CQC scheduler again results in priority inversion for classes 1 and 2. In contrast, the ABD

Table 6: Sensitivity Analysis

Initial Quota Slice	20	50	100	500	1000
Wavg	1749	1788	1776	1775	1768

scheduler preserves these priorities with 5 classes and improves Wavg by 19.1% and the response time for Class 1 by 41.5%.

Experiment 3 - Sensitivity to Initial Quota Slice (Table 6) : In order to verify that the ABD scheduler is not dependent on the initial value for the quota slice, we ran Workload E with different initial quota slices. Table 6 shows the average Wavg for each value for the quota slice. Wavg stayed more or less constant throughout the runs and the slight differences are due to normal fluctuations in the system.

Adaptive Experiment (Figures 13, 14) : To evaluate how well the ABD scheduler adapts to changes in the workload, we modified the input stream for Class 1 under Workload 5G so that it follows the input rate shown in Figure 13. The results are shown in Figure 14. The ABD scheduler adapts fairly well to the changes in the input rate by keeping the response time for Class 1 lower than all the other classes except for a brief period between 5.5 and 6 seconds. The overall average response times for the classes 1, 2, 3, 4 and 5 are 2.2, 2.46, 2.6, 3.8 and 6.9 msecs respectively. Consequently, the ABD scheduler preserves the priorities of the classes even in the presence of changes in the workload.

5.4 ABD SCHEDULING SUMMARY

In this chapter, we proposed a new scheduler namely the Adaptive Broadcast Disks (ABD) scheduler for dual-core DSMSs. We described the design and evaluated the ABD scheduler showing that it provides better response time for high-priority classes without starving low-priority classes. It also prevents priority inversion and adapts to changes in the workload during runtime.

6.0 SCHEDULING FOR A MULTI-CORE AQSIOS

As discussed in Section 2.2.3, eventually AQSIOS needs to run in a multi-core environment. One idea is to use an ABD scheduler per core to schedule the CQs. However, this is problematic because the load between the cores has to be perfectly balanced such that none of the cores is overloaded. Achieving a perfectly balanced load among the different cores in the presence of CQ classes while satisfying this thesis's initial problem statement has turned out to be a challenging task. However, building on our experience building the ABD and the CQC schedulers, we design a new multi-core CQ scheduling policy that prevents starvation, respects priorities by preventing priority inversion and uses the system resources in an efficient manner.

6.1 MBD SCHEDULER

To address the issue of scheduling CQs belonging to different classes, we propose the Multi-core Broadcast Disks scheduler (MBD). The MBD scheduler is a two-level scheduler similar to the CQC and ABD schedulers. However, the MBD scheduler is a CQ-level scheduler unlike the CQC and the ABD schedulers which are class-level schedulers. The CQC and ABD schedulers isolate the CQs within the same class and then assign each class a level 2 scheduler. Using classes, as the minimum schedulable unit, is inefficient in a multi-core environment because the number of classes is often less than the number of cores. Thus, some of the cores will be idle with nothing to execute while other cores are overloaded. However, a CQ-level scheduler improves parallelization because the number of CQs is larger than the number of classes. Consequently, it can utilize all the cores allocated to the system in a much better manner. For these reasons, the MBD scheduler is designed as a CQ-level scheduler.

6.1.1 MBD schedule

Just like the ABD scheduler, the MBD scheduler utilizes an MBD schedule which is built using the same manner as the ABD schedule described in Section 5.2. However, instead of having one scheduling point (one thread or core for the operational and output operators), it has multiple scheduling points (multiple threads or cores). In addition, the ABD scheduler is built to allocate quota slices to query classes. On the contrary, the MBD scheduler is built to schedule CQs instead of query classes. Otherwise, the MBD schedule is generated using the ABD schedule generating algorithm (see Algorithm 2). The following section describes how the CQ priority is calculated.

6.1.2 CQ priority

In order to build an MBD schedule for the CQs in the system, we need to determine the priority of each CQ based on its class priority and its load. The priority of CQ_i is calculated according to the following formula:

$Pr_i = Pr_{class(i)} \times \frac{cost(i)}{cost(class(i))} \times multiplier$

Where cost(i) is the expected cost per tuple of CQ_i. Cost(class(i)) is the expected cost per tuple of all the CQs that belong to the same class as CQ_i. $Pr_{class(i)}$ is the priority of the class CQ_i belongs to.

The priority of a CQ might be less than 1.0 when there are a lot of CQs in a class or when a CQ is very light. In order to generate a correct MBD schedule, the minimum priority of any CQ should be 1.0 so that all the CQs get to execute at least once per schedule. In order to avoid having a CQ priority which is less than one, we multiply all the priorities of CQs by a multiplier such that the minimum priority is 1.0. The multiplier is calculated according to the following formula:

$\forall CQ_i, multiplier = \max_{\lceil value_i \rceil}\{value_i = \frac{cost(class(CQ_i))}{Pr_{class(CQ_i)} \times cost(CQ_i)}\}$

This multiplier not only makes sure that the minimum priority of a CQ is 1.0 but also that the schedule is the shortest possible. A short schedule is desirable especially when the scheduler needs to adjust and adapt. However, it carries the danger of losing precision in the priorities. For example, assume CQ_i has priority 1.7 and CQ_j has priority 1.3. If we apply a simple rounding function to the priorities, then the priorities of CQ_i and CQ_j become 2 and 1 respectively making CQ_i twice as important as CQ_j which does not preserve the original priority semantics. If we

apply a floor or a ceiling function, then both CQs would be equally important which also violates the original priority semantics.

To solve this problem, we introduce a probabilistic component in the execution of the schedule as follows. We apply a ceiling function to all the priorities and store the fraction portion separately. In our example, both CQ_i and CQ_j get 2 slots. However, $fraction_i$ is 0.7 while $fraction_j$ is 0.3. We generate the MBD schedule using the ceiling priorities. Then, to preserve the original priority semantics, the first slot in the schedule when CQ_i is selected, the MBD scheduler generates a random number between 0 and 1. If this number is less than the $fraction_i$ then CQ_i executes, otherwise the scheduler selects the next CQ in the schedule to execute. This preserves the original priorities and generates a short schedule without preempting the CQ execution during runtime.

6.1.3 CQ Execution

During runtime, when one of the cores becomes idle, its thread retrieves the next CQ to execute from the MBD schedule. If this CQ is already running, then it skips over to the next one and raises a flag in the MBD schedule marking that this CQ needs to be scheduled another round. The reason for this action is to prevent having a CQ run on two cores at the same time while increasing data and core locality at the cores. Once the core selects a CQ to execute, it copies the tuples from the source operator and places them in the input queue of the first operator Op_i^1 and starts executing it. Any tuples that arrive after Op_i^1 is scheduled are not processed in this round. CQ_i is executed starting from Op_i^1 to the output operator only once before the thread returns to select another CQ from the schedule. If another scheduler marked CQ_i to execute another round, then the thread executes CQ_i again starting from Op_i^1.

In order to avoid two threads selecting the same CQ to execute, the thread needs to acquire a lock on the MBD schedule before selecting a CQ and marking it as scheduled. Before it starts executing the selected CQ, it unlocks the schedule so that other threads can select their next CQ.

6.1.4 Shared Operators

Sharing operators or operator paths allows for more efficient execution and better utilization of the system resources. However, the MBD scheduler is a CQ-level scheduler. This creates a problem

Algorithm 4 Decrease Violator Priority Algorithm

INPUT: A set of CQ classes SC sorted in non-increasing order of priority

```
1: for all CQ classes i in SC do
2:    if i's response time > i + 1's response time and i's priority > i + 1's priority then
3:       if i's running frequency > 1 then
4:          decrement i + 1's running frequency by 1
5:       else
6:          increment i's running frequency by 1
7:       end if
8:    end if
9: end for
10: for all CQ classes i in SC do
11:    if running frequency of i = running frequency of i+1 and i's priority > i+1's priority then
12:       increment i's running frequency by 1
13:    end if
14: end for
15: if running frequencies did not change then
16:    multiply all running frequencies by 2 and rerun algorithm
17: end if
```

when two or more CQs share an operator or an operator path. In these cases, we combine the sharing CQs into one CQ and calculate a combined CQ priority. The combined CQ priority of two CQs: CQ_i and CQ_j is:

$$Pr_{ij} = Pr_i + Pr_j$$

This guarantees that during runtime, the two CQs would not be competing for the execution of the shared operator path. Also, by adding the two priorities, the CQ_i and CQ_j would still receive the same priority as if they were separate without synchronizing the shared path.

Algorithm 5 Increase Violator Priority Algorithm

INPUT: A set of CQ classes SC sorted in non-increasing order of priority

1: **for all** CQ classes i in SC **do**
2: **if** i's response time $> i+1$'s response time and i's priority $> i+1$'s priority **then**
3: increase i's running frequency by 1
4: **end if**
5: **end for**

6.1.5 Priority Inversion Module

Similarly to the ABD scheduler, the MBD scheduler is susceptible to priority inversion and solves it using a priority inversion module. Priority inversion is detected when a higher priority class has a higher response time than a lower priority class. In this thesis, we do not label the case when a two classes of different priorities have the same response time as priority inversion (see Section 2.3.2). However, if desired by the administrator, the following priority inversion modules can be modified to detect that as well.

At the end of each schedule round, the MBD scheduler checks if there is priority inversion among the classes. When priority inversion is detected, the MBD scheduler corrects it by adjusting the running priorities or the frequencies of the classes and using these frequencies to rebuild the schedule. The CQ scheduler assigned the last CQ in the schedule performs this task. We identified four ways in which the frequencies can be adjusted to correct against priority inversion. All of these ways use additive updates to the frequencies instead of multiplicative updates in order to avoid increasing the size of the MBD schedule drastically.

- **Decrease Violator Priority (DVP)**, as shown in Algorithm 4, *decrements the frequency of the class that is violating the priority semantics*. For example: if the frequencies of classes (1, 2, 3) are (50, 40, 30) and Class 3 is violating priority semantics then the frequencies become (50, 40, 29). This becomes problematic if the resulting frequency of a class becomes zero. To avoid this situation, the frequency of the higher priority classes gets incremented. For example, if the frequencies of classes (1, 2, 3) are (60, 20, 1) and Class 3 is violating the priorities then the DVP module cannot decrease its frequency. Thus, it increments the frequency of Class 2

Algorithm 6 Decrease Violator Side Priority Algorithm

INPUT: A set of CQ classes SC sorted in non-increasing order of priority

Let Sf be a set of boolean flags initialized to FALSE equal in size to SC
for all CQ classes i in SC **do**
    if i's response time $> i+1$'s response time and i's priority $> i+1$'s priority **then**
        set $Sf[i]$ to TRUE
    end if
end for
set i = 0 and $decr$ = 0
while $i <$ length of SC **do**
    if $SF[i]$ is TRUE **then**
        increment $decr$ by 1
    end if
    decrement i's running frequency by $decr$ and decrement i
end while
for all CQ classes i in SC **do**
    if running frequency of $i < 0$ **then**
        set running frequency of i to 1
    end if
    if running frequency of i = running frequency of $i+1$ and i's priority $> i+1$'s priority **then**
        increment i's running frequency by 1
    end if
end for
if running frequencies did not change **then**
    increment all running frequencies by 1 and rerun algorithm
end if

resulting in (60, 21, 1) frequencies. However, this approach could run out of strategies. For instance, if the frequencies are (30, 20, 19) and Class 2 is violating priority semantics, then Class 2's frequency cannot be decreased otherwise the relative ordering of the classes would

Algorithm 7 Increase Violator Side Priority Algorithm

INPUT: A set of CQ classes SC sorted in non-increasing order of priority

```
Let Sf be a set of boolean flags equal in size to SC
for all CQ classes i in SC do
    if i's response time > i + 1's response time and i's priority > i + 1's priority then
        set Sf[i] to TRUE
    end if
end for
set i = index of lowest priority CQ in SC and inc to 0
while i > 1 do
    if SF[i] is TRUE then
        increment inc by 1
    end if
    increment i's running frequency by inc and decrement i
end while
```

be violated. In this case, the frequencies are doubled before decrementing Class 2's frequency resulting in (60, 39, 38). This module runs into the risk of having a very long schedule once a class in the middle of the frequency range keeps violating priorities. However, it has the advantage of adjusting the frequencies in a slow manner without drastic changes.

- **Increase Victim Priority (IVP)**, as shown in Algorithm 5, *increments the frequency of the class whose response time is higher than that of a lower class*. If this increment results in a tie in frequency, the frequency of the class that was originally more important gets incremented as well. For example: if the frequencies of classes (1, 2, 3, 4, 5) are (50, 40, 30, 20, 10) and Class 2's response time is higher than that of Class 3 then the frequencies become (50, 41, 30, 20, 10).
- **Decrease Violator Side Priority (DSP)**, as shown in Algorithm 6, *determines the classes that are violating the frequency semantics and decrements the frequency of all of the classes that have equal or lower frequency than them*. For example: if the frequencies of classes (1, 2, 3, 4, 5) are (50, 40, 30, 20, 10) and Class 3 is violating priority semantics then the frequencies

of classes 3, 4, and 5 are decremented to become (50, 40, 29, 19, 9). If the frequency of the violating class C_V is 1 or if it runs out of strategies, then the frequencies of all the classes whose frequencies are higher that C_V's frequency are incremented by 1. For instance, if the frequencies are (3, 2, 1) and Class 3's response time is lower than that for Class 2, then the frequencies of classes 1 and 2 are incremented by one to become (4, 3, 1).

- **Increase Violator Side Priority (ISP)**, as shown in Algorithm 7, *determines the classes whose response time is higher that their lower priority classes and increments the frequency of all of the classes that have equal or higher priority*. For example: if the frequencies of classes (1, 2, 3, 4, 5) are (50, 40, 30, 20, 10) and Class 2's response time is higher than that of Class 3 then both classes 1 and 2's frequencies are incremented to become (51, 41, 30, 20, 10).

6.1.6 Adapting to Selectivity Changes

During runtime, the selectivities of the operators could fluctuate causing the selectivity of the CQ to change as well. Because the MBD scheduler is a CQ-level scheduler, it is susceptible to changes in selectivities. When the selectivity of CQ_i changes, its cost per tuple $cost(i)$ and the cost per tuple of its class $cost(class(i))$ also change. This could alter the priority of CQ_i thus making the MBD schedule invalid. In cases when the change in selectivities does not impact the response time of the class, the MBD scheduler does not change the MBD schedule. However, once the change in selectivities results in priority inversion, at the end of the schedule round, the priority inversion module is triggered. There are two ways that the priority inversion module would react to this:

- **_S:** updates the selectivities only without changing the class frequencies. Then, it rebuilds the MBD schedule. A change in selectivities is detected only if the selectivities have changed by a threshold e.g., ten percent.
- **PrInvModule_S:** updates the selectivities only without changing the class frequencies when a change in selectivities is detected. When priority inversion is detected, then only the frequencies are adjusted according to the priority inversion module. Any of the priority inversion modules designed in Section 6.1.5 namely DVP, DSP, IVP, and ISP can be used.
- **PrInvModule_SA:** updates both the selectivities and the frequencies before rebuilding the MBD schedule. The selectivities are updated regardless if the change is more than the threshold

or not. The frequencies are always updated according to the priority inversion module. Any of the priority inversion modules designed in Section 6.1.5 namely DVP, DSP, IVP, and ISP can be used.

6.1.7 Adapting to Priority Changes

Besides the selectivities, the priorities of the classes could change during runtime by explicit action from the system administrator. In this case, the MBD scheduler detects the change and rebuilds the MBD schedule using the new priorities. Also, the assignment of some CQs to classes could change during runtime, triggering the MBD scheduler to rebuild the MBD schedule. These rebuilds add overhead to the execution. However, they are necessary to keep honoring the priorities and the classes. Thus, the rebuilds are only executed at the end of the MBD schedule to avoid disruptions to the execution.

6.2 EXPERIMENTAL SETUP

In order to evaluate the performance of the MBD scheduler, and its variants described in the previous section, we performed a thorough evaluation study.

Table 7: Specifications for Workloads 3D_6, 3E_6, and 3F_6

	Workload 3D_6			**Workload 3E_6**			**Workload 3F_6**		
	Class1	**Class2**	**Class3**	**Class1**	**Class2**	**Class3**	**Class1**	**Class2**	**Class3**
Number of CQs	21	21	21	14	21	28	28	21	14
Types	All types of CQs								
Priorities	60	30	10	30	20	10	30	20	10
Input rate	3600 Tuples/sec								

Table 8: Specifications for Workloads 5I, 5G_2, and 5G_4

Workload 5I					
	Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**
Number of CQs	12	12	15	12	12
Types of CQs	all				
Priorities	50	40	30	20	10
Input rate	3600 Tuples/sec				
Workload 5G					
	Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**
Number of CQs	12	20	28	24	16
Types of CQs	join & aggr	all	all	all	all
Priorities	50	40	30	20	10
Input rate - 5G_4 (4 cores)	1200 Tuples/sec				
Input rate - 5G_2 (2 cores)	600 Tuples/sec				

The version of AQSIOS which runs in a multi-core environment is still under development. Thus, to evaluate the MBD scheduler, we modified our simulator SimAQSIOS so that it simulates AQSIOS running in multiple threads each on a dedicated core. Unless otherwise mentioned, the source operators are scheduled using a round robin scheduler on a separate dedicated core. Before an operational operator connected to a source operator can execute, its input tuples are copied from its source's output queue into its input queue. This implementation is necessary to avoid locking of tuples and queues during the operator's execution. However, once scheduled, a CQ can only execute the tuples that have arrived prior to its scheduling point. Any tuples that arrive after that are not seen by the CQ and must wait until its next scheduling point.

Table 9: Scale_CQs Experiment Workloads

Workload 5S_2					
	Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**
Number of CQs	6	10	14	12	8
Types of CQs	join & aggr	all	all	all	all
Priorities	50	40	30	20	10
Input rate (tup/sec)	1200	1200	1200	1200	1200

6.2.1 Schedulers

Under SimAQSIOS, we implemented our proposed MBD scheduler in addition to the following baseline schedulers:

- **Multicore Round Robin scheduler (MRR)**: a CQ-level scheduler that has equal priority for each CQ. MRR schedules the CQs in a round robin fashion. Each CQ is allowed to execute all of the tuples that have arrived prior to its execution. The operators within each CQ are executed in an input-output fashion so as to maximize train execution.
- **Multicore Operator-Based Round Robin scheduler (MORR)**: an operator-level scheduler that has equal priority for each operator. Care is taken so that the downstream operator can execute at the same time as the upstream operator by copying the tuples that have arrived prior to execution into the downstream operator's input queue at the time of scheduling. Any tuples that have arrived since then are not seen by the downstream operator.
- **Multicore Lottery scheduler (ML)**: a CQ-level scheduler that assigns a number of tickets to the CQs according to their priority. A lottery is drawn at each scheduling point to determine the next CQ to execute. Similarly to other CQ-level schedulers, once a CQ is scheduled, it executes all of the tuples that have arrived prior to its scheduling time.
- **Multicore Operator-level Lottery scheduler (MOL)**: an operator-level ML scheduler that assigns each operator tickets according to its priority. An operator's priority is determined by

Table 10: Scale_Overld Experiment: Specifications of Workload 5O

Workload 5O						
	Number of cores	**Class 1**	**Class 2**	**Class 3**	**Class 4**	**Class 5**
Number of CQs	2	6	10	14	12	8
	4	12	20	28	24	16
	8	24	40	56	48	32
	16	48	80	112	96	64
Types of CQs		join & aggr	all	all	all	all
Priorities		50	40	30	20	10

the operator's expected cost per tuple relative to the CQ class cost per tuple and the CQ class priority.

- **Multicore Proportional Round Robin scheduler (MPRR)**: a round robin CQ level scheduler that places constraints on the percentage of tuples to execute by each CQ. Each CQ is allowed to execute a percentage of its input queue proportional to its class priority.

- **Multicore Per-Class-Thread scheduler (PCT)**: a class-level scheduler that assigns to each class a thread. If the number of cores is equal to the number of classes, then each class thread is assigned to a dedicated core. If the number of cores is less than the number of classes, then the class threads compete over which will execute next. The decision in this case is left to the operating system scheduler. If the number of cores exceeds the number of classes, then each class thread receives a dedicated core with the extra cores being left idle.

- **Multicore Per-Class-Thread Split scheduler (PCT_S)**: is a class-level scheduler that assigns to each class a thread the same as the PCT scheduler. However, when the number of classes is less than the number of cores, then the PCT_S scheduler splits the most important classes into two by dividing the number of CQs equally between the two threads.

Table 11: Specifications of Workload 5L

Workload 5L					
	Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**
Number of CQs	20	32	38	38	32
Types of CQs	all				

6.2.2 Workloads

We evaluate our schedulers with a variety of workloads. These workloads have a variety of load distribution among the classes: increasing according to priority, decreasing inversely to priority and no relationship to priority. We evaluate the schedulers with workloads formed of three and five classes. We chose the input rate such that the workload is heavily loaded but not overloaded in the core configuration used.

- **Workload 5G_4** shown in Table 8 and Figure 15: is composed of five classes with priorities 50, 40, 30, 20 and 10 for classes 1, 2, 3, 4 and 5 respectively. The CQs in the classes have join, selection, aggregation and projection operations. The workload is run on four operational cores with one more core for the source operators. The input rate is 1200 tuples/second.
- **Workload 3D_6** shown in Table 7 and Figure 16: is composed of three classes with priorities 60, 30 and 10 for the respective classes 1, 2 and 3. The load in each class is identical so as to isolate the effect of the load per class on the response time per class. The CQs in the classes have join, selection, aggregation and projection operations. The workload is run on six operational cores with an additional source core. The input rate is 3600 tuples/second.
- **Workload 5E_6** shown in Table 7 and Figure 17: is composed of three classes with priorities 30, 20 and 10 for the respective classes 1, 2 and 3. The load in Class 1 is lower than that of Class 2 which is lower than that of Class 3. The CQs in each of the classes have join, selection, aggregation and projection operations. This workload is run on six operational cores with an additional core dedicated to the source operators. The input rate is 3600 tuples/second for all of the input streams for shared across all of the classes.

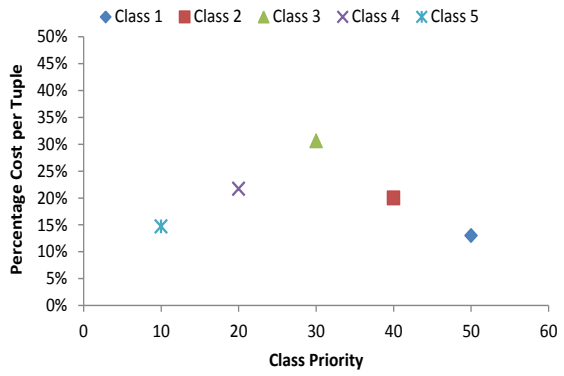

Figure 15: Workloads 5G, 5S and 5O have five classes with Class 3 being heaviest.

