Performance Evaluation of Scheduling Algorithms in Aurora

Jeff Chen

Department of Computer Science
Brown University

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

May 15, 2002

Stanley B. Zdonik

Professor Stan Zdonik
Advisor

Table of Contents

Performance Evaluation of Scheduling Algorithms in Aurora

Jeff Chen
Department of Computer Science
Brown University
Providence, RI. 02912-1910

Abstract

This report describes the implementation of an Aurora scheduler simulator and the performance evaluation of several scheduling algorithms. The simulator is a close representation of the prototype Aurora. It is capable of simulating various kinds of networks, as well as various kinds of scheduling decisions. The simulator also has output monitors which calculate the latency and utility for a variety of experiments and simulations.

1 Introduction

1.1 Motivation

The emergence of sensors and similar small-scale embedded computing devices that continuously produce large volumes of data they obtain from their environment has emphasized the importance of continuous query processing systems. Millions of questions from millions of sources are to be answered. The time to answer is limited yet the continuous data are coming in infinitely.

Existing database management systems are ill-equipped for supporting real-time data streams and continuous queries because they are designed on the assumption that the system is working on a passive data repository storing a large yet finite collection of data. They process only human-initiated queries.

In the pervasive computing environment, however, data is no longer passively stored; queries are no longer passively asked. On the contrary, data are coming in as streams, and queries are asked continuously. Every mistake (or bad decision) might be unacceptable and every delay might be catastrophic. The challenges of pervasive computing consist of time limitation, scalability, unreliable sources, and human demanded quality of service.

1.4 Simulation Environment

The simulator is developed with C++ and CSIM18 under UNIX (Solaris SunOS 5.7). The simulations are run on Sun Ultra 10 machines, with 440 MHz CPU and 256MB RAM.

CSIM is a library of routines, it is a process-oriented, general purpose simulation toolkit written with general C language functions. It allows creation and implementation of process-oriented, discrete-event simulation models which can simulate complex systems and offer insight into the system's dynamic behavior

In this simulation, we will use CSIM to report the state of each process. Each stream generator, each worker thread, the scheduler, the output monitor, and the storage manager are all CSIM processes. The use of CSIM is very useful for debugging and understanding how the network and scheduling works. We will also use CSIM's internal clock and facility utilization to evaluate the performance of scheduling algorithms.

1.5 Scheduling Algorithms

The scheduling algorithms are gathered from variety of fields. We borrowed the LWF (longest wait first) from the network routing, EDF (earliest deadline first) from multimedia scheduling, and IXW (input × wait time) from broadcasting. Of course, the collection includes the simple ones such as RANDOM and FIFO (first-in-first-out) as well.

This report will evaluate the algorithms and see if these proven optimized algorithms in other fields will work well under the Aurora environment and see if there is room for improvement.

There are also other modifications and variations of algorithms. A detail summary about all the algorithms tested can be fund in section 2.1.

1.6 Organization

This report is organized as follows. Section 2 discusses the Aurora architecture. Section 3 describes the simulator components, and the flowcharts of its logics. Section 4 describes the experimental frame work, with results and observation from each simulation. Section 5 provides a detail analysis of the behavior from the experiments. Section 6 concludes the report. Section 7 describes future work.

3 The Simulation Model

This section describes the individual components of the scheduler simulator.

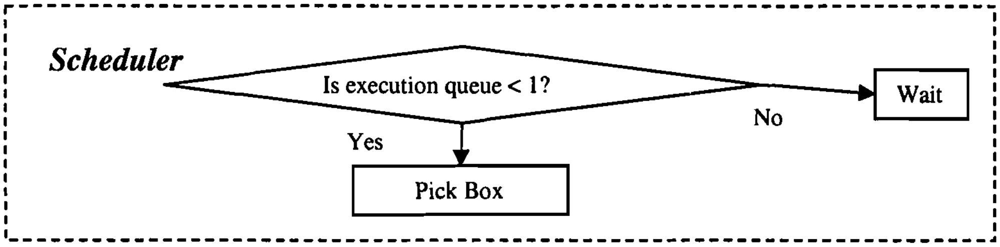
3.1 Main

- Setup parameters.
- Create and initialize query network.
- Start simulation.