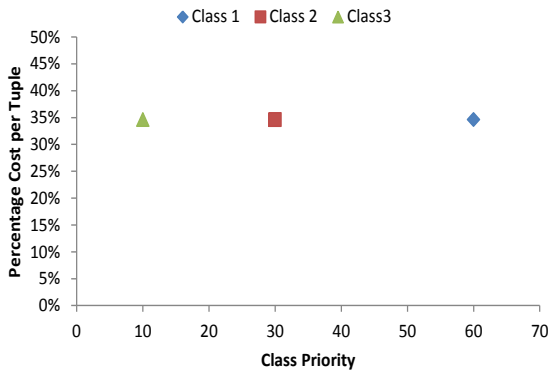

Figure 16: Workload 3D_6 has the same load in each of the three classes.

- **Workload 3F_6** shown in Table 7 and Figure 18: the CQs in this workload are identical to the CQs in Workload 3E_6. However, their assignment to classes is different. Class 2 stays the same with the same priority. Class 1 in Workload 3F_6 has the same CQs as Class 3 in Workload 3E_6 making it the heaviest loaded class but with a priority of 30. Class 3 in Workload 3F_6 is identical to Class 1 in Workload 3E_6 but with a priority of 10. The workload is run on six operational cores with an additional core dedicated to the source operators. The input rate is 3600 tuples/second for all of the input streams across all the classes.

- **Workload 5I** shown in Table 8 and Figure 19: the CQs in this workload are identical to the ones in Workloads 3D_6, 3E_6 and 3F_6. However, their assignment to classes is different. They are distributed across 5 classes: 1, 2, 3, 4 and 5 with respective priorities: 50, 40, 30, 20 and 10. The heaviest class in this workload is Class 3. These CQs are run on six operational cores with an additional core dedicated to the source operators. The input rate is 3600 tuples/second for all of the input streams.

- **Workload 5S_2** shown in Table 9 and Figure 15: is half of Workload 5G_4. It contains half of the CQs in each class because it is run on two cores. It also has five classes with priorities 50, 40, 30, 20 and 10 for classes 1, 2, 3, 4 and 5 respectively. The workload is run on two operational cores with an additional core dedicated to the source operators. The input rate is

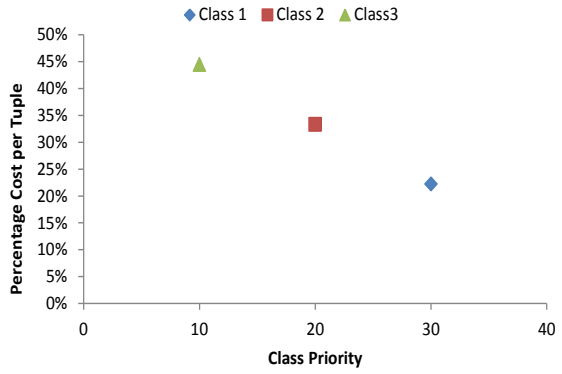

Figure 17: The load among Workload 3E_6's classes increases as the priority decreases.

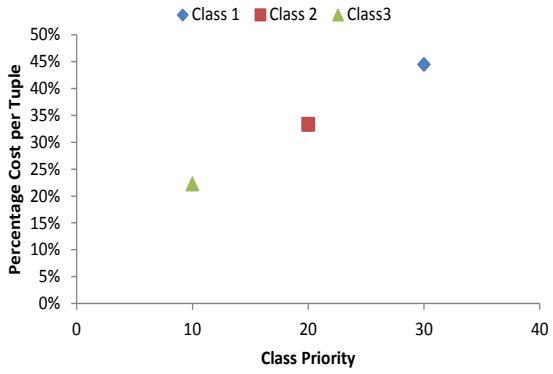

Figure 18: The load among Workload 3F_6's classes increases proportionally to the priority.

1200 tuples/second. For this workload, the subscript number indicates the number of cores it is run on and the number of CQs. Thus, Workload 5S_3 contains fifty percent increase of the CQs and is run on three cores. Workload 5S_4 contains double the CQs and is run on 4 cores.

- **Workload 5G_2** as shown in Table 8 and Figure 15: has the same CQs as Workload 5G_4. However, it is run on two operational cores with an input rate of 600 tuples/second for each of the data sources. Also for this workload, the subscript indicates the number of cores it is run on. It also indicates the scale of the input rate. Thus, Workload 5G_8 is run on eight cores and has an input rate that is double that of Workload 5G_4 and four times that of Workload 5G_2.
- **Workload 5O** as shown in Table 10 and Figure 15: is composed of five classes with priorities 50, 40, 30, 20 and 10 for classes 1, 2, 3 ,4 and 5. The heaviest class in this workload is Class 3 and Class 1 is the lightest. The subscript also indicates the number of cores it is run on and the number of CQs. The input rate for this workload increases from 600 to 2400 tuples/second.
- **Workload 5L** as shown in Table 11 and Figure 20: is also composed of five classes. However, it has a union CQ and an except CQ that are shared among all the classes in this workload. It runs on six operational cores and one core dedicated to the sources which are shared among all the classes. Figure 20 shows the starting priorities 80, 30, 30, 10 and 10 for classes 1, 2, 3, 4 and 5 and the starting distribution of load at the beginning its run.

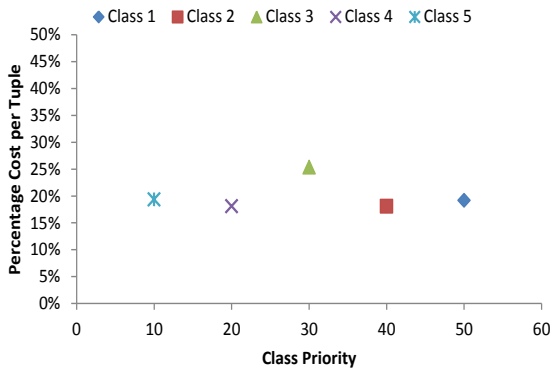

Figure 19: Workload 5I's load varies slightly among the classes with Class 3 being the heaviest.

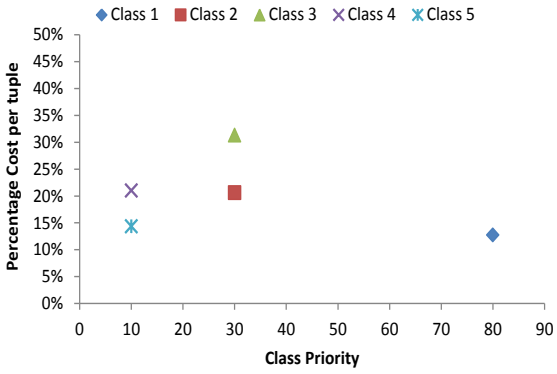

Figure 20: Workload 5L (starting point) has same priority for classes 2 & 3 and classes 4 & 5

6.2.3 Evaluation Metrics

In order to evaluate the schedulers, we use the metrics discussed in Section 2.3:

- **Response time per class:** we measure the response time of each class at the 10-percentile, 25-percentile, median (50-percentile), average, 75-percentile and 90-percentile. These values are calculated after aggregating all the response times of the tuples outputted by all the CQs belonging to the class.
- **Priority inversion ratio:** we calculate the priority inversion ratio at the average, median, 10-percentile, 25-percentile, 75-percentile and 90-percentile levels by using the response times for each class at these levels.
- **Over-time response time:** we measure the average response time of each class every 0.1 second.
- **Over-time priority inversion:** we calculate the priority inversion ratio every 0.1 second by using the over-time response time.
- **Weighted response time of the workload:** we calculate the weighted response time of the workload by priorities and the response times of each class.

- **Starvation ratio:** we divide the response time of the lowest priority class by the response time of the highest priority class as an indication of starvation.
- **Convergence time:** when the MBD schedule is updated, we measure the time of the last update to determine how long it took the scheduler to reach a stable schedule.
- **Convergence updates:** when the MBD schedule is updated, we collect the number of updates before reaching a stable schedule.

6.3 EXPERIMENTAL RESULTS

In this section, we present the results of a set of experiments that evaluate the performance of the MBD scheduler. Section 6.3.1 contains Compare_5G, Compare_3D_6, Compare_3E_6, Compare_3F_6 and Compare_5I where the MBD scheduler is compared against the schedulers mentioned in Section 6.2.1. Then, Section 6.3.2 evaluates the scalability of the MBD scheduler. In Section 6.3.3, we evaluate the priority inversion modules discussed in Section 6.1.5. Then in Section 6.3.4, we compare the different approaches to adapting to selectivity changes. After that, Section 6.3.5 evaluates the performance of the priority detection module. In Section 6.3.6, we allocate more cores for the source operators and compare the results to having one core for the sources. Finally, Section 6.3.7 puts it all together in an experiment where the selectivities, input rates and priorities change in order to compare the MBD scheduler with all of its modules enabled to the schedulers mentioned in Section 6.2.1.

6.3.1 Comparison Experiments: Evaluating the Schedulers

In this section, we compare the performance of the MBD scheduler to the schedulers mentioned in Section 6.2.1 namely: MRR, MORR, ML, MOL, MPRR, PCT and PCT_S. We conduct multiple experiments namely Compare_5G, Compare_3D, Compare_3E, Compare_3F and Compare_5I. The Workloads are shown in Tables 8 and 7.

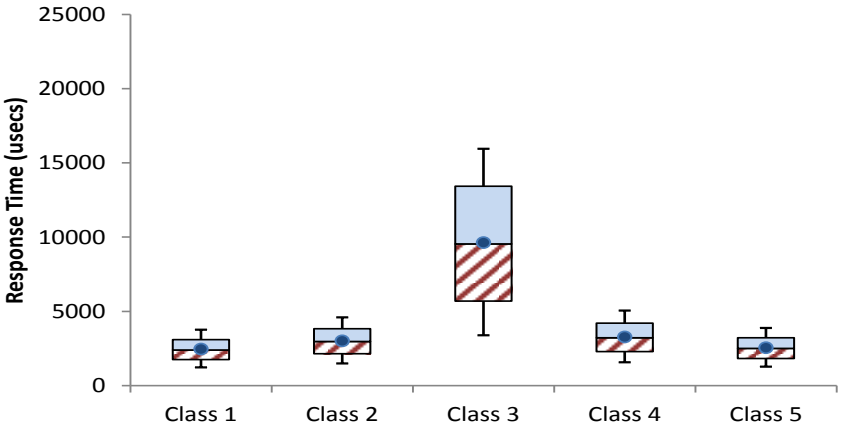

Figure 21: Compare_5G Experiment: PCT allocates the same resources to each class thread. Class 3, the heaviest class, has the highest response time.

6.3.1.1 Compare_5G Experiment (Figures 21 - 29)

This experiment evaluates the schedulers described in Section 6.2.1 with Workload 5G_4 shown in Table 8. Workload 5G_4 has five classes with Class 3 being the heaviest and Class 1 the lightest. It is run on four operational cores and one source core.

The PCT scheduler allocates one thread per class. In this workload, there are five classes which means that there are five operational threads competing for the four operational cores. Whenever a core becomes idle, the thread that is currently not executing is scheduled. This scheduler is very susceptible to the load per class. Because it evenly divides the capacity of the system among the classes, a class with a heavier load would have a higher response time. The performance of the PCT scheduler under this workload is shown in Figure 21. As expected, the response time of Class 3 is the highest because Class 3 is the heaviest class in this workload. Also, because this scheduler does not distinguish between the different classes, it has priority inversion. Because the number of classes is greater than the number of operational cores, the performance of the PCT_S scheduler is identical to the performance of the PCT scheduler.

Similarly to the PCT scheduler, the MRR scheduler and the MORR scheduler do not distinguish between the classes of CQs and optimize the average response time of the system. As shown

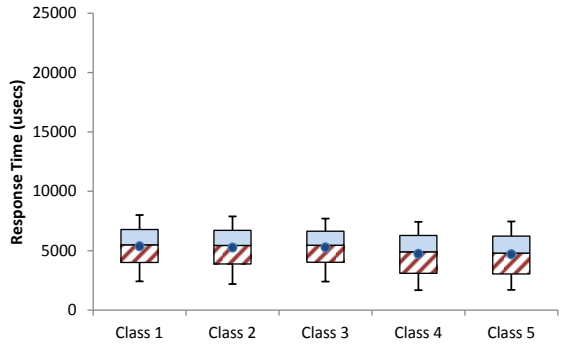

Figure 22: Compare_5G Experiment: MORR has similar response time for all the CQs regardless of CQ class.

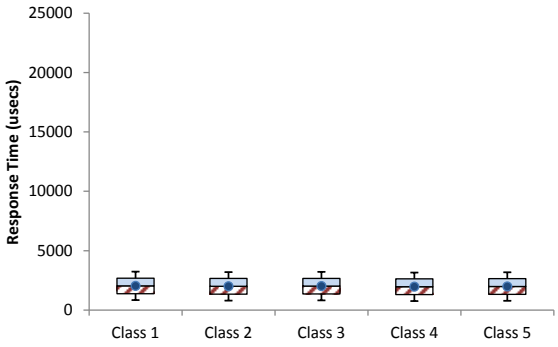

Figure 23: Compare_5G Experiment: MRR has better response time than MORR for all CQs regardless of CQ class.

in Figures 23 and 22, the MRR scheduler has a better average response time than the MORR scheduler because it pipelines the execution of the tuples from the input to the output operators taking advantage of train execution.

The performance of the ML and MOL schedulers is shown in Figures 25 and 24 respectively. The ML scheduler has a better response time than the MOL scheduler because of the pipelining effect of executing the CQs from input to output. It also follows the priorities of the classes in a better fashion. The MOL scheduler divides the priority of the class among its operators according to the operator's relative load in the class. However, this does not guarantee that an operator from a higher class would receive a higher priority because a heavy operator from a low-priority class could potentially have the same priority as a light operator from a high-priority class. This could happen at the CQ-level; but it is more likely at the operator level. In addition, the lottery scheduling scheme does not provide any guarantees on the wait time between execution of an upstream operator and a downstream operator. This results in very little to no effect of the priorities on the processing of the tuples and thus on the response times of the classes.

Both the MBD scheduler and the MPRR scheduler are aware of the classes and optimize the response time of Class 1 as shown in Figures 26 and 27. The MPRR scheduler, however, starves its low-priority classes thus driving their response time to be very high. On the contrary, the MBD

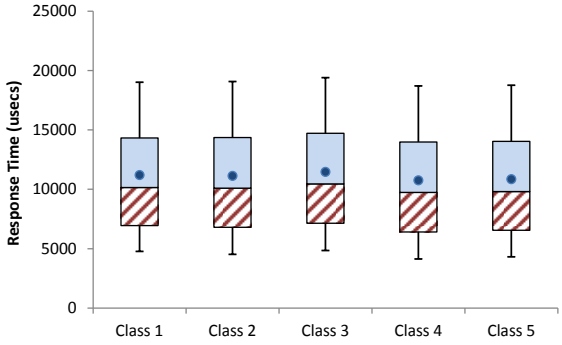

Figure 24: Compare_5G Experiment: MOL has no class preferences so all classes have a high response time.

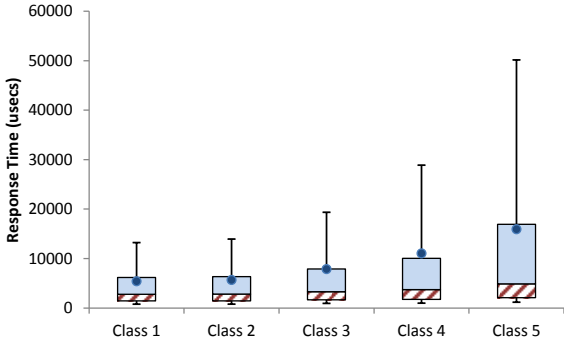

Figure 25: Compare_5G Experiment: ML respects class priorities but has high variation in Class 5's response time.

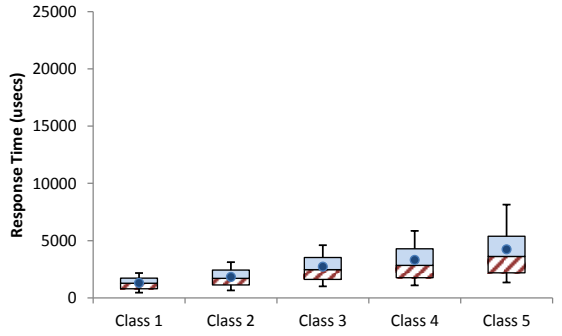

Figure 26: Compare_5G Experiment: MBD provides the lowest response time for Class 1 while slightly increasing Class 5's.

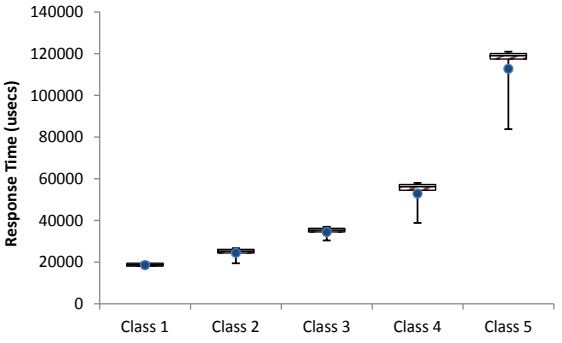

Figure 27: Compare_5G Experiment: MPRR optimizes Class 1's performance while starving other classes.

scheduler has no starvation and provides a lower response time than the MPRR scheduler.

Figure 28 shows the priority inversion ratio for each one of these schedulers at the average, 50-percentile (median), 75-percentile, 90-percentile and 95-percentile. All of the schedulers except for the MBD scheduler and the MPRR scheduler have priority inversion at all the percentiles and the average. Figure 29 shows the priority inversion at the average and the ratio of Class 3's response

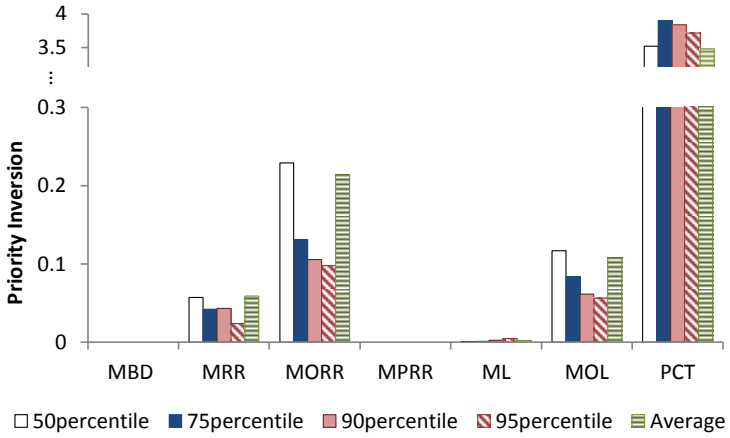

Figure 28: Compare_5G Experiment: Schedulers MBD and MPRR are the only schedulers without priority inversion at any percentile level.

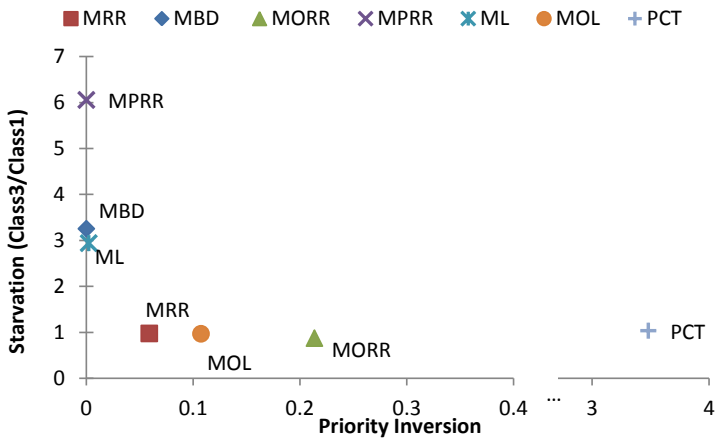

Figure 29: Compare_5G Experiment: The MBD scheduler provides the best trade-off between having no priority inversion and optimizing response time of all classes.

time to Class 1. The MPRR scheduler has a higher response time ratio than the MBD scheduler. Other schedulers have lower response time ratios but with priority inversion. As a result, the MBD scheduler provides the best performance among the schedulers in terms of response time, priority inversion and starvation which makes it suitable for class-aware DSMSs.

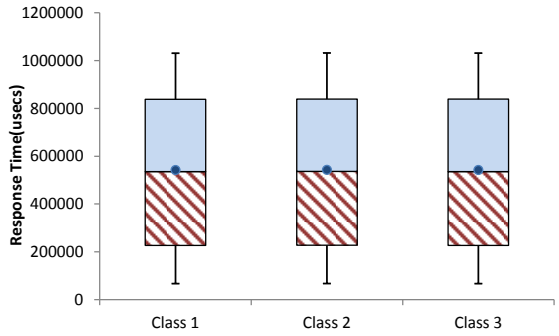

Figure 30: Compare_3D_6 Experiment: PCT has a high response time. The 90-percentile reaches 1 sec.

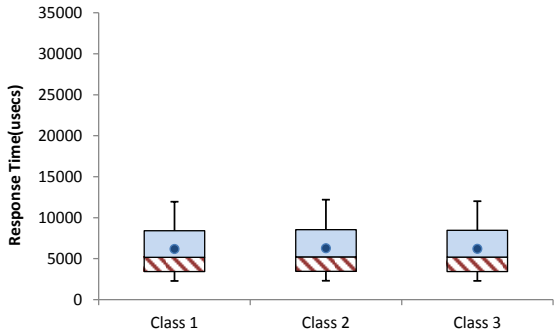

Figure 31: Compare_3D_6 Experiment: PCT_S splits each class into 2 reducing the response time.

6.3.1.2 Compare_3D_6 Experiment (Figures 30 - 39) Workload 3D_6 is shown in Table 7. It is composed of three classes with identical load in each class. This workload is run on six operational cores with an additional core dedicated to the source operators.

Figure 30 shows the performance of the PCT scheduler. For this workload, the PCT scheduler cannot utilize all of the cores because it has only three classes but six operational cores. Thus, the system gets overloaded resulting in a response time above one second.

The PCT_S scheduler splits each of the classes into two so that it can have six concurrent threads. As shown in Figure 31, the response time in this case is below 15 msecs and the system is not overloaded. The PCT_S scheduler is aware of the different classes but is not aware of their priorities. It isolates the classes but treats them equally.

Similarly to the PCT_S scheduler, the MORR and MRR schedulers are not aware of the priorities of the classes. However, both schedulers optimize the overall response time of the workload. Thus, the response time of each of the classes is lower than 5 msecs for MRR as shown in Figure 33 and lower than 7 msecs for MORR as shown in Figure 32.

Figure 35 shows the performance of the ML scheduler running Workload 3D_6. However, some of the tuples in Class 3 have a much higher response time than others. The 95-percentile is above 30 msecs whereas the median is slightly above 6 msecs. Thus, some of the tuples in

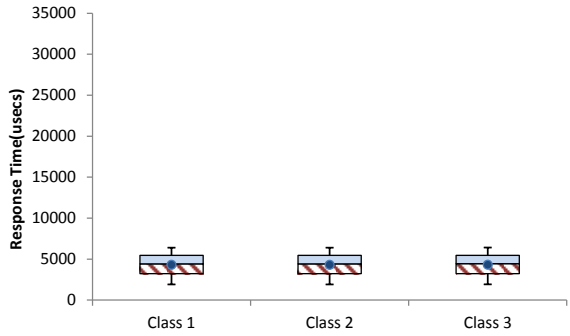

Figure 32: Compare_3D_6 Experiment: MORR has better response time than PCT_S but without preference for Class 1.