3.2 Simulator

- Call Scheduler.
- Call Stream generator.
- Call Storage Manager.
- Call Worker Thread.
- Wait for the simulation to be finished.
- Print report.

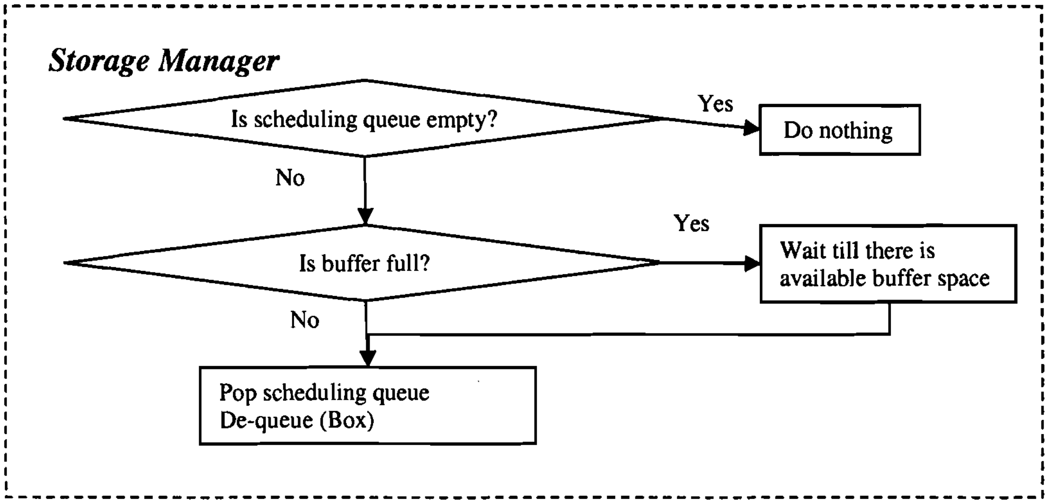
3.3 Scheduler



3.4 Stream Generator

- Generate Stream.
- Call Router.

3.5 Storage Manager



3.9 De-queue Box

- Bring desired number of messages to buffer (use time).
- Update buffer state.
- Set Box executable.

3.10 Router

- Pass message to current box's next arc(s).
- Check if next box is schedulable.

4.2 The Optimal Inter-arrival Time

4.2.1 Purpose:

This experiment determines the optimal data inter-arrival time (the rate in which messages are generated) where the network can handle.

4.2.2 Setup:

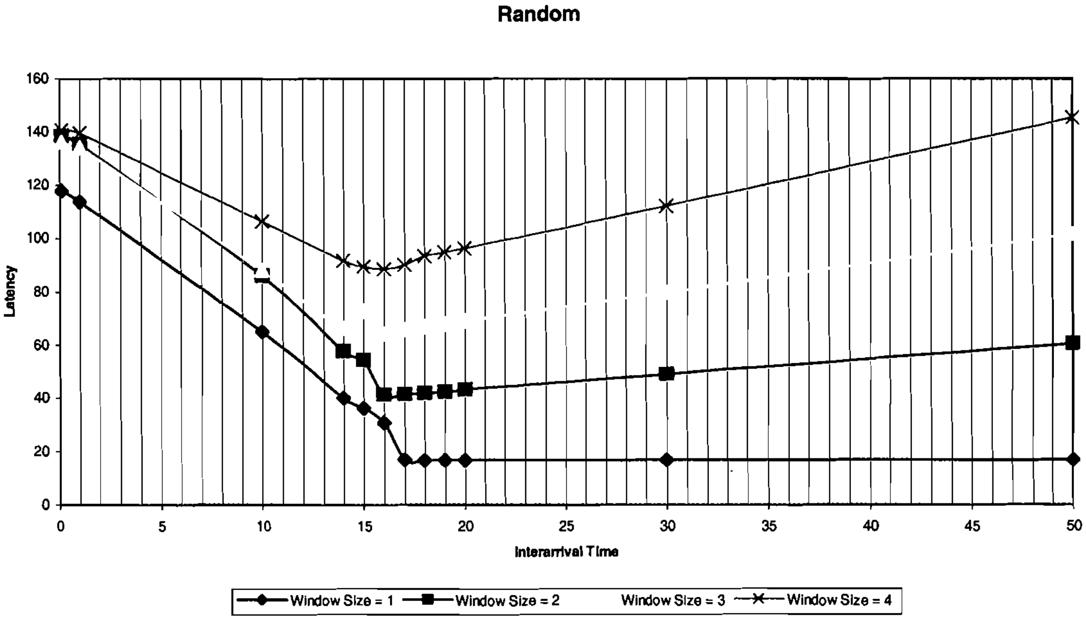
This experiment measures the latency, which is the duration that each message lasts from its time of creation till reaching the application. (In other word, the time a message stayed in the network.)