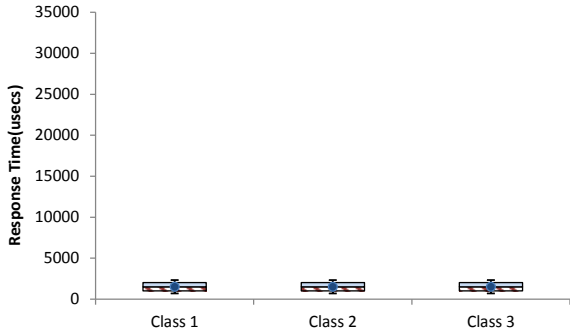

Figure 33: Compare_3D_6 Experiment: MRR provides even better response time than MORR but also without preference for Class 1.

Class 3 suffer from longer wait times and temporary starvation. As shown in Figure 34, the MOL scheduler does not have this problem of response time variability but rather results in very close response times of the different classes. This is because the MOL scheduler is an operator level scheduler that cannot control the wait time the tuples spend at the queues of operators. Thus, all of the tuples tend to wait a similar amount of time because of the randomized property of the scheduling.

Figure 36 shows that the MBD scheduler optimizes the response time of Class 1 without starving classes 2 and 3. This is unlike the MPRR scheduler as shown in Figure 37. Under the MPRR scheduler, the response time of Class 1 is the lowest in this workload but the response time of Class 3 is much higher: Class 3's response time is ten times Class 1's response time.

Figure 38 shows the priority inversion ratio for each of the schedulers at the average, median (50-percentile) , 75-percentile, 90-percentile and 95-percentiles. Schedulers MRR, MORR, MOL, PCT and PCT_S have priority inversion making them not suitable for class-aware scheduling. Schedulers ML, MBD and MPRR do not have priority inversion. Figure 39 shows the ratio of the response times of Class 3 to Class 1 and the priority inversion ratio at the average for each of the schedulers. The MPRR scheduler results in the highest response time ratio of 10 while the PCT_S scheduler results in the highest priority inversion ratio of 0.021. Out of the schedulers with

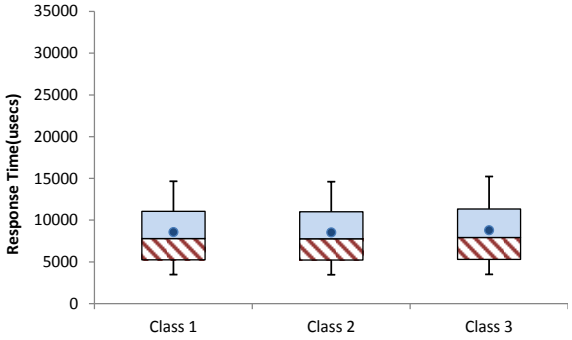

Figure 34: Compare_3D_6 Experiment: MOL has no preference for Class 1 and has a higher response time for the classes than PCT_S.

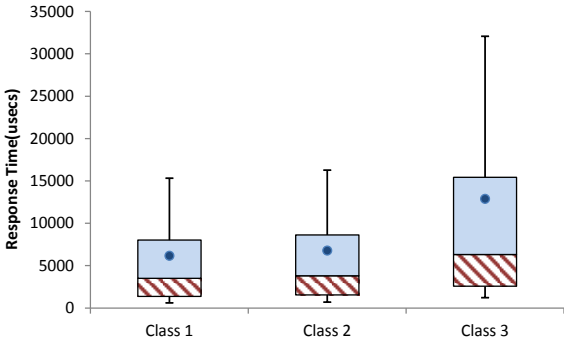

Figure 35: Compare_3D_6 Experiment: ML has a tighter response time range for Class 1 than Class 3 (30 msecs).

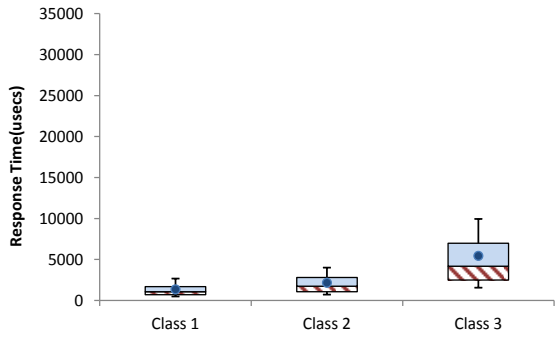

Figure 36: Compare_3D_6 Experiment: MBD optimizes the response time of Class 1 without starving Class 3.

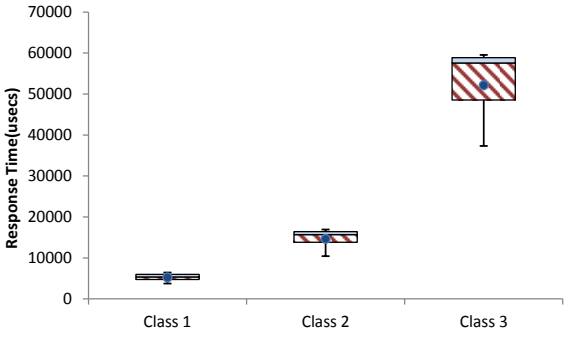

Figure 37: Compare_3D_6 Experiment: MPRR starves Class 3 driving its response time to 60 msecs at the 90 percentile.

zero priority inversion for this workload namely MBD, ML and MPRR, the MBD scheduler has the lowest response time for the individual classes and the entire workload. Thus, it is the best performing scheduler for this workload.

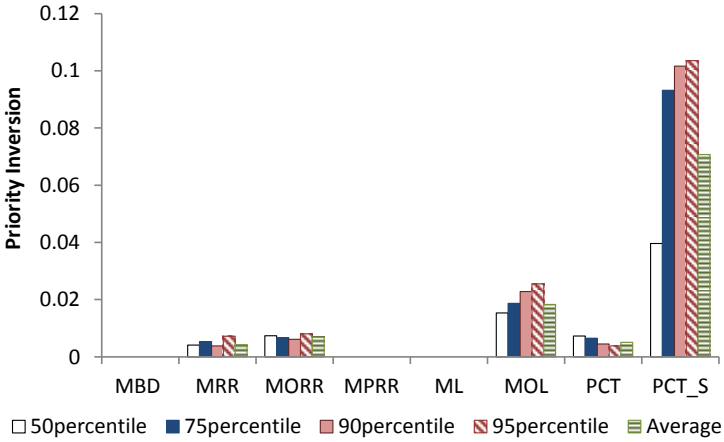

Figure 38: Compare_3D_6 Experiment: MBD, MPRR and ML have no priority inversion for Workload 3D_6

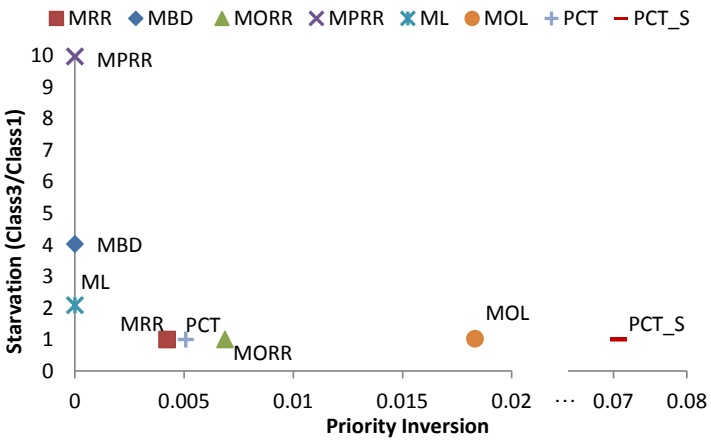

Figure 39: Compare_3D_6 Experiment: ML, MPRR and MBD have zero priority inversion at the average. ML has a better starvation ratio than MBD but provides a much higher response time for Class 1 and Workload 3D_6.

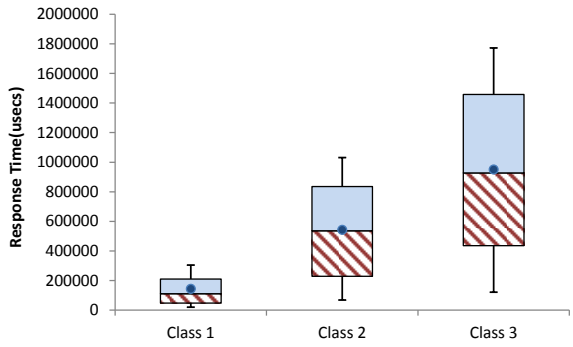

Figure 40: Compare_3E_6 Experiment: PCT's response time follows the load of the classes. Class 3 has the highest response time.

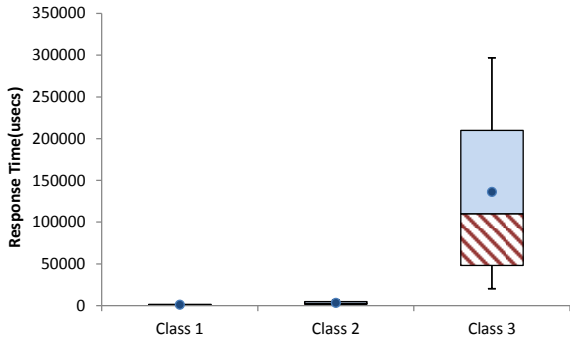

Figure 41: Compare_3E_6 Experiment: PCT_S's response time follows the load of the classes. Class 3 has the highest response time.

6.3.1.3 Compare_3E_6 Experiment (Figures 40 - 49) Workload 5E_6 is shown in Table 7. It is composed of three classes with priorities 30, 20 and 10 for the respective classes 1, 2 and 3. The load in Class 1 is lower than that of Class 2 which is lower than that of Class 3. This workload is run on six operational cores with an additional core dedicated to the source operators.

Figure 40 shows the performance of the PCT scheduler when Workload 3E_6 is run. Similarly to Workload 3D_6, the PCT scheduler cannot utilize all of the cores because it has only three classes but six operational cores. Thus, the system gets overloaded. This affects Class 3 mostly because it is the heaviest class in the system resulting in an average response time close to one second. The PCT_S scheduler splits each of the classes into two so that it can have six concurrent threads. However, it keeps the semantics of the classes because it does not attempt to load balance the threads. As shown in Figure 41, the response time for Class 3 is below one second but it still ends up starving with its response time being 123 times Class 1's response time.

Similarly to Workload 3D_6, the MORR and MRR schedulers are not aware of the priorities of the classes and treat all of the operators and continuous queries equally. Thus, the response time of each of the classes is lower than 7 msecs for MORR as shown in Figure 42 and lower than 3 msecs for MRR as shown in Figure 43.

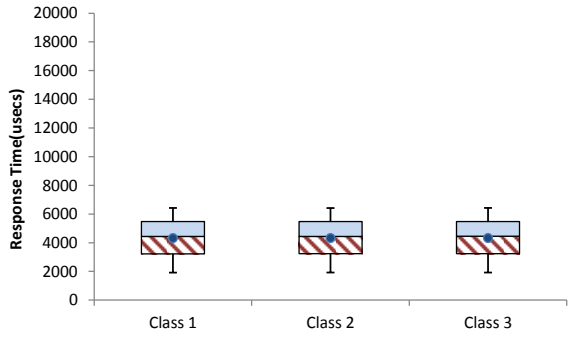

Figure 42: Compare_3E_6 Experiment: MORR provides equivalent performance for all classes. Both median and average are above 4 msecs.

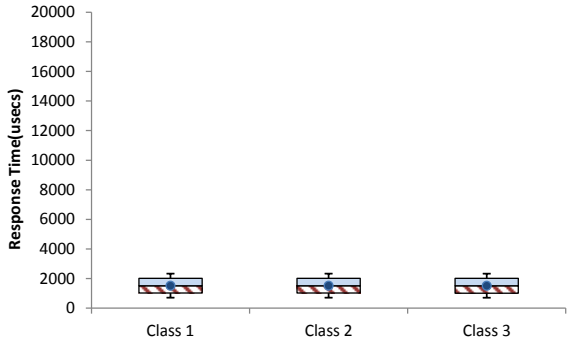

Figure 43: Compare_3E_6 Experiment: MRR provides lower response times for the classes than MORR but no preference for Class 1.

As shown in Figure 44, the MOL scheduler also has response time variability but it spans equally all of the classes for Workload 3E_6. This is because all of the tuples tend to wait a similar amount of time because of the randomized property of lottery scheduling. However under the ML scheduler as shown in Figure 45, the response time of Class 1 has a shorter range than that of Class 3. Also Class 1's average response time is lower than under MOL.

Figure 46 shows that the MBD scheduler optimizes the response time of Class 1 without starving classes 2 and 3. All of the classes have a short response time variability. Figure 47 shows that under the MPRR scheduler, the response time of Class 1 is lower than that of Class 3. However, this is at the expense of unnecessarily high response time for the latter class even though Class 1's response time under MPRR is still much higher than that under MBD.

Figure 48 shows the priority inversion ratio for each of the schedulers under Workload 3E_6 at the average, median, 75-percentile, 90-percentile and 95-percentiles. Schedulers MRR, MORR, and ML have priority inversion but the rest of the schedulers do not. Figure 49 shows the ratio of the response times of Class 3 to Class 1 and the priority inversion ratio for each of the schedulers. The PCT_S scheduler results in the highest response time ratio of 123 while the MORR scheduler results in the highest priority inversion ratio of 0.009. In general, the priority inversion for this workload across some schedulers is low because their dependency on the distribution of the load

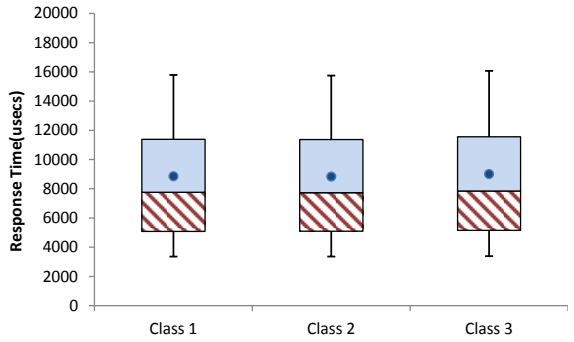

Figure 44: Compare_3E_6 Experiment: MOL has a high variability in the response times of all classes with no preference for Class 1.

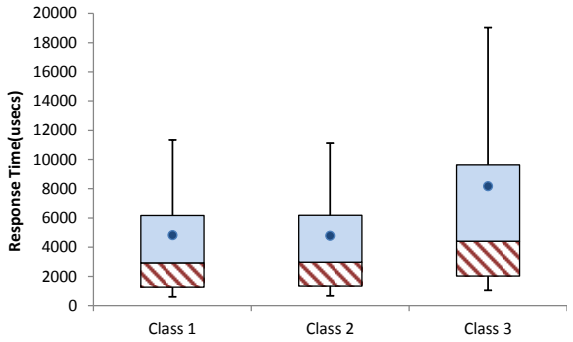

Figure 45: Compare_3E_6 Experiment: ML has smaller variation in the response time for Class 1 and much larger variation for Class 3.

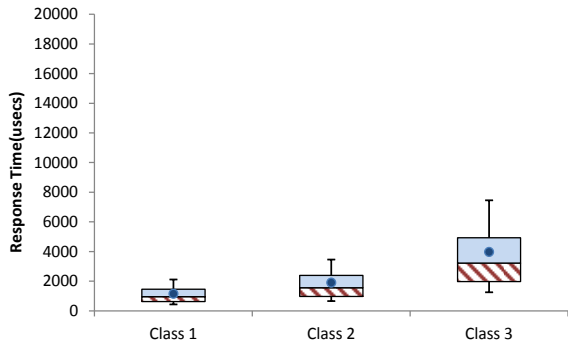

Figure 46: Compare_3E_6 Experiment: MBD has a tight range of response times with Class 1's response time being the lowest.

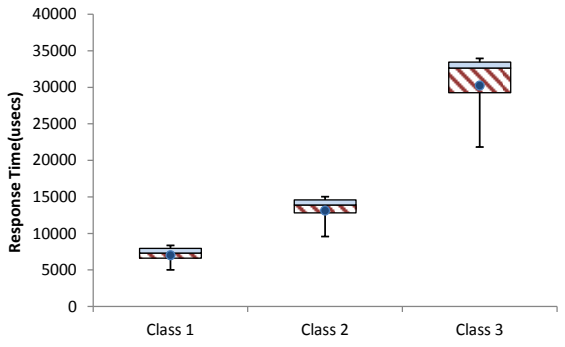

Figure 47: Compare_3E_6 Experiment: MPRR provides better response time for Class 1 but a much higher response time for Class 3.

among the classes. The MBD scheduler for this workload again has no priority inversion and as shown in Figure 49 the ratio of Class 3's response time to Class 1's response time is only 3.6.

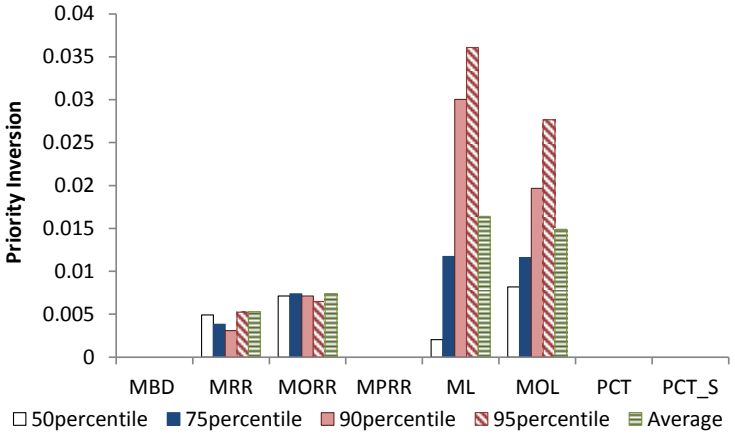

Figure 48: Compare_3E_6 Experiment: MBD, MPRR, PCT and PCT_S have no priority inversion.

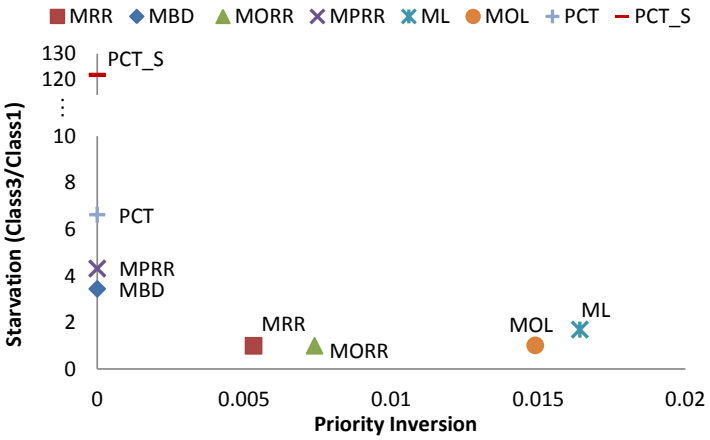

Figure 49: Compare_3E_6 Experiment: the MBD scheduler has no priority inversion, low ratio of Class 3's response time to Class 1 and best response time for Class 1.

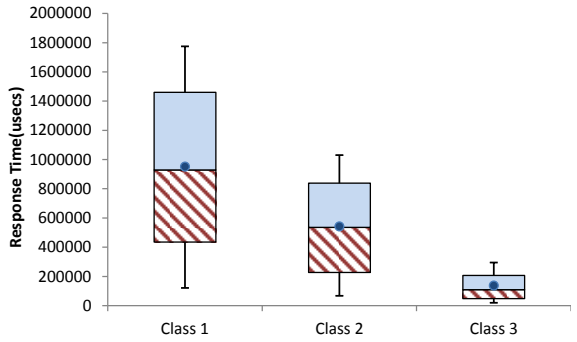

Figure 50: Compare_3F_6 Experiment: PCT has the same performance for Workloads 3F_6 and 3E_6 with reversed class labels.

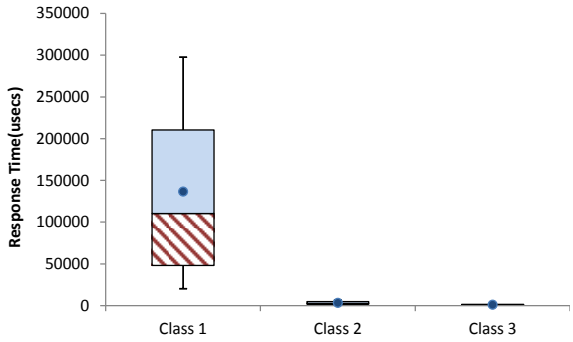

Figure 51: Compare_3F_6 Experiment: PCT_S performance is identical to that for Workload 3E_6 with reversed class labels.

6.3.1.4 Compare_3F_6 Experiment (Figures 50 - 59) Workload 3F_6 is shown in Table 7. The CQs in this workload are identical to the ones in Workload 3E_6. However, their assignment to classes is different for classes 1 and 3. Class 2 stays the same with the same priority. Class 1 in Workload 3F_6 has the same CQs as Class 3 in Workload 3E_6. Class 3 in Workload 3F_6 is identical to Class 1 in Workload 3E_6. The workload is run on six operational cores with an additional core dedicated to the source operators. The input rate is 3600 tuples/second for all of the input streams across all the classes.

Figure 50 shows the performance of the PCT scheduler when Workload 3F_6 is run. Similarly to Workloads 3D_6 and 3E_6, the PCT scheduler cannot utilize all of the cores and the system gets overloaded. This affects Class 1 driving its average response time close to one second because it is the heaviest class in this workload. The PCT_S scheduler, for this workload, splits each of the classes into two so that it can have six concurrent threads. As shown in Figure 51, the response time in this case is below half a second but Class 1 ends up with a significantly higher response time than classes 2 and 3.

As mentioned earlier, the MORR and MRR schedulers are not aware of the priorities of the classes and treat all of the operators and CQs in the same fashion. Because the only difference between Workloads 3E_6 and 3F_6 is that classes 3 and 1 are switched, the performance of schedulers

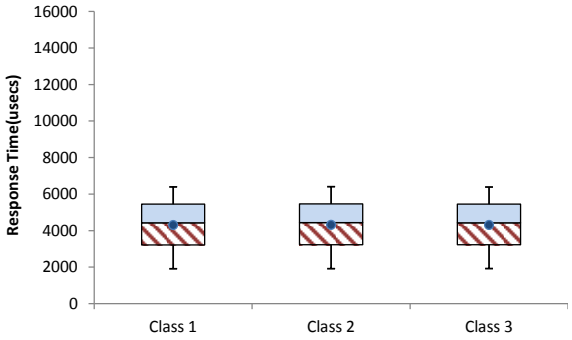

Figure 52: Compare_3F_6 Experiment: MORR cannot distinguish between Workloads 3E_6 and 3F_6, providing the same performance.

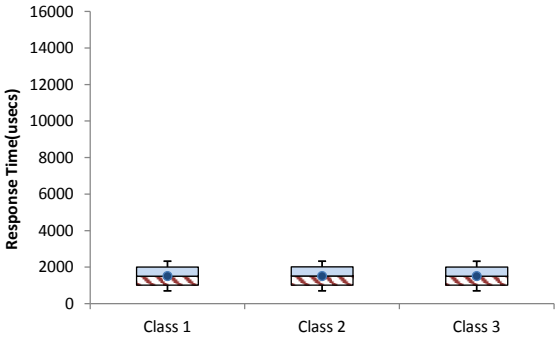

Figure 53: Compare_3F_6 Experiment: MRR also provides the same performance for Workloads 3F_6 and 3E_6.

MORR and MRR is identical for both workloads except for the swapped class labels as shown in Figures 53 and 52.

As shown in Figure 54, the MOL scheduler also for this workload has response time variability spanning equally all of the classes for Workload 3F_6. This is because all of the tuples tend to wait a similar amount of time because of the randomized property of the scheduling. Thus, its performance is identical to that under Workload 3E_6. As shown in Figure 55, the ML scheduler attempts to optimize the performance of Class 1 resulting in a tighter response time range than that under the MOL scheduler. However, this range is still very large for any of the classes spanning more than 10 msecs.

Figure 56 shows that the MBD scheduler is aware of the classes and their priorities. Unlike the ML scheduler, it results in tighter ranges for all of the classes. Also, it optimizes the response time of Class 1 without starving classes 2 and 3. The MPRR scheduler is also aware of the classes and their priorities. As shown in Figure 57, the priorities are obeyed. However, classes 2 and 3 have very high response times compared to Class 1.

Figure 58 shows the priority inversion ratio for each of the schedulers under Workload 3F_6 at the average, median, 75-percentile, 90-percentile and 95-percentiles. All of the schedulers with the exception of MBD and MPRR have priority inversion. Figure 59 shows the ratio of the response

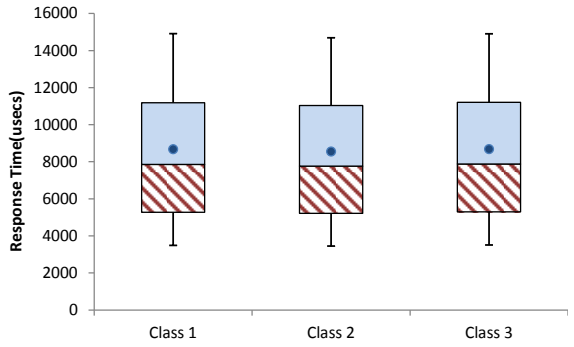

Figure 54: Compare_3F_6 Experiment: MOL has a large range of response time for the classes which is identical to Workload 3E_6.

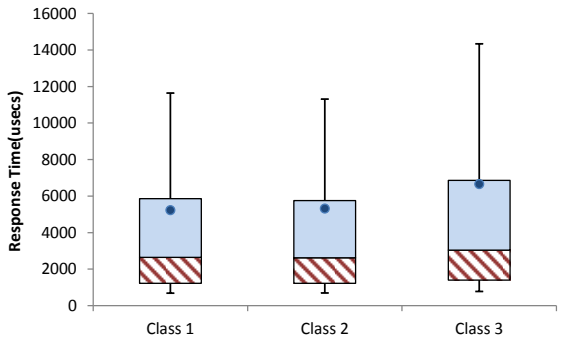

Figure 55: Compare_3F_6 Experiment: ML has less variation of response time for Class 1 than that of Class 3 but they are 10 msecs.

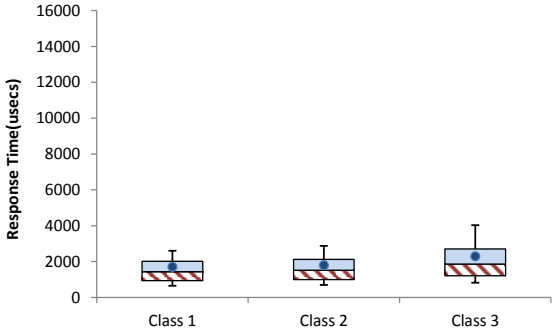

Figure 56: Compare_3F_6 Experiment: MBD optimizes the response time of Class 1 while slightly increasing that of Class 3.

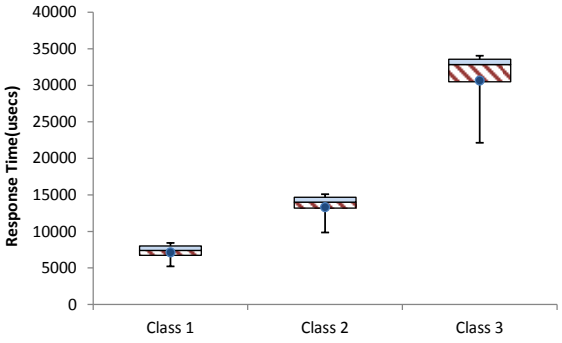

Figure 57: Compare_3F_6 Experiment: MPRR again starves Class 3 driving its average response time to 30 msecs.

times of Class 3 to Class 1 and the priority inversion ratio for each of the schedulers. The MPRR scheduler results in the highest response time ratio of 4.3 while the PCT_S scheduler results in the highest priority inversion ratio of 63.5. However, the MBD scheduler has zero priority inversion and a starvation ratio less than 1.5.

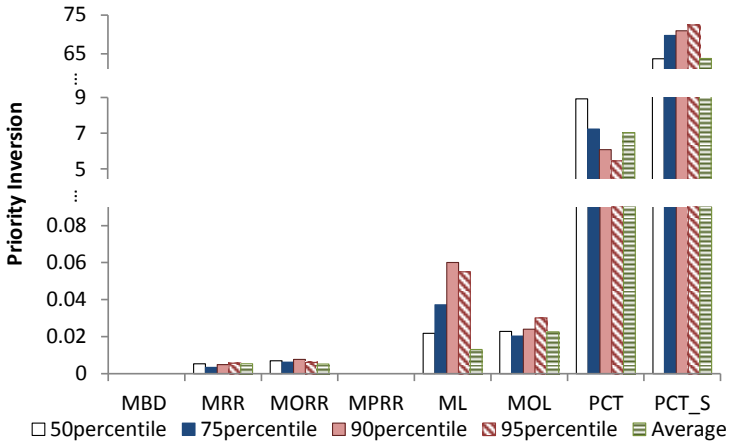

Figure 58: Compare_3F_6 Experiment: only the MBD and the MPRR schedulers have zero priority inversion at all percentiles.

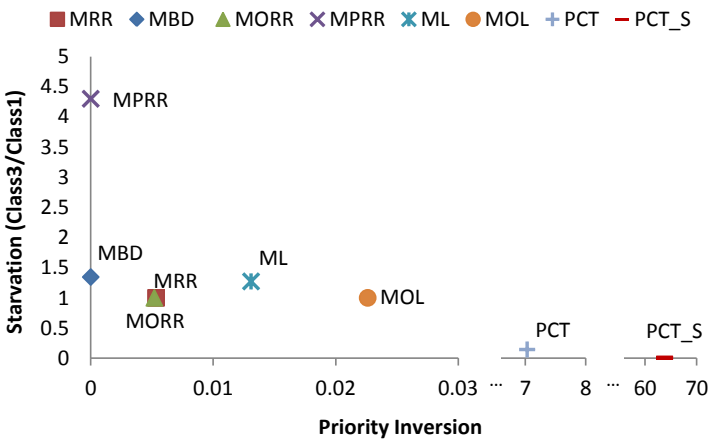

Figure 59: Compare_3F_6 Experiment: the MBD scheduler provides a low starvation ratio while still maintaining a zero priority inversion ratio.

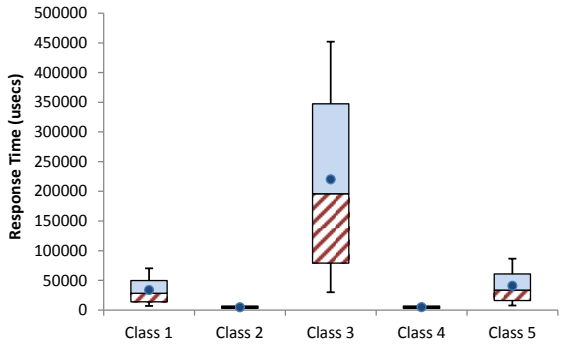

Figure 60: Compare_5I Experiment: PCT's performance is influenced by the load distribution, thus Class 3 reaches 450 msecs.

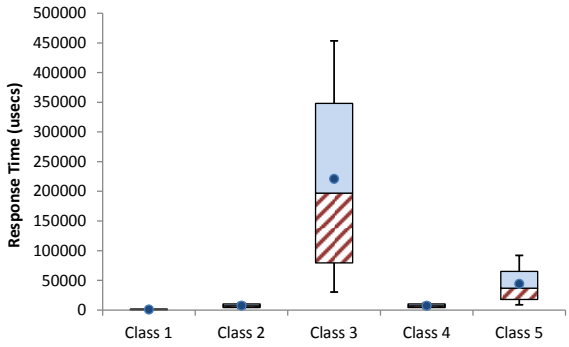

Figure 61: Compare_5I Experiment: PCT_S divides Class 1 into 2 threads reducing its response time but no change to Class 3's.

6.3.1.5 Compare_5I Experiment (Figures 60 - 69) Workload 5I is shown in Table 8. The CQs in this workload are identical to the ones in Workloads 3D_6, 3E_6 and 3F_6. However, their assignment to classes is different. They are distributed across 5 classes: 1, 2, 3, 4 and 5 with respective priorities: 50, 40, 30, 20 and 10. The heaviest class in this workload is Class 3. These CQs are run on six operational cores with an additional core dedicated to the source operators. The input rate is 3600 tuples/second for all of the input streams.

Figure 60 shows the performance of the PCT scheduler when Workload 5I is run. Again for this workload, the PCT scheduler cannot utilize all of the cores and the system gets overloaded. This affects Class 3 the most, driving its response time to be much higher than classes 4 and 5 because it is the heaviest class in the system. In addition, Class 1's response time is higher than that of classes 2 and 4. The PCT_S scheduler, for this workload, splits Class 1 into two threads so that it can have six concurrent threads. As shown in Figure 61, the response time of Class 3 does not change from the PCT scheduler because it is still allocated the same amount of resources. The only difference is that Class 1's response time is lower under the PCT_S scheduler.

Recall that the MORR and MRR schedulers allocate the same amount of resources to the operators and CQs regardless of their priorities. As shown in Figures 63 and 62, both schedulers result in a low response time for this workload but with no preference for higher priority classes.

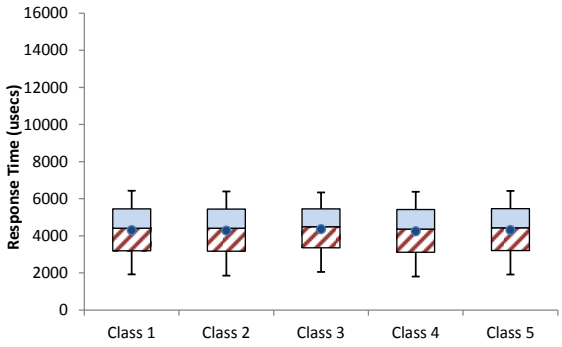

Figure 62: Compare_5I Experiment: MORR cannot distinguish between the classes and all classes have similar response times.

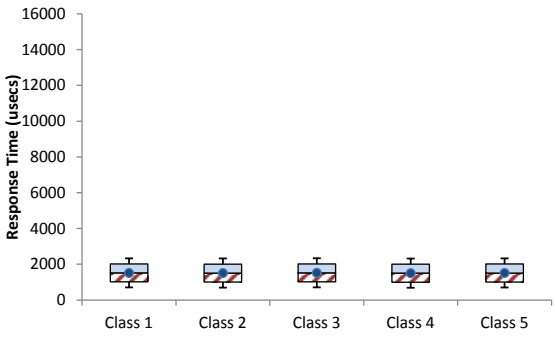

Figure 63: Compare_5I Experiment: MRR treats all the CQs equally without any preference to higher priority classes.

Figures 65 and 64 show that both the ML and MOL schedulers have large response time variability spanning more than 10 msecs. The MOL scheduler has the same range for all the classes whereas the ML scheduler has a shorter range for classes 1 and 2 balanced by a huge range for Class 5.

Figure 66 shows that the MBD scheduler, also for Workload 5I, optimizes the response time of the higher priority classes without starving the other classes. On the contrary, the MPRR scheduler while also being aware of the classes and their priorities, starves the lower priority classes driving the response times of classes 4 and 5 to be above 30 msecs as shown in Figure 67.

Figure 68 compares the priority inversion ratio under each of the schedulers for Workload 5I at the average, median, 75-percentile, 90-percentile and 95-percentiles. All of the schedulers with the exception of the MBD and MPRR schedulers have priority inversion. Figure 69 shows the ratio of the response times of Class 3 to Class 1 and the priority inversion ratio for each of the schedulers. The PCT_S scheduler results in the highest response time ratio of 48 while the PCT scheduler results in the highest priority inversion ratio of 60.9. Among the schedulers with zero priority inversion, the MBD scheduler still succeeds at limiting the response time of Class 5 so that the ratio of Class 5's response time to Class 1's response time is 6.2. This makes it the best scheduler to use for Workload 5I as well.

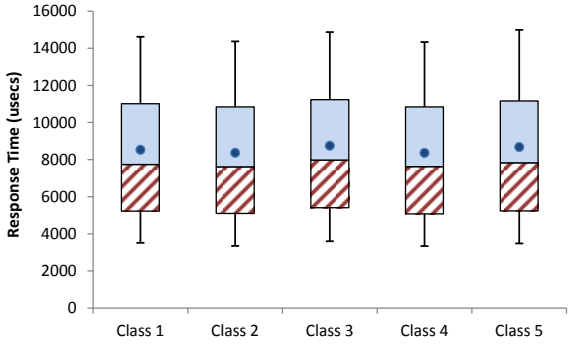

Figure 64: Compare_5I Experiment: MOL suffers from a large range in the response time of each class regardless of priority.

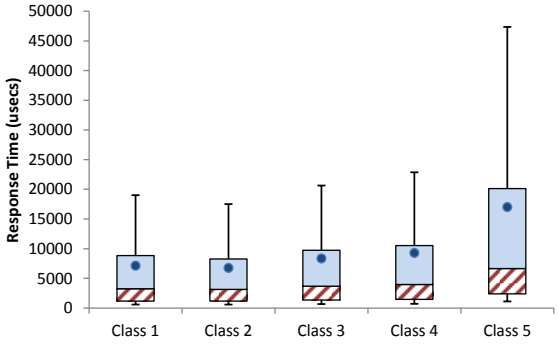

Figure 65: Compare_5I Experiment: ML has tighter ranges for classes 1 and 2 compared to a huge range for Class 5's response time.

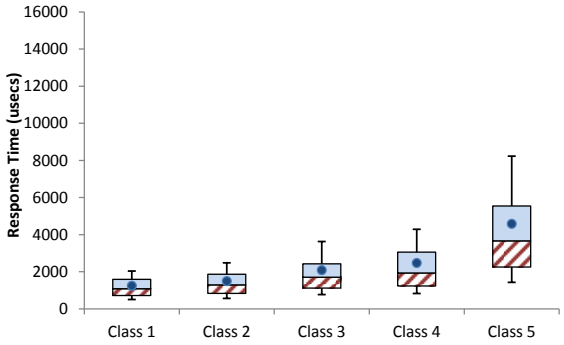

Figure 66: Compare_5I Experiment: MBD has lower response times for high priority classes without starving lower priority classes.

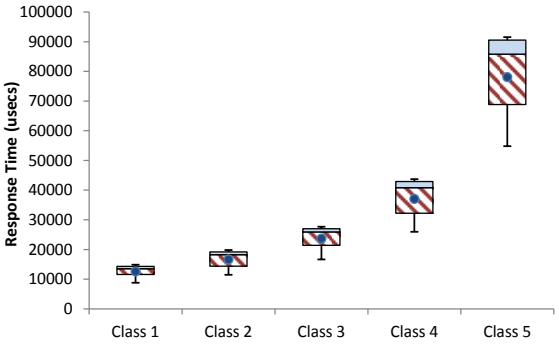

Figure 67: Compare_5I Experiment: MPRR drives the response time of Class 5 to be close to 90 msecs preferring higher priority classes.

6.3.1.6 Comparison Experiments Summary As shown in the previous experiments, the MBD scheduler is the best class-aware scheduler for the multi-core system model. It optimizes the response time of the high-priority classes while still providing good performance to the low-priority classes. For the workloads run in this section, it had no priority inversion at any of the average, median or other percentile levels unlike many of the other evaluated schedulers. It also provided

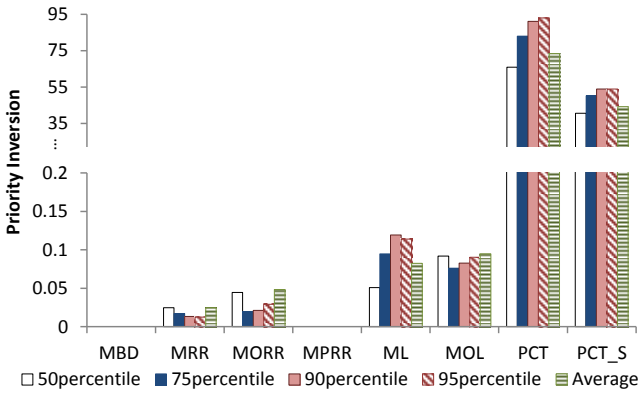

Figure 68: Compare_5I Experiment: only the MBD and MPRR schedulers successfully prevent priority inversion for Workload 5I

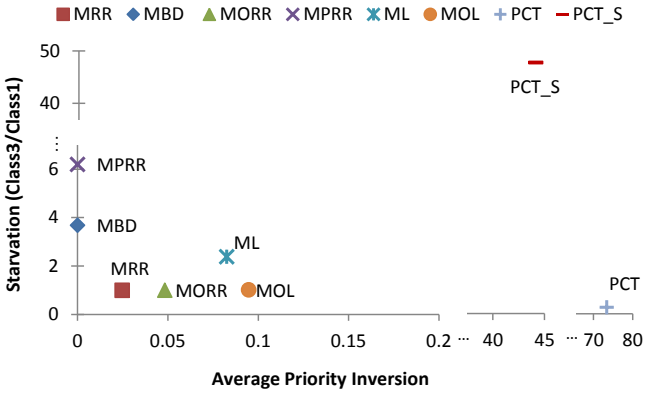

Figure 69: Compare_5I Experiment: the MBD scheduler not only prevents priority inversion but also provides a lower starvation ratio than MPRR.

low response times for the high-priority classes without starving any of the other classes. Overall, under the MBD scheduler the workloads had the lowest response time among the schedulers without priority inversion.

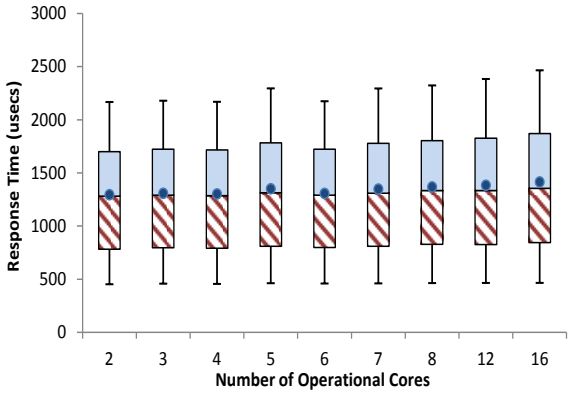

Figure 70: Scale_CQs Experiment: Class 1 response time stays below 2.5 msecs.

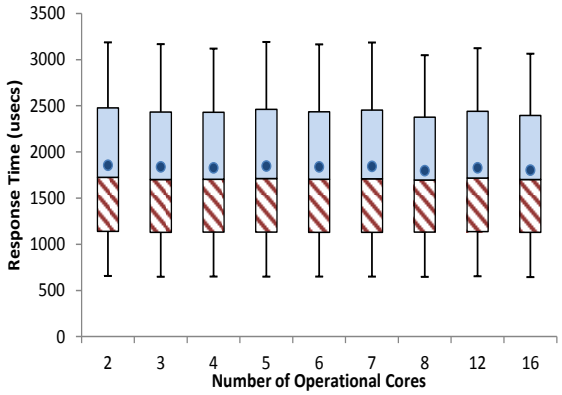

Figure 71: Scale_CQs Experiment: Class 2 response time stays below 3.5 msecs.

6.3.2 Scalability Experiments

In order to evaluate the MBD scheduler's scalability and how effectively it utilizes the cores allocated to the DSMS, we devised three experiments: Scale_CQs that evaluates how the MBD scheduler scales when the number of CQs increases proportionally to the number of cores, Scale_Input that evaluates how the MBD scheduler scales as the input rate increases proportionally to the number of cores and Scale_Overld which evaluates how the MBD scheduler reacts to an increase in the input rate in multiple number of cores.

6.3.2.1 Scale_CQs Experiment (Figures 70 - 75) We vary the number of operational cores from 2 to 16. The number of CQs per class increases proportionally to the number of cores. Table 9 shows Workload 5S_2 when the number of cores is two. Workload 5S_4 contains double the CQs as Workload 5S_2 and runs on four cores. All of CQs in a class under Workload 5S_2 are replicated twice so as to scale the load. In general, we replicate the CQs in each class according to the number of operational cores. Figures 70, 71, 72, 73, and 74 show the response time per class for each of these workloads as the load increases proportionally to the number of operational cores used. The

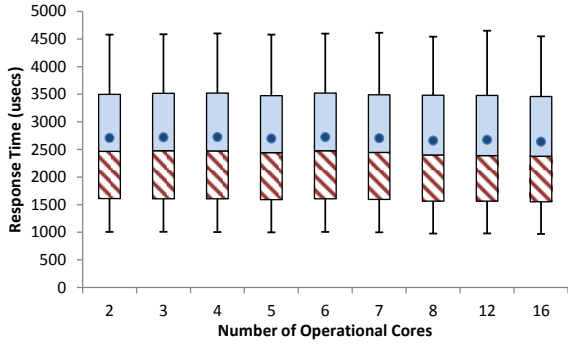

Figure 72: Scale_CQs Experiment: Class 3 response time stays below 5 msecs.

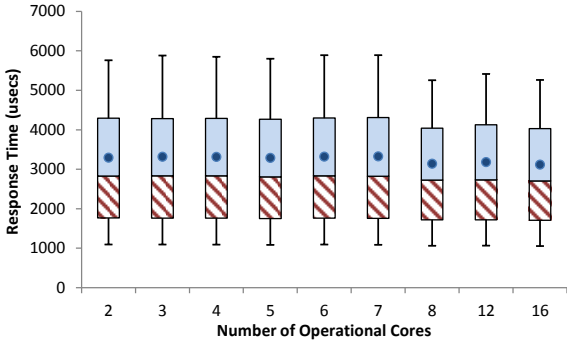

Figure 73: Scale_CQs Experiment: Class 4 response time stays below 6 msecs.

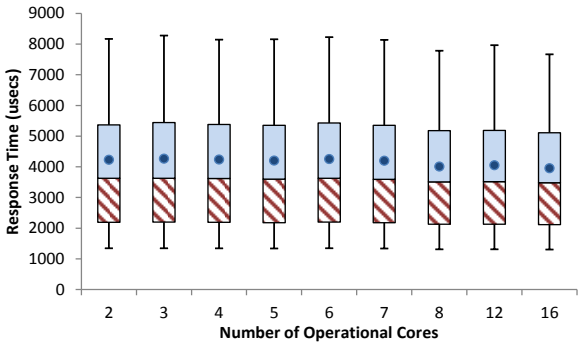

Figure 74: Scale_CQs Experiment: Class 5 response time stays below 9 msecs.

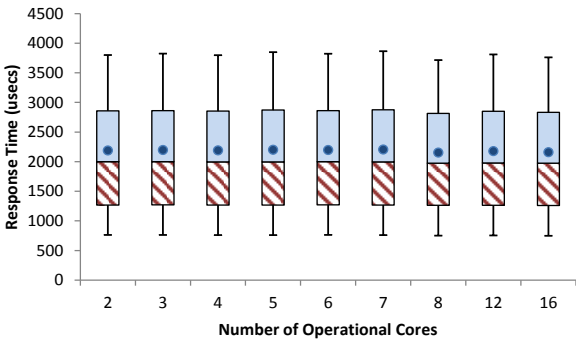

Figure 75: Scale_CQs Experiment: the weighted response time stays below 4 msecs.

response time per class does not increase. Because the MBD scheduler is a CQ-level scheduler, increasing the number of CQs makes room for more concurrent CQs executing on different cores. Thus, the average response times of the Class 1 stays in the range of [1.28, 1.56] msecs, of Class 2 in the range of [1.72, 1.85] msecs, Class 3 in the range of [2.54, 2.67] msecs, Class 4 in the range of [3.00, 3.13] msecs, and Class 5 in the range of [3.85, 4.13] msecs. As shown in Figure 75, the weighted average response time stays in the range of [2.11, 2.18] msecs. This shows that the MBD scheduler scales well over the operational cores as the number of CQs increases.

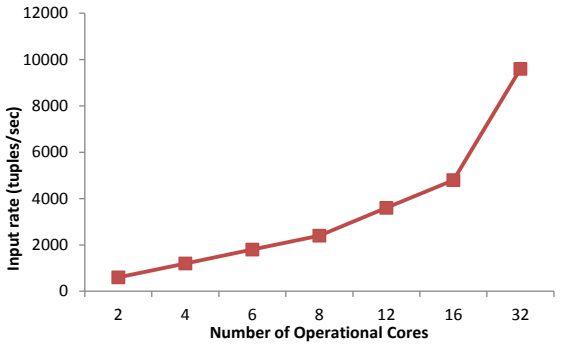

Figure 76: Scale_Input Experiment: the input rate increases proportionally to the number of cores.

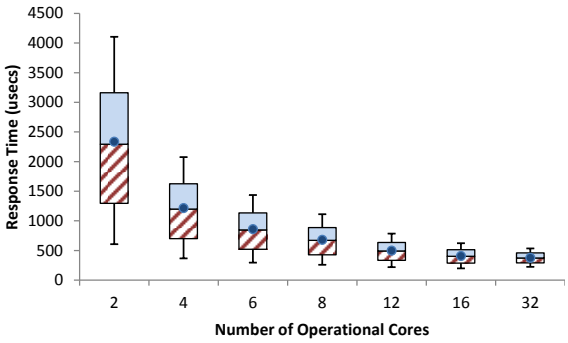

Figure 77: Scale_Input Experiment: Class 1 response time decreases as the input rate increases because of batch processing.

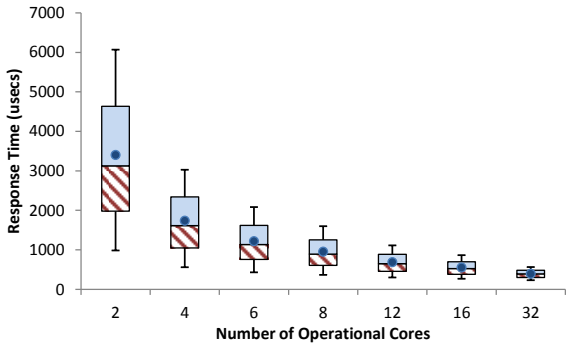

Figure 78: Scale_Input Experiment: Class 2 response time decreases as the input rate increases because of batch processing.

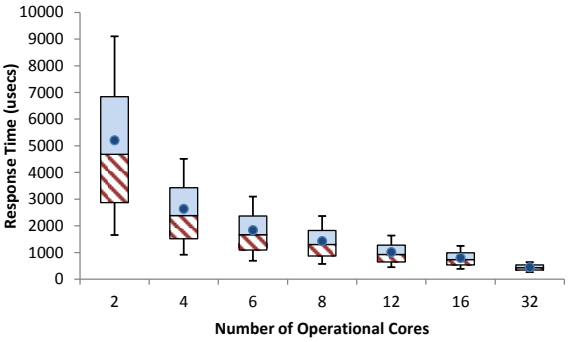

Figure 79: Scale_Input Experiment: Class 3 response time decreases as the input rate increases because of batch processing.

6.3.2.2 Scale_Input Experiment (Figures 76 - 81) In this experiment, we evaluate the scalability of the MBD scheduler when the input rate increases proportionally to the number of operational cores as shown in Figure 76. The workload used for this experiment is Workload 5G_2 as shown in Table 8. The input rate for this workload running on two operational cores is 600 tuples/second for each of the data sources. Figures 77, 78, 79, 80, and 81 show the response times of

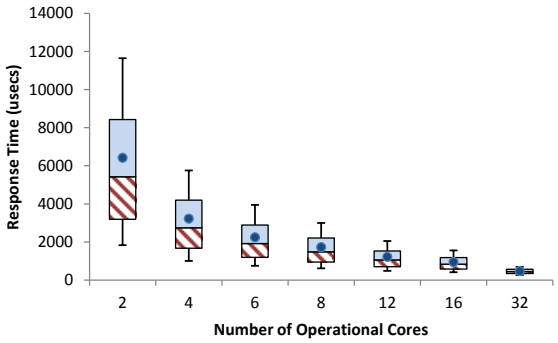

Figure 80: Scale_Input Experiment: Class 4 response time decreases as the input rate increases because of batch processing.

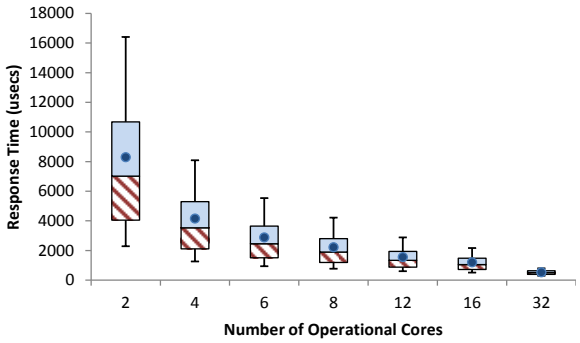

Figure 81: Scale_Input Experiment: Class 5 response time decreases as the input rate increases because of batch processing.

each class as the number of operational cores and input rate increases. Interestingly, the response time for all of the classes decreases as the input rate and the number of cores increase. Unlike the Scale_CQs experiment, in the Scale_Input experiment, the load in the system does not increase proportionally to the increase in the input rate. This is due to the switching overhead costs when executing operators and the consolidation of these costs during batch execution. The MBD scheduler takes advantage of batch processing by allowing each operator and CQ to process all of its input queue when scheduled. This spreads the overhead cost of the operator over multiple tuples and reduces the relative load of the system while running on more operational cores. Thus, in this experiment, the relative load to capacity in each class becomes lighter as the number of operational cores increases. This results in lower response times of the CQs of all the classes. Consequently, the MBD scheduler scales very well as the input rate increases proportionally to the number of operational cores.

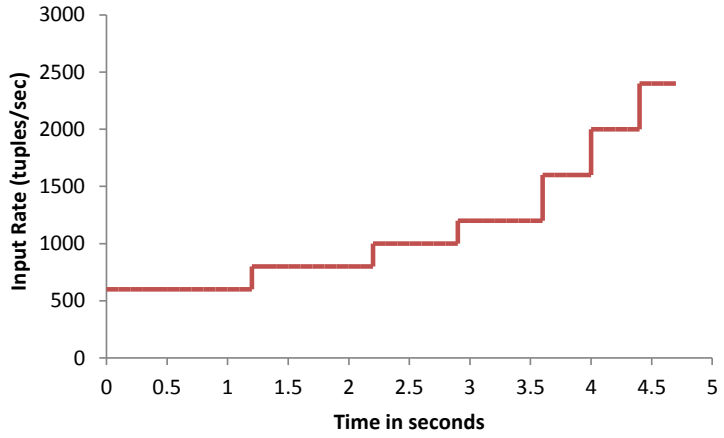

Figure 82: Scale_Overld Experiment: the input rate increases from 600 to 2400 tuples/sec.

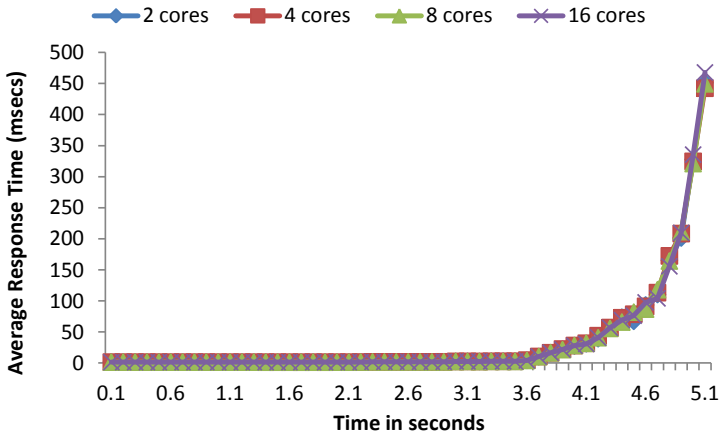

Figure 83: Scale_Overld Experiment: the average response time increases the same way under either of the workloads.

6.3.2.3 Scale_Overld Experiment (Figures 82 & 83) We evaluate the scalability of the MBD scheduler as the input rate increases over time as shown in Figure 82. We run Workloads O_2, O_4, O_8, and O_16 shown in Table 10 where the number of CQs increases proportionally to the number of cores. Figure 83 shows that the average response time follows the same pattern. It starts increasing drastically at the 3.6 second mark. This corresponds to 1600 tuples per second

input rate. After that point, the average response time increases exponentially indicating that the CQs are overloaded. Thus, the load increases as the CPU capacity increases proving that the MBD scheduler scales well.

6.3.2.4 Scalability Experiments Summary The experiments run in this section show that the MBD scheduler scales very well on multiple dimensions. Scale_CQs experiment shows that the MBD scheduler is scalable over the number of CQs. When the number of CQs increases proportionally to the number of cores, the MBD scheduler takes advantage of the new opportunities for parallelization to schedule the CQs over the number of cores. Scale_Input experiment shows that the MBD scheduler also scales when the input rate increases proportionally to the number of cores. In fact, the load with the same workload, higher input rate and more cores decreases under the MBD scheduler resulting in a better response time. Last, Scale_Overload shows that the MBD scheduler scales the CPU capacity when the number of CQs increases proportionally to the number of cores. The workload overloads at the same input rate. These experiments show that the MBD scheduler is appropriate for multi-core environments.

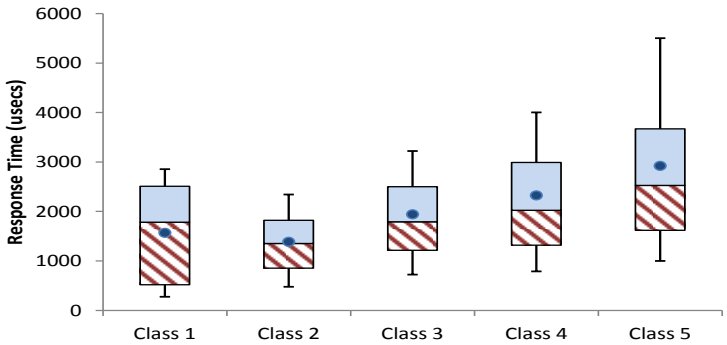

Figure 84: LowInput Class 1 Experiment: Class 1 has a higher response time than Class 2.

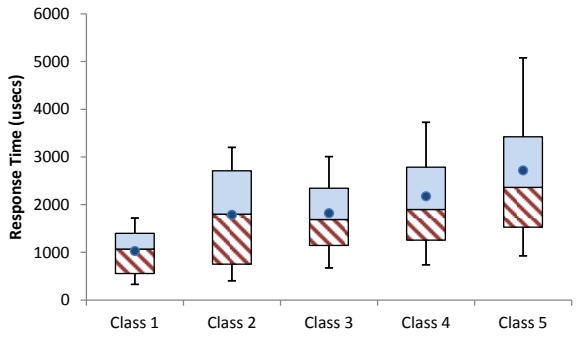

Figure 85: LowInput Class 2 Experiment: Class 2 has a higher response time at the 75-percentile than Class 3.

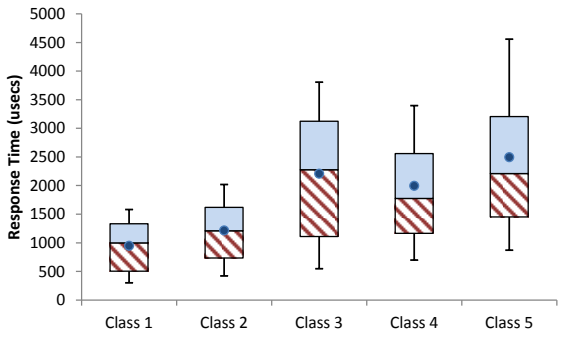

Figure 86: LowInput Class 3 Experiment: Class 3 has a higher response time than Class 4.

6.3.3 Priority Inversion Experiments

In this section, we evaluate the priority inversion modules described in Section 6.1.5. We designed experiments that induce priority inversion for the MBD scheduler when the priority inversion modules are disabled. Then we enable the priority inversion modules, run the workload again and compare the results. Recall that the MBD scheduler has four priority inversion modules namely:

- **Decrease Violator Priority (DVP):** decrements the frequency of the class that is violating the priority semantics.
- **Increase Victim Priority (IVP):** increments the frequency of the class whose response time is higher than that of a lower class.
- **Decrease Violator Side Priority (DSP):** determines the classes that are violating the frequency semantics and decrements the frequency of all of the classes that have equal or lower priority.
- **Increase Violator Side Priority (ISP):** determines the classes whose response time is higher that their lower priority classes and increments the frequency of all of the classes that have equal or higher priority.

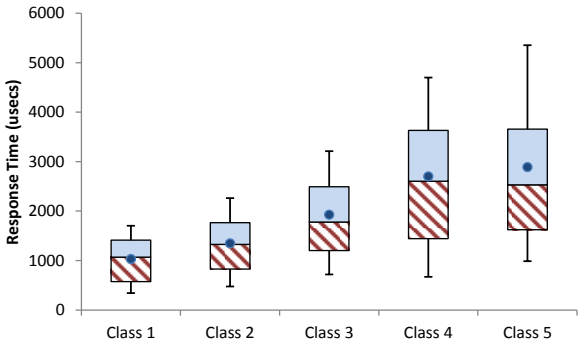

Figure 87: LowInput Class 4 Experiment: Class 4 has a similar response time to Class 5.

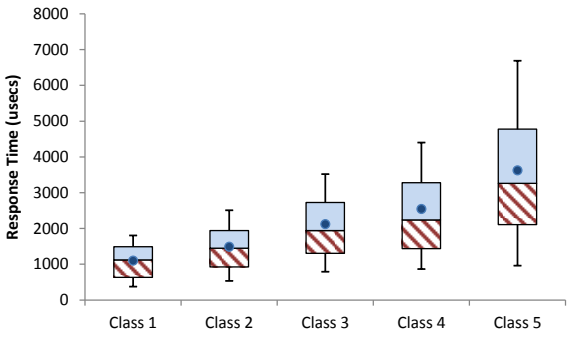

Figure 88: LowInput Class 5 Experiment: Class 5's response time increases, no priority inversion.

For these experiments, we use Workload 5G_4 shown in Table 8 as run on four cores. Figure 84 shows the performance of the MBD scheduler when all of the classes are run with a 1200 tuples/second input rate with the exception of Class 1 which is run with a 500 tuples/second input rate. Even with thirty percent of the scheduling points given to Class 1, Class 2 still has a better average response time. This results in priority inversion which violates the class semantics.

Then, we change the workload by lowering Class 2's input rate to 500 tuples/second while keeping 1200 tuples/second as an input rate for classes 1, 3, 4 and 5. Figure 85 shows that the MBD schedule does not result in priority inversion at the average but there is priority inversion at the seventy-five percentile for this workload.

Figure 86 shows the performance of the MBD scheduler when all of the classes are run with a 1200 tuples/second input rate with the exception of Class 3 which is run with a 500 tuples/second input rate. Just like the case with Class 1, this results in priority inversion where Class 4 has a better average response time than Class 3. This is because Class 3 has a very low input rate relative to the other classes which decreases the number of tuples ready to execute at its scheduling points and reduces its share of CPU time.

Figures 87 and 88 show the performance of the MBD scheduler when all of the classes are run with a 1200 tuples/second input rate with the exception of class 4 and 5 respectively which is run

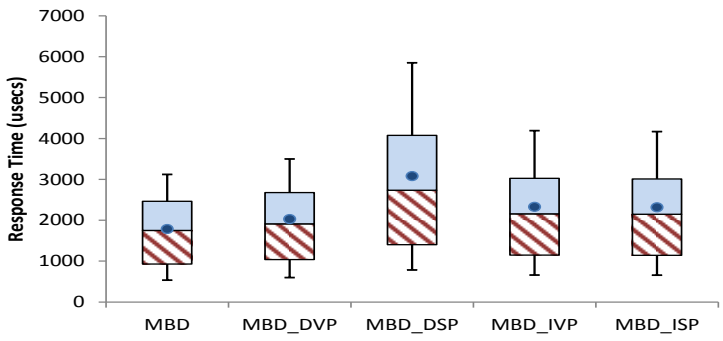

Figure 89: LowInput Class 1 Experiment: MBD_DVP has the closest response time to when no modules are enabled (MBD).

with a 500 tuples/second input rate. In these two cases, the MBD scheduler does not exhibit any priority inversion and is capable of handling these workloads.

For the workloads where lowering one of the classes input rates resulted in priority inversion, we enabled the priority inversion detection modules discussed in Section 6.1.5. Thus, we run Workload 5G_4 when Class 1 or Class 3 have a low input rate of 500 tuples/second and enable the modules at the average. Then we run Workload 5G_4 again when Class 1, Class 2 or Class 3 have a low input rate and enable the modules at the 75-percentile.

6.3.3.1 LowInput Class 1 Experiment (Figures 89 - 91) For this experiment, the priority inversion detection modules were set to detect at the average. We run Workload 5G_4 as shown in Table 8 with Class 1's input rate set to 500 tuples per second. As discussed in Section 6.3.3, the MBD scheduler results in priority inversion for this workload. However, once the priority inversion detection modules were enabled, none of them results in priority inversion at the average. They varied in the amount of time it took to converge to a schedule without priority inversion, the number of updates to the schedule, and the resulting response time. Figure 89 shows the weighted response times of the workload once the modules were enabled as compared to the MBD scheduler. MBD_DVP results in a response time closest to that of MBD and the lowest among the priority inversion detection modules because it gradually corrects the schedule by decreasing the violating

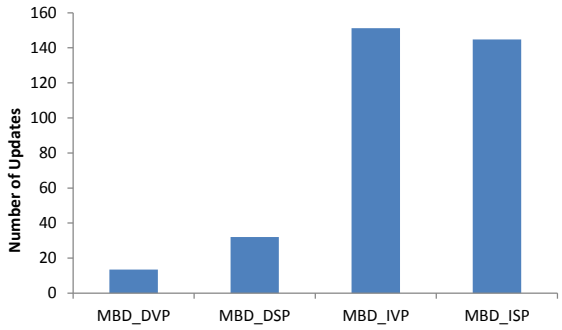

Figure 90: LowInput Class 1 Experiment: MBD_DVP reaches a stable schedule with the least number of updates.

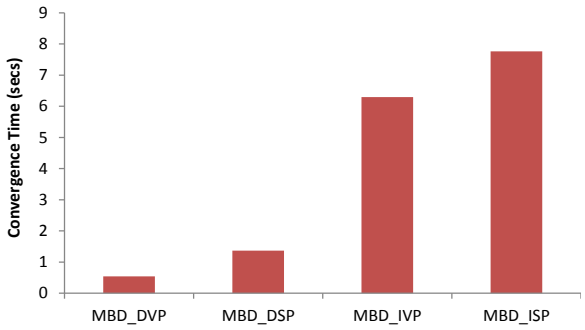

Figure 91: LowInput Class 1 Experiment: MBD_DVP converges the fastest out of the priority inversion modules.

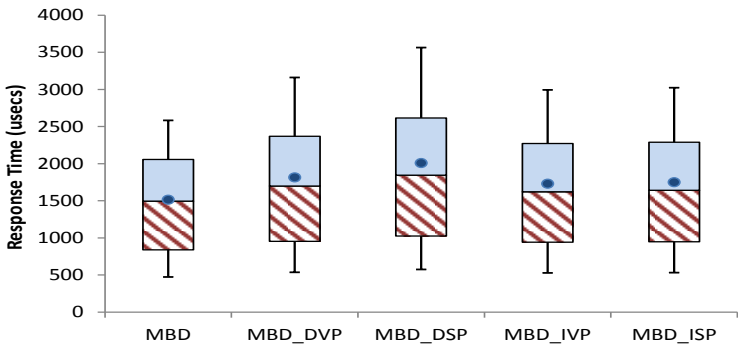

Figure 92: LowInput Class 3 Experiment: the MBD_IVP module has the closest response time to when no modules are enabled (MBD).

class's priority by one every time. MBD_DSP results in the highest response time because it penalizes not only the violator but also all the classes with a lower priority than the violator. This approach increases the response time of the lower priority classes and consequently the response time of the whole workload. As shown in Figure 90, MBD_DVP converges to a stable schedule with the least number of updates compared to the other modules. It also converges the fastest as shown in Figure 91 making it an ideal candidate to adapt to changes in the input rate.

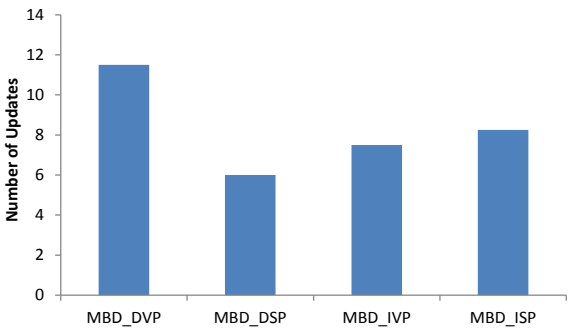

Figure 93: LowInput Class 3 Experiment: MBD_DSP reaches a stable schedule with the least number of updates.

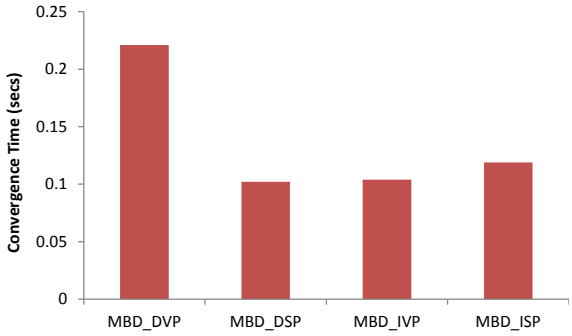

Figure 94: LowInput Class 3 Experiment: MBD_DSP converges the fastest out of the priority inversion modules.

6.3.3.2 LowInput Class 3 Experiment (Figures 92 - 94) In this experiment, we set Class 3's input rate to be 500 tuples per second for Workload 5G_4. As described in Section 6.3.3, the MBD scheduler results in priority inversion as shown in Figure 86. Once we enable the priority inversion detection modules at the average level, the scheduler detects the priority inversion and corrects it. Figure 92 shows the response time when the different modules are enabled. For this workload, MBD_IVP has the closest response time to the MBD scheduler. However, Figures 93 and 94 show that MBD_DSP converges the fastest out of all of the modules and has the lowest number of updates. This is because MBS_DSP takes the most drastic approach of decreasing the CPU portion of the side that is violating priority semantics. This results in an increase in the response time of the classes on the violating side and thus a fast correction to the priority inversion.

6.3.3.3 Low75 Class 1 Experiment (Figures 95 - 98) We lower the input rate of Class 1 to be 500 tuples per second for Workload 5G_4 just as in experiment LowInput Class 1. However, we enable the priority inversion modules at the seventy-five percentile. Figure 95 shows the response time for each of the modules. The performance of the modules is the same as the LowInput experiment: MBD_DSP has the highest response time while MBD_DVP has the lowest. However, unlike

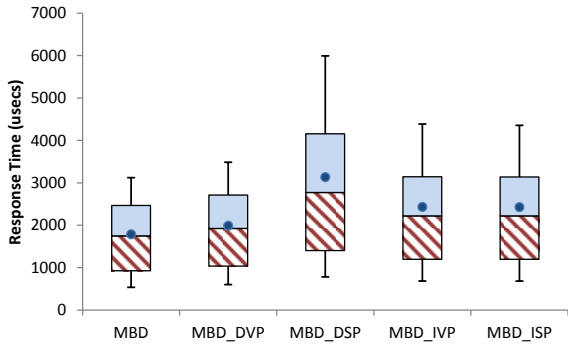

Figure 95: Low75 Class 1 Experiment: the MBD_DVP module has the closest response time to when no modules are enabled (MBD).

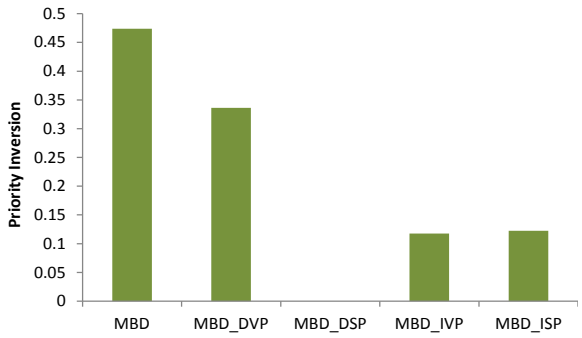

Figure 96: Low75 Class 1 Experiment: only the MBD_DSP module successfully prevents the priority inversion at the 75-percentile.

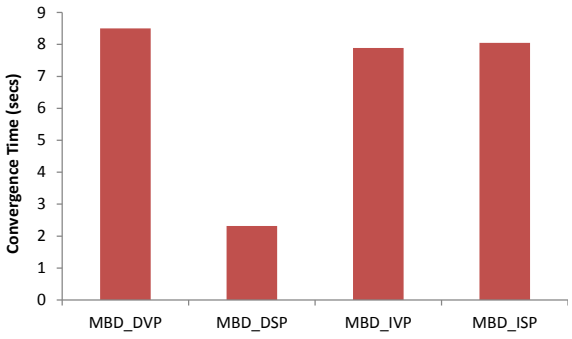

Figure 97: Low75 Class 1 Experiment: MBD_DSP converges the fastest out of the priority inversion modules.

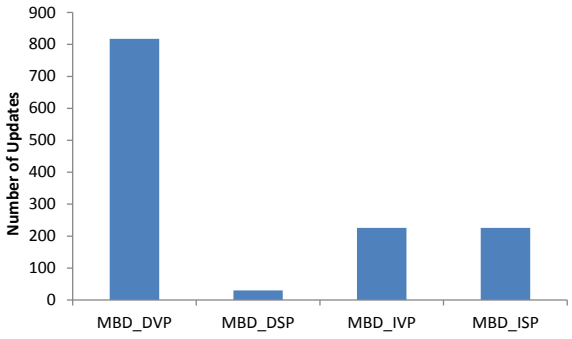

Figure 98: Low75 Class 1 Experiment: MBD_DSP reaches a stable schedule with the least number of updates.

the LowInput Class 1 experiment, only MBD_DSP successfully corrects the priority inversion at the seventy-five percentile as shown in Figure 96. Figures 97 and 98 show that MBD_DSP also converges the fastest and with the lowest number of updates to reach a stable schedule.

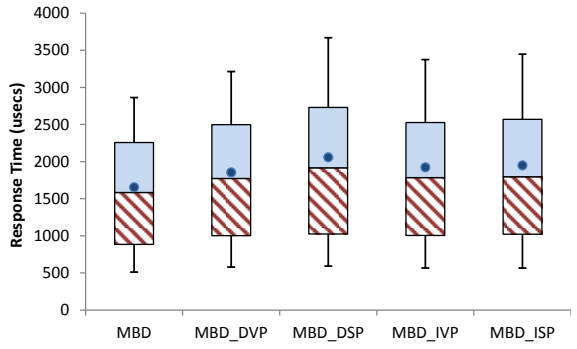

Figure 99: Low75 Class 2 Experiment: the MBD_DVP module has the closest response time to when no modules are enabled (MBD).

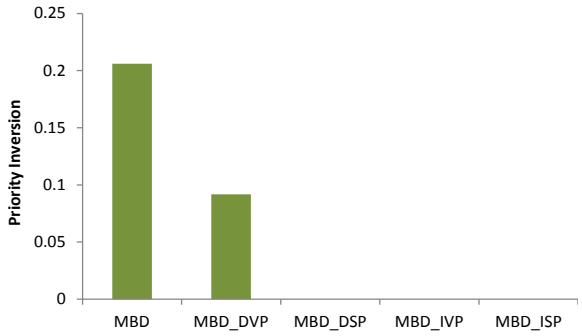

Figure 100: Low75 Class 2 Experiment: only the MBD_DVP module cannot prevent the priority inversion at the 75-percentile.

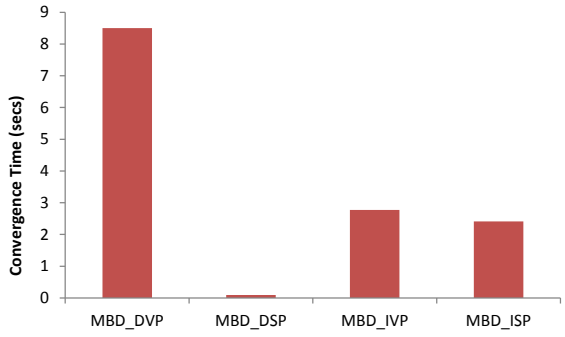

Figure 101: Low75 Class 2 Experiment: MBD_DSP converges the fastest out of the priority inversion modules.

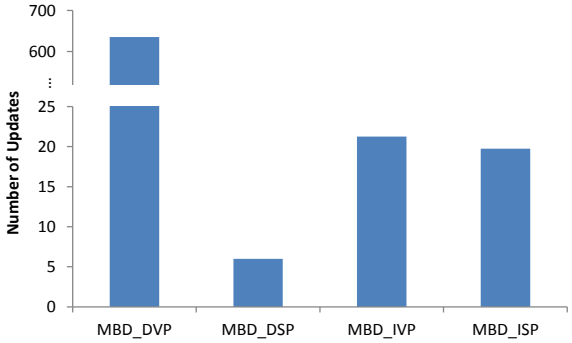

Figure 102: Low75 Class 2 Experiment: MBD_DSP reaches a stable schedule with the least number of updates.

6.3.3.4 Low75 Class 2 Experiment (Figures 99 - 102) We lower the input rate of Class 2 to be 500 tuples per second for Workload 5G_4 and we enable the priority inversion modules at the seventy-five percentile. Figure 99 shows the response time for each of the modules. The performance of the modules is the same as the LowInput experiments: MBD_DSP has the highest response time while MBD_DVP has the lowest. All of the modules with the exception of

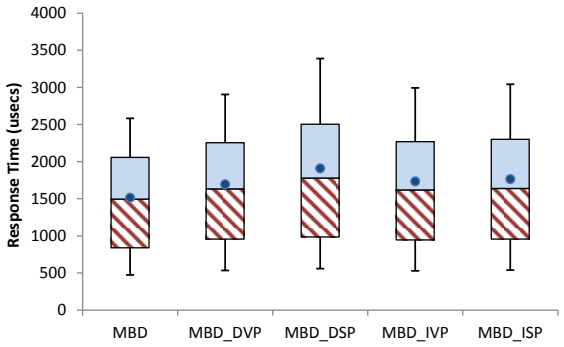

Figure 103: Low75 Class 3 Experiment: the MBD_IVP module has the closest response time to when no modules are enabled (MBD).

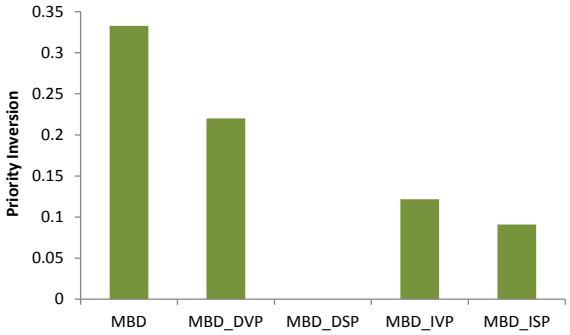

Figure 104: Low75 Class 3 Experiment: only the MBD_DSP module successfully prevents the priority inversion at the 75-percentile.

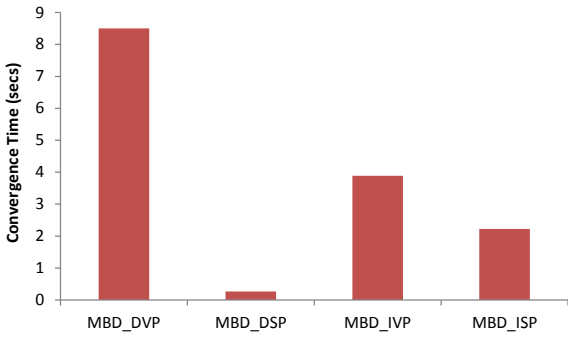

Figure 105: Low75 Class 3 Experiment: MBD_DSP converges the fastest out of the priority inversion modules.

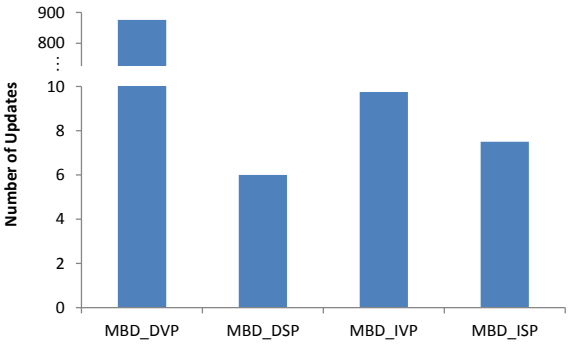

Figure 106: Low75 Class 3 Experiment: MBD_DSP reaches a stable schedule with the least number of updates.

the MBD_DVP module successfully correct the priority inversion at the seventy-five percentile as shown in Figure 100. Figures 101 and 102 show that MBD_DSP also converges the fastest and with the lowest number of updates to reach a stable schedule.

6.3.3.5 Low75 Class 3 Experiment (Figures 103 - 106) For this experiment, we decrease the input rate of Class 3 to be 500 tuples per second for Workload 5G_4 and we enable the priority inversion modules at the seventy-five percentile. Figure 103 shows the response time for each of the modules. The performance of the modules is the same as the LowInput Class 3 experiment: MBD_DSP has the highest response time while MBD_DVP has the lowest. Only the MBD_DSP module successfully corrects the priority inversion at the seventy-five percentile as shown in Figure 104. Figures 105 and 106 show that MBD_DSP again converges the fastest and with the lowest number of updates to reach a stable schedule. This result is consistent with all of the 75-percentile experiments in this section.

6.3.3.6 Priority Inversion Experiments Summary These experiments show that the best module to use with the MBD scheduler is the DSP module that guarantees no priority inversion at the level it is enabled. In addition, it converges the fastest to a stable scheduler making it ideal for our system. This is crucial at higher percentile levels because the DSP module takes a drastic approach to the violator of priority inversion while still maintaining a short MBD schedule. At the average level, the DVP module could be used because it adjusts the schedule slowly and maintains a low response time. In order to evaluate the performance of these modules in combination with the other MBD modules, we run both the DSP and the DVP modules when a priority inversion module is needed at the average level in the rest of the experiments.

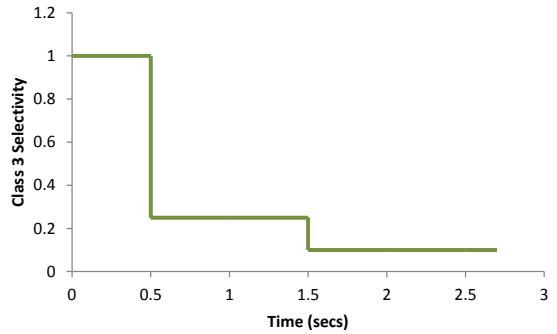

Figure 107: Sel-Change Experiment: the selectivity of the selection operators in Class 3 decreases over time to become 0.1.

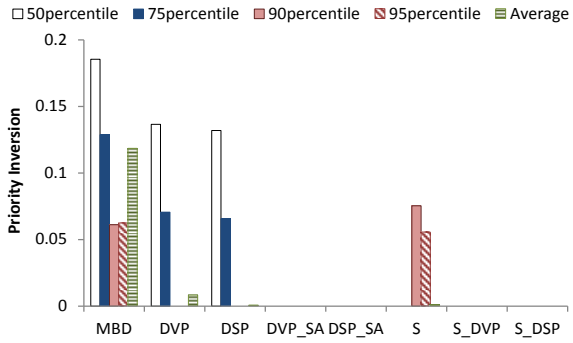

Figure 108: Sel-Change Experiment: DVP_SA, DSP_SA, S_DVP and S_DSP detect and correct the priority inversion.

Figure 109: Sel-Change Experiment: the DSP_SA module converges the fastest.

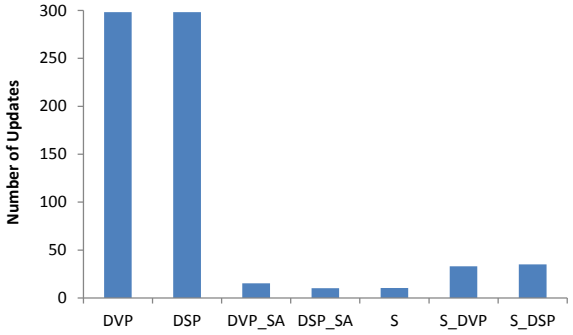

Figure 110: Sel-Change Experiment: DSP_SA reaches a stable schedule with the least updates.

6.3.4 Selectivity Experiment

As mentioned in Section 6.1.6, the selectivity of operators could change during runtime making the MBD schedule obsolete. The MBD scheduler can react either by only updating the selectivities or by updating both the selectivities and the frequencies in the event of priority inversion. In this section, we evaluate the following modules:

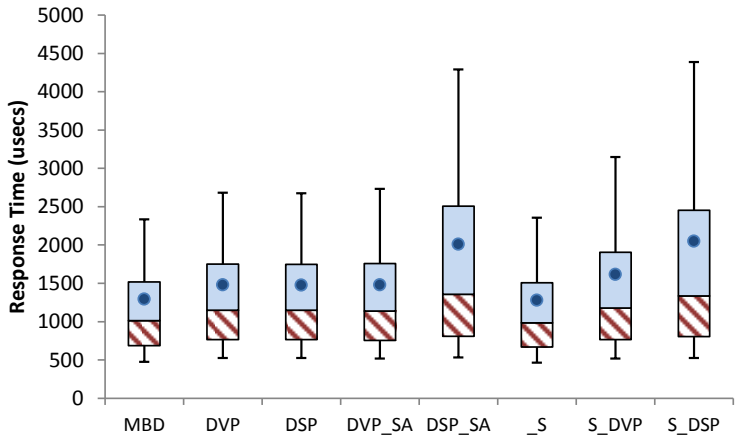

Figure 111: Sel-Change Experiment: the _S module has the lowest response time. Out of the modules with zero priority inversion, the DVP_SA module has the lowest response time.

- **_S:** updates the selectivities only without changing the class frequencies. A change in selectivities is detected only if the selectivities have changed by more than ten percent. We chose the threshold value of ten percent after conducting experiments with multiple values. A threshold lower than ten percent made the scheduler susceptible to more frequent schedule rebuilds because selectivities could fluctuate slightly during runtime. A higher threshold would not allow the scheduler to react changes in selectivity in a responsive manner.
- **DVP_S:** updates the selectivities only without changing the class frequencies when a change in selectivities is detected. When priority inversion is detected, then only the frequencies are adjusted according to the DVP module.
- **DSP_S:** updates the selectivities only without changing the class frequencies when a change in selectivities is detected. When priority inversion is detected, then only the frequencies are adjusted according to the DSP module.
- **DVP_SA:** updates both the selectivities and the frequencies according to the DVP module before rebuilding the MBD schedule when priority inversion is detected.
- **DSP_SA:** updates both the selectivities and the frequencies according to the DSP module before rebuilding the MBD schedule when priority inversion is detected.

6.3.4.1 Sel-Change Experiment (Figures 107 - 111) For this experiment, we use Workload 5I as shown in Table 8. However for Class 3, we change the selectivity of the selection operators according to Figure 107. We also add a selection operator, whose selectivity follows Figure 107, to all of the incoming input streams for the join CQs in Class 3. Workload 5I is run on six cores with an additional core dedicated to receiving input streams. Figure 108 shows that the MBD scheduler results in priority inversion when none of the priority inversion modules are enabled. Once the selectivity detection is enabled MBD_S, it is still unable to completely remove the priority inversion. The priority inversion modules DSP and DVP cannot fix the priority inversion either when the selectivity detection module is disabled as shown in Figure 108.

Thus, both the priority inversion module and the selectivity modules need to be enabled in order to detect any changes in the workload and effectively react to them. As mentioned in Section 6.1.6, there are two ways to integrate the selectivity module with the priority inversion module namely _S and _SA. For this experiment, we combine the DSP and the DVP modules with the selectivity module using _S and _SA. As shown in Figure 108, DSP_SA, S_DSP, DVP_SA and S_DVP successfully detect the priority inversion and correct it. However, as shown in Figures 110 and 109, the MBD scheduler with the DSP_SA module is the fastest at reaching a stable schedule and with the least number of updates because of its drastic decrease in the violator's priority side and adjusting both the selectivities and CQ priorities when a violation is detected. However, it results in a slight increase of the weighted response time of the system as shown in Figure 111. The DVP_SA module is the second fastest at reaching a stable schedule because of its gradual update of frequencies. Consequently, it has a lower weighted response time as shown in Figure 111.

6.3.4.2 Selectivity Experiment Summary In this section, the _SA method performed the best when selectivities change during runtime. Combined with DVP to become DVP_SA, it produced the best response time. However, the MBD scheduler converged the fastest while running the DSP_SA module.

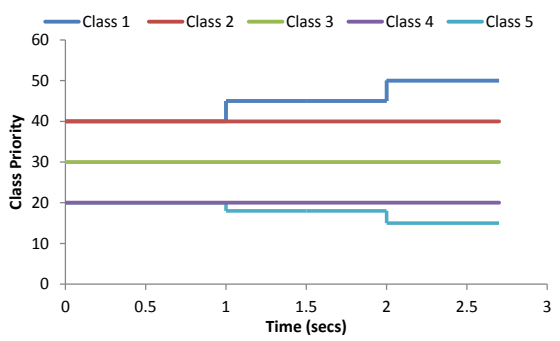

Figure 112: Pr-Change Experiment: classes (1, 2) and (4,5) start with equal priority. Then, Class 1's becomes 50 and Class 5's 15.

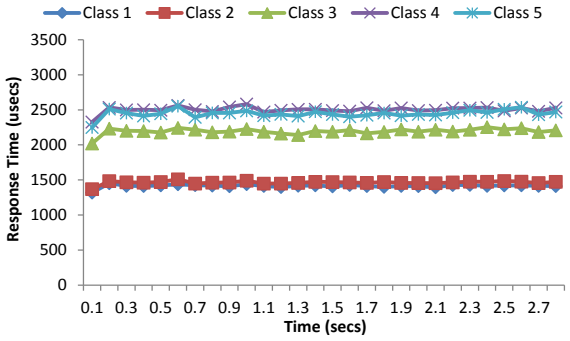

Figure 113: Pr-Change Experiment: the MBD scheduler without its priority module cannot distinguish classes 1 and 2 or 4 and 5.

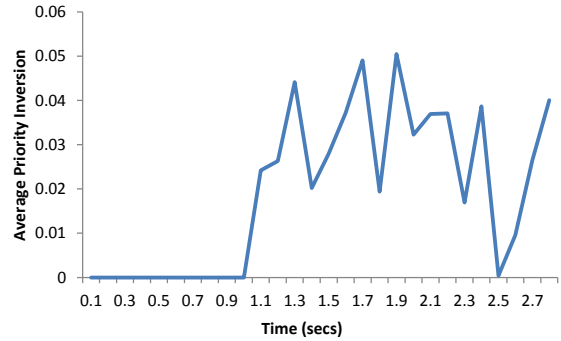

Figure 114: Pr-Change Experiment: the MBD scheduler without its priority module has priority inversion.

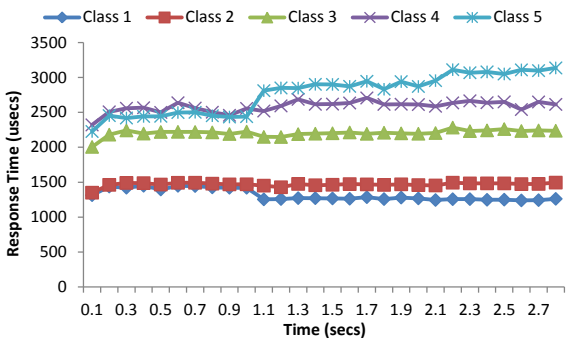

Figure 115: Pr-Change Experiment: MBD_UP has no priority inversion by detecting the priority changes and adjusting the MBD schedule.

6.3.5 Priority Changing Experiments

To evaluate how the MBD scheduler adapts to class priorities changing during runtime, we devise two experiments: Pr-Change and Pr-Change-Merge using Workload 5I shown in Table 8.

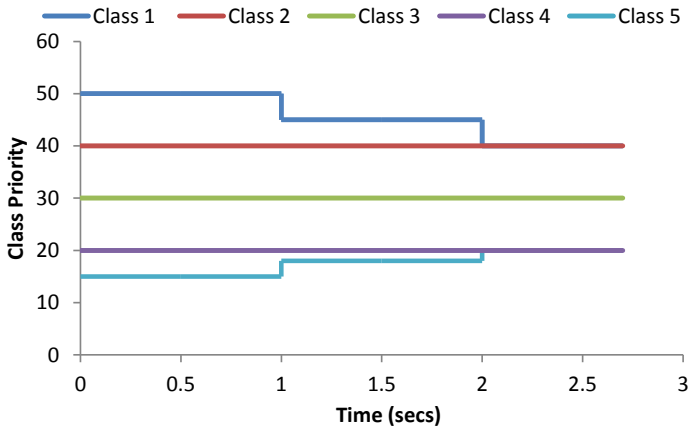

Figure 116: Pr-Change-Merge Experiment: classes (1, 2) and (4,5) start with different priorities. Class 1's priority decreases from 50 to 40. Class 5's priority increases from 15 to 20.

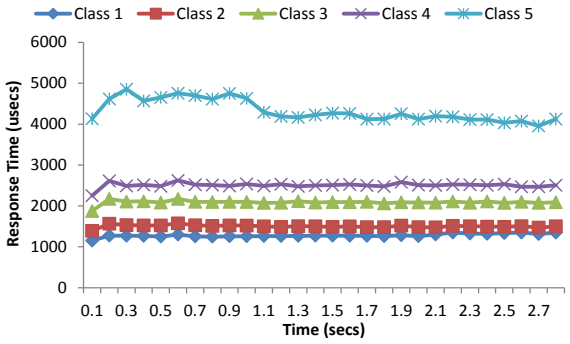

Figure 117: Pr-Change-Merge Experiment: MBD without a priority module keeps Class 5's response time higher than Class 4's.

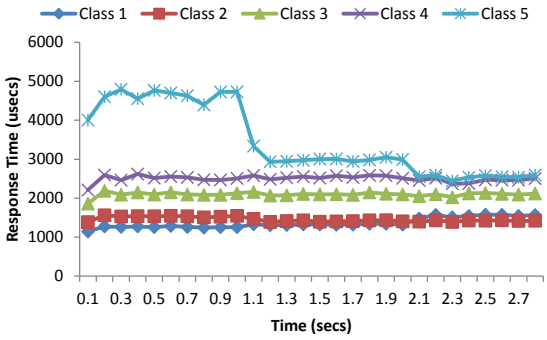

Figure 118: Pr-Change-Merge Experiment: MBD_UP detects the changes in priority thus lowering Class 5's response time.

6.3.5.1 Pr-Change Experiment (Figures 112 - 115) For this experiment, we modify the priorities as shown in Figure 112. The performance of the MBD scheduler when it is not aware of the priority changes is shown in Figure 113. The MBD scheduler does not detect that classes 1 and 2 have different priorities at 1 second. Thus, it treats them equally by allocating the same CPU time to each class. This also applies to classes 4 and 5. As a result, the workload suffers from

priority inversion as shown in Figure 114. Once the priority detection module (_UP) is enabled, the MBD_UP scheduler detects the change in priorities at one and at two seconds. It updates the MBD schedule according to the new priorities and it results in no priority inversion as shown in Figure 115.

6.3.5.2 Pr-Change-Merge Experiment (Figures 116 - 118)

We modify the priorities as shown in Figure 116. The workload starts with five classes where each class has a different priority. During runtime, two pairs of classes namely classes 1 & 2 and classes 4 & 5 end up with the same priority. The performance of the MBD scheduler when the priority detection module is disabled is shown in Figure 117. The MBD scheduler does not detect the change in priority and keeps treating Class 4 as having a higher priority than Class 5. However, once the priority module (_UP) is enabled, the MBD_UP scheduler detects the change in priority and treats classes 4 & 5 and classes 1 & 2 as equal. This results in a lower response time for Class 5 as shown in Figure 118.

6.3.5.3 Priority Changing Experiments Summary

Thus, the MBD scheduler also needs to react to changes in the priorities of the classes during runtime by having its priority module (_UP) enabled. When two classes start by being equal then become different, the _UP module prevents priority inversion. When two different classes change their priorities so that they have the same priority, the _UP module reduces the response time of the class with the lower priority to make them similar.

6.3.6 Scheduling Sources Experiment

In this section, we evaluate other scheduling options for the source operators when allocated two cores. This does not apply to DSMSs that have a single routing module to input the tuples. Under AQSIOS, however, the tuples are input to the system via source operators. Thus, these operators can execute on multiple cores. To schedule these source operators, we compare the MORR scheduler described in Section 6.2.1 and an MBD-variant scheduler. The latter scheduler uses the MBD schedule but at the operator-level where the operator priority is equal to its class priority.

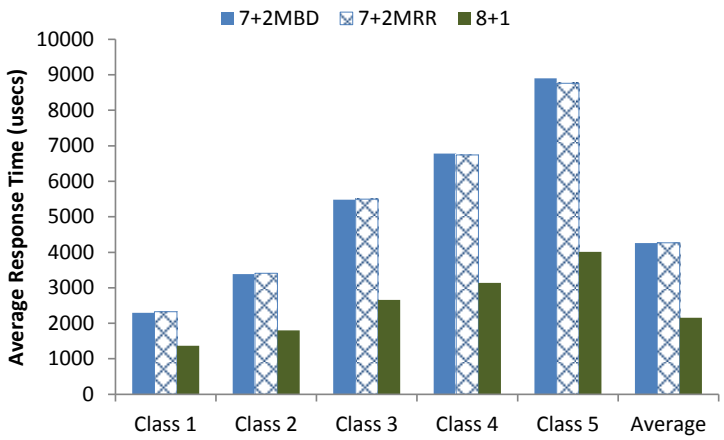

Figure 119: Scheduling Sources Experiment: the response time is the lowest when the number of cores used for the operational and output operators is 8 and 1 core for the sources.

We run Workload 5S_8 which has four times the CQs in Workload 5S_2 shown in Table 9. We allocate for the source operators two cores instead of one and we compare it to the result shown in the Scale_CQs experiment (Section 6.3.2). When two cores are allocated to the sources, 7+2MRR shows the performance when an MORR scheduler is used to schedule the sources. 7+2MBD shows the results when an MBD-variant scheduler is used for the sources. 8+1 is when the MBD scheduler is run with only one core allocated to the sources. As shown in Figure 119, there is no significant difference between the performance of the MBD and the MORR schedulers for the source operators. The best performance for this workload is achieved when 8+1 is run because the operational and output operators get more CPU time to execute longer batches of tuples produced by the sources. Recall that we use the input rate of 4800 tuples/second such that the workload is heavily loaded but not overloaded. A higher input rate would provide longer batches of tuples but would also overload the operational and output operators.

The choice of source scheduler for this experiment did not make any visible difference because the sources' output rate is highly dependent on the input rate of the data streams. As long as the source scheduler does not starve any of the source operators, the MBD scheduler responsible for the operational and output operators determines the order of processing of the sources' output tuples.

Thus, the source scheduler needs to schedule a source operator before its downstream operational operators execute next. This requires the two schedulers to coordinate when a CQ executes which could be costly because the source scheduler cannot determine when tuples would be available for the source operator. Also, it is common in our target applications that the source operators are shared among many CQs that belong to many different CQ classes. Under these conditions, round robin scheduling provides the best performance even if more cores are needed to execute the sources. A load manager could be employed to instruct the scheduler on the number of cores to use for the sources and for the operational and output operators.

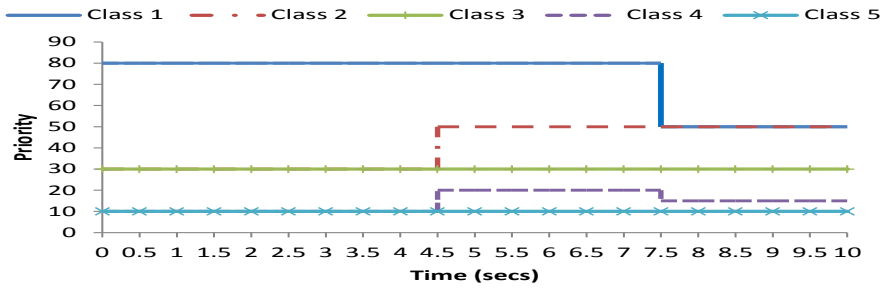

Figure 120: PIAT Experiment: the priorities change during runtime. Only Class 3's priority does not change.

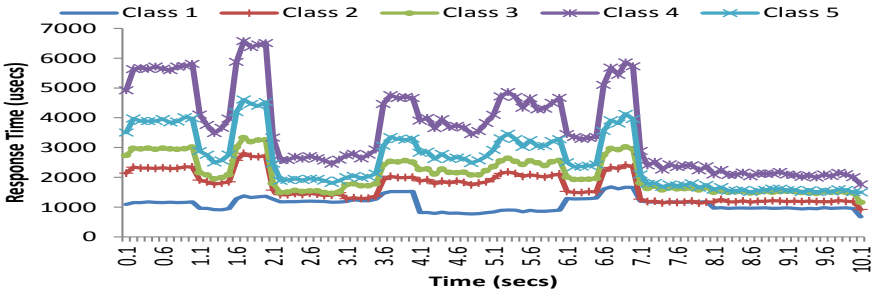

Figure 121: PIAT Experiment: Class 5's input rate has two spikes of 2000 tuples/sec. Class 1's input rate fluctuates over time.

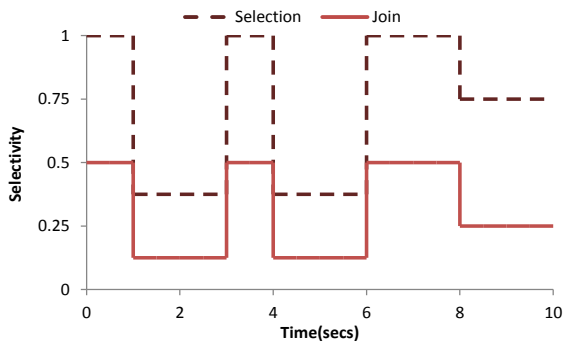

Figure 122: PIAT Experiment: the selectivities of the selection operators and the joins in Class 2 fluctuates over time.

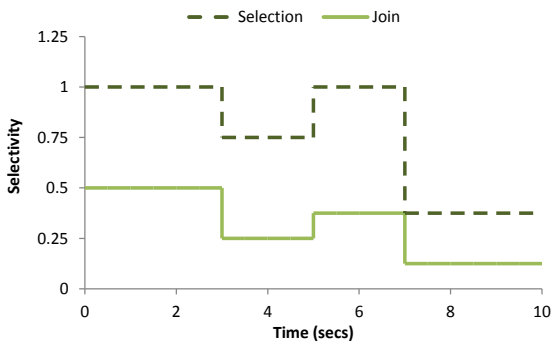

Figure 123: PIAT Experiment: the selectivities of the selection and join CQs in Class 3 change over time.

6.3.7 Putting It All Together (PIAT Experiment: Figures 120 - 135)

In this experiment, the selectivities, priorities and input rates of the classes change dynamically during runtime. Workload 5L is shown in Table 11. It is run on six operational cores and one core dedicated to source operators. The changes in priority are shown in Figure 120. During the ten second run of the workload, Class 1 changes priority from originally 80 to 50 at 7.5 seconds. Class 2 changes priority from 30 to 50 at 4.5 seconds. Class 4 changes priority twice: once at 4.5 seconds from 10 to 20 then at 7.5 seconds from 20 to 15.

The changes to the input rate are shown in Figure 121. Classes 2, 3 and 4 do not experience a change in the input rate so they follow a Poisson distribution at 1200 tuples/second all the time. However, classes 1 and 5 experience a change in the input rate resulting in a decreased load for Class 1 and spikes in the load for Class 5.

The selectivities of the joins and selection operators of classes 2 and 3 change during runtime as shown in Figures 122 and 123. This affects the number of tuples outputted by these two types of CQs.

Workload 5L has two shared CQs among all of the classes. There is a union CQ and an except CQ. Because these CQs are shared by all the classes, they get promoted to Class 1 during runtime.

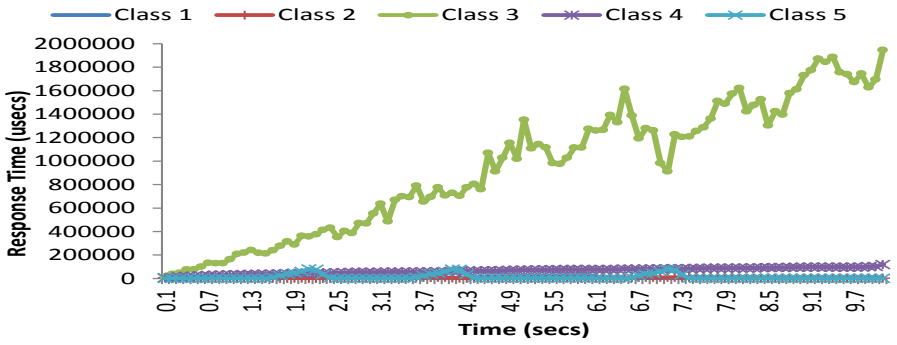

Figure 124: PIAT Experiment: PCT starves Class 3 increasing its response time above 2 seconds.

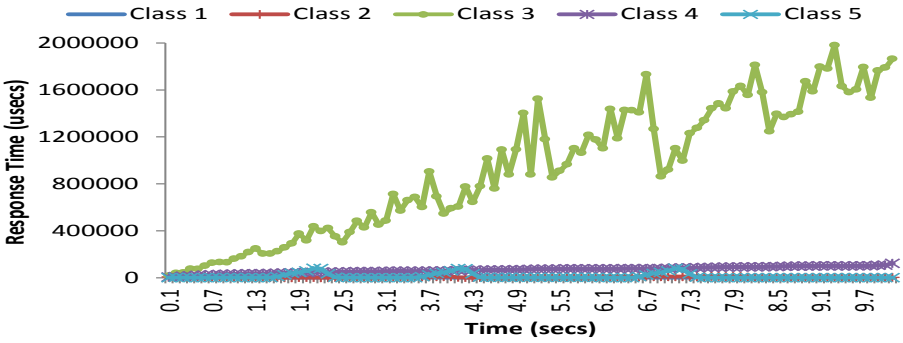

Figure 125: PIAT Experiment: PCT_S starves Class 3 increasing its response time to 2 seconds.

However, with a priority inversion module, this should not result in priority inversion.

Figures 124 and 125 show the running of Workload 5L under the PCT and PCT_S schedulers respectively. Neither of these schedulers can take full advantage of the cores and thus result in overloading Class 3 which is the heaviest class in this workload. The response time of Class 3 ends up continuously increasing and reaches two seconds. This results in extreme priority inversion.

Figure 126 shows the performance of the MRR scheduler over time when Workload 5L is run. The MRR scheduler is not aware of the classes and their priorities and thus optimizes for the average response time. As shown in Figure 126, none of the classes overload but the high priority classes do not benefit under this scheduler. Similarly, the MORR scheduler does not distinguish

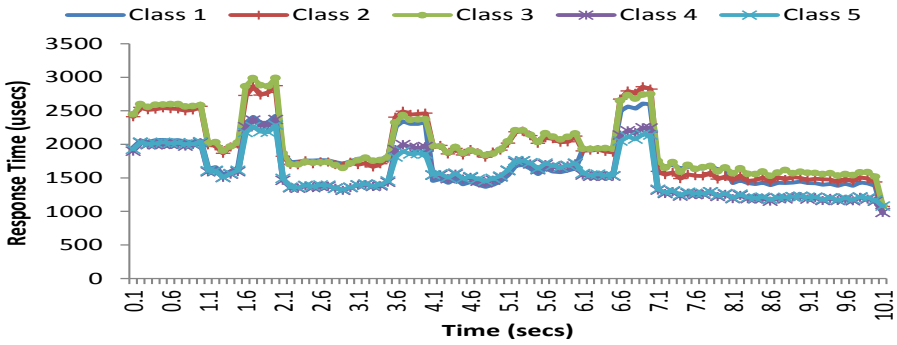

Figure 126: PIAT Experiment: Under MRR, classes 4 and 5 have the lowest response time. Class 3 has the highest.

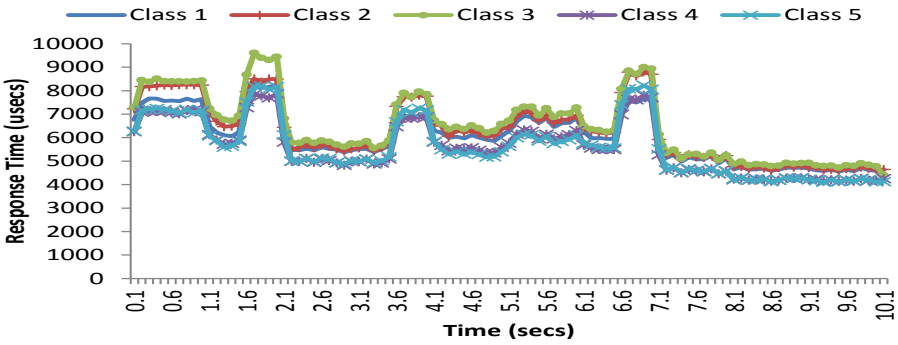

Figure 127: PIAT Experiment: Under MORR, classes 4 and 5 have the lowest response time. Class 3 has the highest.

between the classes as shown in Figure 127. However, the response time under MORR is higher than that under MRR because it is an operator-level scheduler and does not pipeline the execution of tuples up the CQ network. Both the MRR and MORR schedulers do not avoid priority inversion as shown in Figure 135.

Figure 128 shows the response time over time when the CQs are scheduled using the ML scheduler. The ML scheduler cannot provide any guarantees as to when a CQ can execute, even worse, it forces a CQ to wait behind other CQs even if it is higher priority. This results in an unpre-

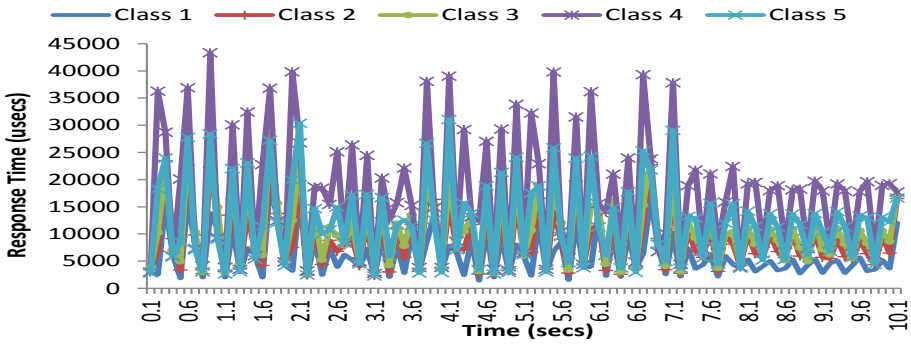

Figure 128: PIAT Experiment: Under ML, the response times of the classes fluctuate dramatically due to the random effects.

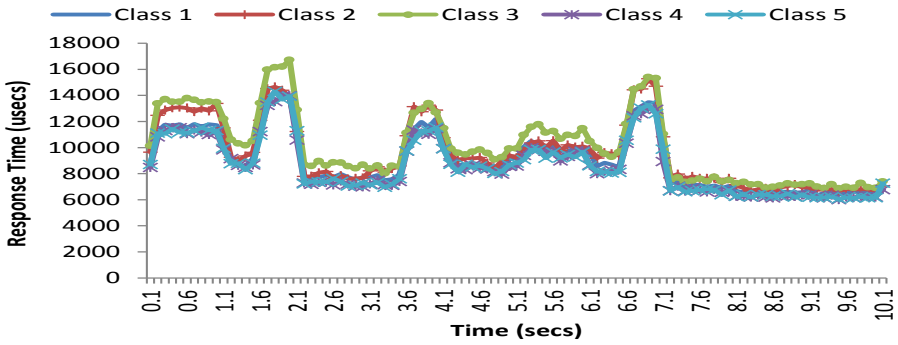

Figure 129: PIAT Experiment: Under MOL, classes 4 and 5 have the lowest response time. Class 3 has the highest.

dictable performance due to its dependency on random number generation. The MOL scheduler performs better as shown in Figure 129. Because the MOL scheduler is an operator level scheduler, a CQ that belongs to a more important class does not experience long waiting times. However neither the ML or MOL schedulers can respect the classes and avoid priority inversion.

Figure 130 shows the performance of Workload 5L under the MPRR scheduler. This scheduler increases the response times of classes 4 and 5 close to 120 msecs. Initially, it has no priority inversion but as the priorities change it does not detect these changes. This results in priority

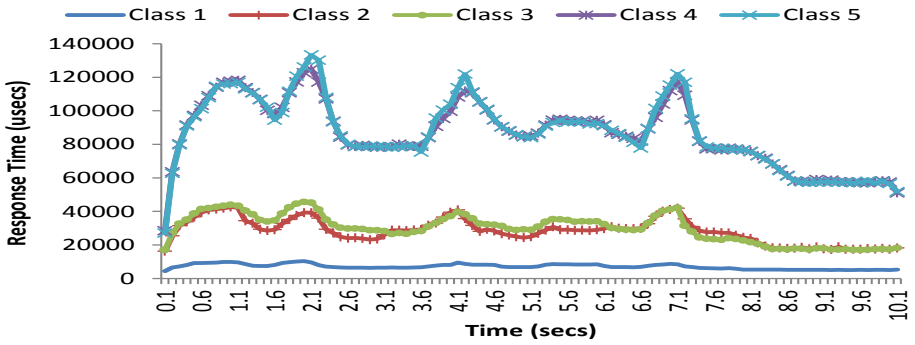

Figure 130: PIAT Experiment: Under MPRR, Class 1 has the lowest response time. Classes 4 and 5 have the highest.

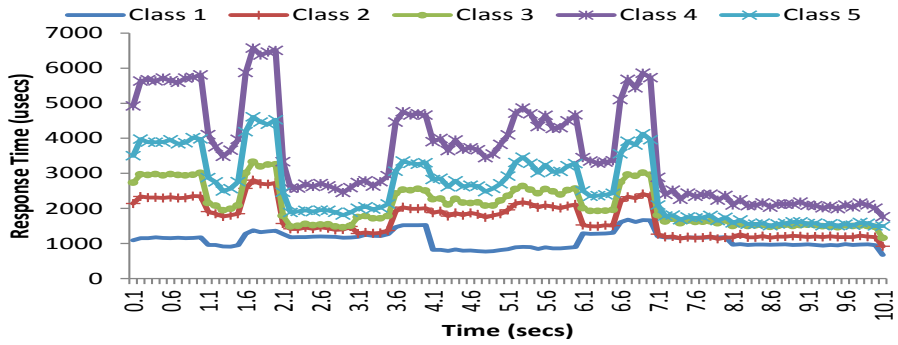

Figure 131: PIAT Experiment: MBD does not detect the changes in priority. Class 4 and 5 stay the same priority. Class 4 response time is higher than Class 5.

inversion after 4.5 seconds when the priorities change as shown in Figure 135. In addition, the response time of all of the classes is much higher than that under MBD-based schedulers. This result is consistent with the performance of the MPRR scheduler in Section 6.3.1.

Figure 131 shows the performance of the MBD scheduler when none of the modules are enabled. Thus, it cannot detect the changes in priority or selectivity, or any priority inversion. Classes 4 and 5 are treated as equals throughout the run of the experiment. This results in priority inversion as shown in Figure 135 because Class 4 is heavier than Class 5 taking up more CPU time and

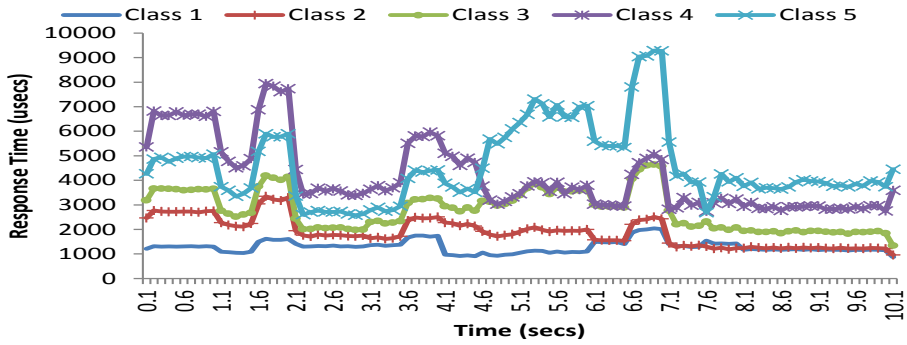

Figure 132: PIAT Experiment: DVP reacts to the changes in the workload and to the changes in priority. At 4.5 seconds, Class 4's response time becomes lower than Class 5's due to the change in priority.

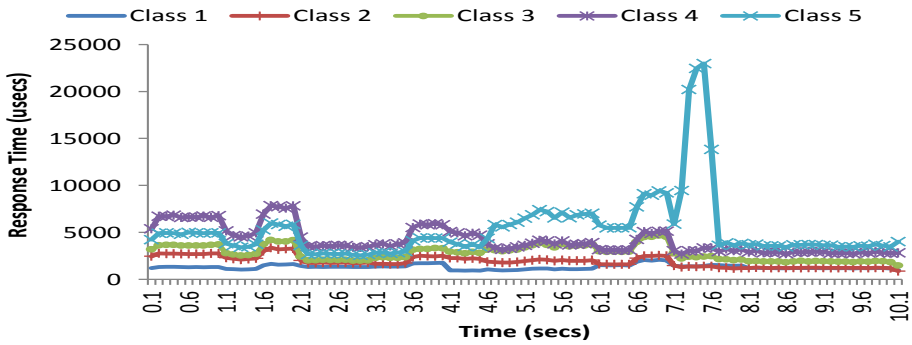

Figure 133: PIAT Experiment: DSP reacts to the changes in the workload and to the changes in priority. At 4.5 seconds, Class 4's response time becomes lower than Class 5's due to the change in priority.

forcing longer waits on Class 5.

Figure 132 shows the response times of the classes under the MBD scheduler once the priority detection module _UP and the DVP_SA module are enabled. The scheduler detects the changes in priority and adapts better to the changes in the input rate and selectivities as shown in Figures 134 and 135.

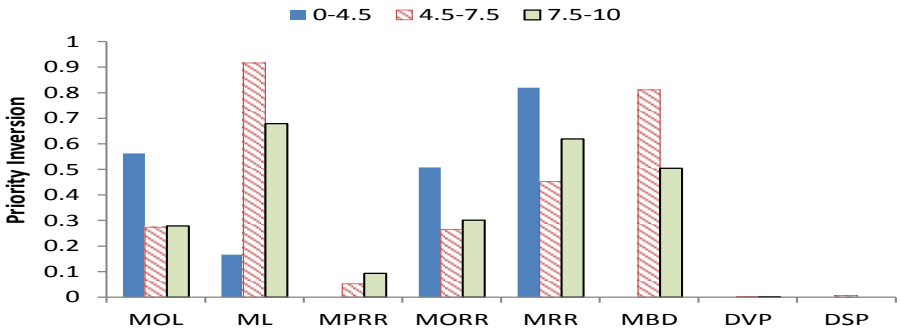

Figure 134: PIAT Experiment: DSP and DVP have the lowest priority inversion for this workload.

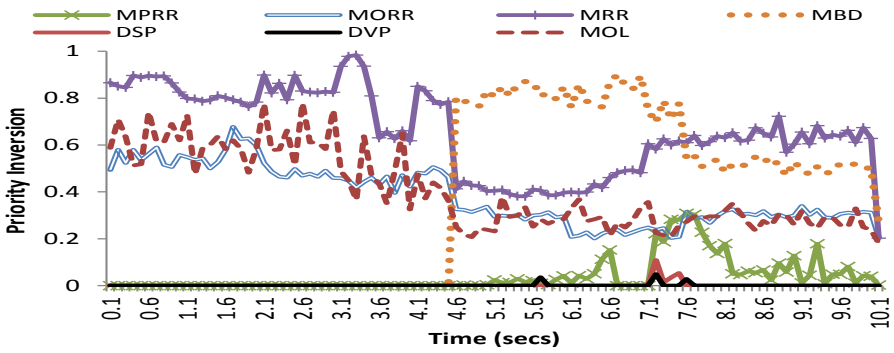

Figure 135: PIAT Experiment: DSP and DVP detect and correct priority inversion when it occurs.

Figure 133 shows the performance of the MBD scheduler once the priority detection module _UP and the DSP_SA module are enabled. The scheduler again adapts well to the changes in priority, input rates and selectivity. It has no priority inversion except for the period between 7.2 and 7.6 as shown in Figure 135. However, it quickly adapts and removes the priority inversion. All in all, the MBD_DSP_SA_UP scheduler or the MBD_DVP_SA_UP scheduler performs the best for Workload 5L in that both adapt the fastest to the changes in the workload while adhering to the goals of preventing priority inversion and starvation, utilizing the system resources efficiently and respecting the priorities of the classes.

6.3.8 MBD scheduling summary

In this chapter, we proposed a new scheduler called the Multicore Broadcast Disks (MBD) scheduler for DSMSs running in a multi-core environment. We evaluated the MBD scheduler's scalability, adaptivity to the selectivity and priority changes during runtime and priority inversion detection and correction. We compared the performance of the MBD scheduler to that of many baseline schedulers. Our results show that the MBD scheduler performs very well, i.e., respects priorities of the classes, prevents starvation and priority inversion, adapts to the changes in the workload and scales well to utilize all the operational cores allocated to the DSMS.

7.0 CONCLUSION AND FUTURE WORK

In this dissertation, we propose new scheduling strategies for a class-based DSMS running in a single-, dual-, and multi-core environment. The underlying approach is a two-level scheduling framework that allows multiple classes to execute while honoring their priorities.

In a single-core environment, our proposed Continuous Query Class (CQC) scheduler, divides the CPU time among the classes according to their priorities. This guarantees a lower response time for higher priority classes without starving lower priority classes.

In a dual-core environment, our proposed Adaptive Broadcast Disk (ABD) scheduler utilizes a schedule that guarantees a portion of CPU time proportional to each class priority and a low waiting time for the classes in the system. It successfully adapts to changes in the load during runtime including priority inversion that is detected and corrected by its priority inversion module.

In a multi-core environment, our proposed Multi-core Broadcast Disk (MBD) scheduler increases parallelism by considering the CQs as the minimum schedulable unit instead of an entire class of CQs. It divides the CPU time assigned to a class among its CQs according to each CQ's expected cost relative to the expected cost of the class. Based on this division, it generates a new schedule which maintains class semantics. During runtime, when a core becomes available, the scheduler consults the schedule to determine the next CQ to execute on that core. Extra care is taken so that a CQ does not run on two cores at the same time. To prevent priority inversion, the MBD scheduler has a priority inversion module that detects and corrects priority inversion dynamically during runtime. It also adapts to any changes in the selectivities of the CQs and in the priorities of the classes.

We evaluated our schedulers through a real system implementation in AQSIOS (single-core) and through simulation under Sim-AQSIOS (dual- and multi- core). Our experiments show that the proposed schedulers satisfy the goals of this thesis, namely optimizing the performance of critical

CQs while utilizing the system resources efficiently, preventing priority inversion and starvation, and adapting to the characteristics of the workload dynamically during runtime.

The design of each scheduler is carefully tailored to its running environment. The CQC scheduler is designed to schedule source, operational and output operators on a single core. Its low overhead and minimal switching between the classes makes it ideal for single-core environments. The ABD and the MBD schedulers have more context switches and execution overhead making them not as efficient in such an environment.

The ABD scheduler, in a dual core setting, adapts to its environment and is designed to shorten the wait time for the tuples in the system. The dynamic quota slice detection is a crucial part of this scheduler because it allocates to the classes enough time to execute (no starvation) without adding significant wait time on the other classes. The CQC scheduler increases the wait time in such an environment and thus is not suitable for dual core environments. The MBD scheduler also cannot perform as well as the ABD scheduler in a dual core environments. The ABD and the MBD schedulers are inherently different. The ABD scheduler is a class-level scheduler and treats all the operators and CQs within a class in the same manner. The MBD scheduler, however, is a CQ-level scheduler which divides the priority of a class among its CQs based on the cost-per-tuple. This means that the priority of a CQ under the ABD scheduler is different than that under the MBD scheduler. In addition, the MBD scheduler does not have the dynamic quota slice module that the ABD scheduler has. The reason is that under ABD scheduling, the CQs execute in a single-core fashion with the sources on another core. Thus, a CQ with a long input queue would have a negative effect on the overall response time of the system. Under the MBD scheduler, with multiple cores, the other CQs and the overall system performance are not affected as much because there is at least one other core that CQs can execute on. Moreover, the overheads of maintaining the MBD scheduler are also higher than that of the ABD scheduler: frequent scheduling points, selectivity monitoring, and more frequent schedule rebuilds. The ABD scheduler is not affected by changes in selectivities because it is a class-level scheduler. However, a change in the selectivity could cause priority inversion under MBD and an MBD schedule rebuild. These overheads are needed for the MBD scheduler, but are not significant in the larger scheme of things because the system has more resources. However, if run on a single operational core, they would become more significant.

In a multi-core environment, the MBD scheduler is designed to take advantage of having multiple scheduling points. Also by being a CQ-level scheduler, it can parallelize the execution of the CQs on multiple cores while maintaining their class semantics. Neither the CQC nor the ABD scheduler can utilize multiple cores. Being class-level schedulers, the minimum schedulable unit is the whole class which is then assigned to a level 2 scheduler. Thus, only a class, not a CQ can be scheduled on a single core. When the number of classes is less than the number of cores (which is often the case), one or more cores would be underutilized while others are overloaded. Ensuring that this does not happen requires the load manager to intervene and divide the CQs into new classes while maintaining the original class semantics, balancing the CQs on these cores, and shedding tuples when needed. Designing such a manager is out of the scope of this thesis.

7.1 SUMMARY OF CONTRIBUTIONS

In short, the contributions of this thesis are:

- We developed a new, two-level scheduling framework for DSMSs where level 1 manages the overall schedule for the CQ classes and the level 2 schedules the operators that constitute the CQs (CQC and ABD scheduling) or the CQs that constitute the classes (MBD scheduling).
- We designed and implemented a scheduler namely the *CQC scheduler* for single core environments that honors the priorities of the continuous queries belonging to different classes.
- We designed and implemented a class-aware scheduler namely the *ABD scheduler* for dual-core environments that prevents priority inversion and reduces the wait time of the continuous queries.
- We designed and implemented a class-aware scheduler namely the *MBD scheduler* for multi-core environments that is scalable to a large number of cores, prevents priority inversion, and optimizes the weighted response time.
- We introduced the concept of priority inversion to class-based DSMSs and proposed a way to quantify it, namely the priority inversion ratio.
- We introduced the weighted percentile response time metric to evaluate how the schedulers respect the priorities of the CQ classes.

7.2 BROAD IMPACT

The schedulers proposed in this thesis allow monitoring applications to specify the class that a CQ belongs to. Thus, more important CQs receive more CPU time and better response time under our system than in a traditional DSMSs. This allows the monitoring applications to detect important events faster while still detecting less important events.

Our schedulers are not restricted to DSMSs, rather, they could be utilized in any continuous environment where the size of the continuous tasks varies from one period to another. In addition, our scheduling framework can be used to optimize the performance of the system when other metrics are considered and/or other resources are managed.

7.3 FUTURE WORK

As an extension of the work in this thesis and to make the other system components aware of the priority classes, we propose the following updated modules for DSMSs:

- **Affinity-based scheduler:** As part of this thesis, we explored the impact of affinity on the response time in a multi-core environment. For the types of CQs used in the experimental sections when run on AQSIOS, the miss rate was 0.1%. Thus we designed another experiment that has 100 aggregate CQs where each 10 CQs share a selection operator. The window for the aggregations is 1 million time units with an input rate of 1500 tuples/sec. Thus, the number of tuples in the aggregate windows is between 750-1424 tuples depending on the selectivity of the selection operator. We use a Dell Inspiron E1405 laptop with Intel Core Duo processor T5500 @ 1.66 GHz with 1 GB of RAM running Fedora 10 with the 2.6.27 Linux kernel. AQSIOS executes on only one core of the CPU. We measure response times using the PROCESS_CPUTIME clock, which measures only the process active execution time. With such a large amount of windows, we wanted to flood the cache so that executing an operator would result in cache misses for its state. However, the result was a mere 0.04% miss rate for the L1 cache if batch processing was used and a 0.6% miss rate for the L1 cache if one tuple at a time is processed. Out of those misses that went to the L2 cache, the miss rate was 7.9%

when batch processing was used and 2.3% when one tuple was executed at a time. This means that the number of overall cache misses is very low in such an architecture. In case the DSMS architecture changes in a way that the cache misses are higher or when the operator implementation favors cache affinity, we designed a multi-core scheduler inspired by the MBD scheduler that is affinity-aware. The Affinity-aware Multicore Broadcast Disk (AMBD) scheduler used the same MBD schedule for the CQs with the same modules. However, at each scheduling point when a core becomes idle, it does not retrieve the next CQ_i from the schedule. Instead, it slides down the schedule by ten CQs[1]. If it finds a CQ_j that was previously executed on this core, then it executes CQ_j. Otherwise, it executes CQ_i. The AMBD scheduler when run on SimAQSIOS without any caching effect results in ten percent slower average response time than the MBD scheduler, indicating that the overhead is more than the benefit. However, we foresee that in the future if operator implementations and DSMS architectures take advantage of cache affinity, then using AMBD would result in better response times.

- **Class-aware query optimizer:** The query optimizer should consider the classes of the continuous queries while making decisions on sharing operators, input and output queues and operator implementation. This query optimizer would also work together with the scheduler during runtime to devise query plans for newly arriving continuous queries.
- **Class-aware dissemination module:** The dissemination module sends the results of a continuous query to the user over the network. Thus, it determines how fast the user receives those results. A class-aware dissemination module would consider the classes and the priorities of the continuous queries to make sure important results do not wait unnecessarily.
- **Intra-class QoS scheduler:** Continuous queries within a class have the same priority within the system, but some of them might have other Quality of Service (QoS) requirements such as deadlines. This requires a new scheduler that can meet these requirements while still maintaining priority class semantics.

[1]The number of lookahead entries used (10) was chosen arbitrary, but it corresponds to 15% of the total number of CQs in workloads used.

BIBLIOGRAPHY

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: data management for asymmetric communication environments. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, pages 199–210, New York, NY, USA, 1995. ACM.

[3] R. Acker, C. Roth, and R. Bayer. Parallel query processing in databases on multicore architectures. In *Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, 2008.

[4] L. Al Moakar, A. Labrinidis, and P. K. Chrysanthis. Adaptive class-based scheduling of continuous queries. In *Proc. of the Seventh International Workshop on Self-Managing Database Systems*, 2012.

[5] L. Al Moakar, T. N. Pham, P. Neophytou, P. K. Chrysanthis, A. Labrinidis, and M. A. Sharaf. Class-based continuous query scheduling for data streams. In *Proc. of 6th International Workshop on Data Management for Sensor Networks*, 2009.

[6] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[7] R. Avnur and J. M. Hellerstein. Continuous query optimization. Technical Report UCB/CSD-99-1078, EECS Department, University of California, Berkeley, 1999.

[8] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, 2003.

[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):846–860, 2004.

[10] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, 2013.

[11] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 345–354, New York, NY, USA, 1993. ACM.

[12] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng. Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *USENIX Annual Technical Conference, General Track*, pages 337–352, 2005.

[13] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, 2003.

[14] S. Chakravarthy and Q. Jiang. *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Springer Publishing Company, Incorporated, 2009.

[15] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: a proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

[16] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[17] L. Chen, M. Mizuno, and G. Singh. A priority inheritance-based inversion control methodology for general resource access problems. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 202–210, May.

[18] http://www.coral8.com/, 2004.

[19] S. Das, S. Antony, D. Agrawal, and A. El Abbadi. Thread cooperation in multicore architectures for frequency counting over multiple data streams. *Proc. VLDB Endow.*, 2009.

[20] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 261–276, New York, NY, USA, 1999. ACM.

[21] http://esper.codehaus.org, 2006.

[22] B. Gedik, P. S. Yu, and R. R. Bordawekar. Executing stream joins on the cell processor. In *Proceedings of the 33rd international conference on Very large data bases*, 2007.

[23] V. Gil-Costa, R. Barrientos, M. Marin, and C. Bonacic. Scheduling metric-space queries processing on multi-core processors. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010.

[24] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Adaptive scheduling of web transactions. In *ICDE*, pages 357–368, 2009.

[25] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 297–308. VLDB Endowment, 2003.

[26] D. Han, G. Wang, H. Liu, B. Xu, and Y. Fang. An effective scheduling algorithm based-on qos adaptation framework over data streams. In *Information and Automation, 2008. ICIA 2008. International Conference on*, pages 1232 –1235, june 2008.

[27] W.-S. Han and J. Lee. Dependency-aware reordering for parallelizing query optimization in multi-core cpus. In *Proceedings of the 35th SIGMOD international conference on Management of data*, 2009.

[28] D. W. Holmes, J. R. Williams, and P. Tilke. An events based algorithm for distributing concurrent tasks on multi-core architectures. *Computer Physics Communications*, 181(2):341–354, 2010.

[29] R. Johnson, S. Harizopoulos, N. Hardavellas, K. Sabirli, I. Pandis, A. Ailamaki, N. G. Mancheril, and B. Falsafi. To share or not to share? In *Proceedings of the 33rd international conference on Very large data bases*, 2007.

[30] D. Kulkarni, C. V. Ravishankar, and M. Cherniack. Real-time, load-adaptive processing of continuous queries over data streams. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 277–288, New York, NY, USA, 2008. ACM.

[31] A. Labrinidis, H. Qu, and J. Xu. Quality contracts for real-time enterprises. In *Proceedings of the 1st international conference on Business intelligence for the real-time enterprises*, BIRTE'06, pages 143–156, Berlin, Heidelberg, 2007. Springer-Verlag.

[32] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 65–74, New York, NY, USA, 2009. ACM.

[33] X. Li, Z. Jia, L. Ma, Z. Qin, and H. Wang. Qos-aware scheduling for mixed real-time queries over data streams. In *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '09, pages 145–154, Washington, DC, USA, 2009. IEEE Computer Society.

[34] X. Li, Z. Jia, L. Ma, R. Zhang, and H. Wang. Earliest deadline scheduling for continuous queries over data streams. In *Proceedings of the 2009 International Conference on Embedded Software and Systems*, ICESS '09, pages 57–64, Washington, DC, USA, 2009. IEEE Computer Society.

[35] Microsoft. Priority inversion (windows). http://msdn.microsoft.com/en-us/library/windows-/desktop/ms684831(v=vs.85).aspx.

[36] http://www.microsoft.com/sqlserver/2008/en/us/r2-complex-event.aspx, 2008.

[37] L. A. Moakar, P. K. Chrysanthis, C. Chung, S. Guirguis, A. Labrinidis, P. Neophytou, and K. Pruhs. Admission control mechanisms for continuous queries in the cloud. In *ICDE*, 2010.

[38] J. Nagle. On packet switches with infinite storage, 1985.

[39] F. Narayanan, S.; Waas. Dynamic prioritization of database queries. In *ICDE*, 2011.

[40] T. N. Pham, L. A. Moakar, P. K. Chrysanthis, and A. Labrinidis. Dilos: A dynamic integrated load manager and scheduler for continuous queries. In *The Sixth International Workshop on Self-Managing Database Systems*, 2011.

[41] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. In *Proc. VLDB Endow.*, 2008.

[42] H. Qu and A. Labrinidis. Preference-aware query and update scheduling in web-databases. In *ICDE*, 2007.

[43] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 259–269, Dec.

[44] S. Schneidert, H. Andrade, B. Gedik, K.-L. Wu, and D. S. Nikolopoulos. Evaluation of streaming aggregation on parallel hardware architectures. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, 2010.

[45] B. Schroeder, M. Harchol-Balter, A. Iyengar, and E. Nahum. Achieving class-based qos for transactional workloads. In *ICDE*, 2006.

[46] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175–1185, Sep.

[47] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Efficient scheduling of heterogeneous continuous queries. In *VLDB*, 2006.

[48] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Transactions on Database Systems*, 2008.

[49] M. A. Sharaf, S. Guirguis, A. Labrinidis, K. Pruhs, and P. K. Chrysanthis. Asets: A self-managing transaction scheduler. *Proc. of 3rd International Workshop on Self-Managing Database Systems*, pages pp. 56–62, April 2008.

[50] Simpy simulation package. http://simpy.sourceforge.net.

[51] M. Squadrito, L. Esibov, L. C. DiPippo, V. F. Wolfe, G. Cooper, B. Thurasignham, P. Krupp, M. Milligan, and R. Johnston. Concurrency control in real-time object-oriented systems: the affected set priority ceiling protocols. In *Object-Oriented Real-time Distributed Computing, 1998.(ISORC 98) Proceedings. 1998 First International Symposium on*, pages 96–105. IEEE, 1998.

[52] http://www.streambase.com, 2006.

[53] G. Sun, Y. Zhou, Y. Huang, and Y. Zhou. Adaptive scheduling strategy for data stream management system. In *Proceedings of the joint 9th Asia-Pacific web and 8th international conference on web-age information management conference on Advances in data and web management*, APWeb/WAIM'07, pages 511–521, Berlin, Heidelberg, 2007. Springer-Verlag.

[54] http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.index.html, 2008.

[55] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[56] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[57] C.-D. Wang, H. Takada, and K. Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. In *Parallel Architectures, Algorithms, and Networks, 1996. Proceedings., Second International Symposium on*, pages 70–76, Jun.

[58] Y. Wang, W. Xuan, W. Li, B. Song, and X. Li. A real-time scheduling strategy based on priority in data stream system. In *HIS '09: Proceedings of the 2009 Ninth International Conference on Hybrid Intelligent Systems*, 2009.

[59] Y. Wei, V. Prasad, S. H. Son, and J. A. Stankovic. Prediction-based qos management for real-time data streams. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 344–358, Washington, DC, USA, 2006. IEEE Computer Society.

[60] Y. Wei, S. H. Son, and J. A. Stankovic. Rtstream: Real-time query processing for data streams. In *ISORC*, 2006.

[61] J. Wu, K.-L. Tan, and Y. Zhou. Qos-oriented multi-query scheduling over data streams. In *Proceedings of the 14th International Conference on Database Systems for Advanced Applications*, DASFAA '09, pages 215–229, Berlin, Heidelberg, 2009. Springer-Verlag.

[62] Zigbee specification 053474r06, version 1.0, 2004.