The experiment is run with variable window sizes. The network size is 4 by 4, and the numbers of elements are 10. It's assumed that there is only one worker thread, and the buffer is unlimited. Storage manager is not a factor in this experiment.

The Average Latency is calculated by taking the average of the difference between messages' generated time and the time messages' reached the application. The results are plotted with inter-arrival time from 0 to 50 respectively.

4.2.3 Results:

Two Algorithms are used. One is random, where scheduler randomly selects schedulable boxes to schedule. The other one is first-in-first-out (FIFO), where scheduler selects the schedulable box that contains the oldest message.



4.3 Performance Evaluation of Scheduling Algorithms

4.3.1 Purpose:

This experiment compares various algorithm performances near the optimal message inter-arrival time and determines the best scheduling algorithm.

4.3.2 Setup:

This experiment will measure the latency and the utility of each application with respect to each kind of algorithm.

The experiment is run with some fixed parameters. The network size is 4 by 4, and the numbers of elements are 300. It's assumed that there is only one worker thread, and the buffer is unlimited. Storage manager is not a factor in this experiment.

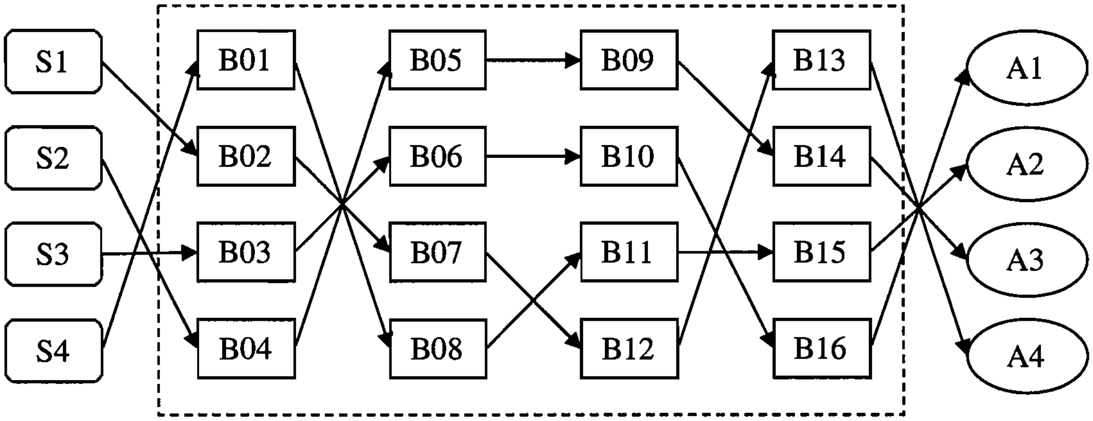


Figure 7: Query network for this experiment.

The same network will be used for all the algorithms. There are no fan-outs in this network.

The quality of service of each application:

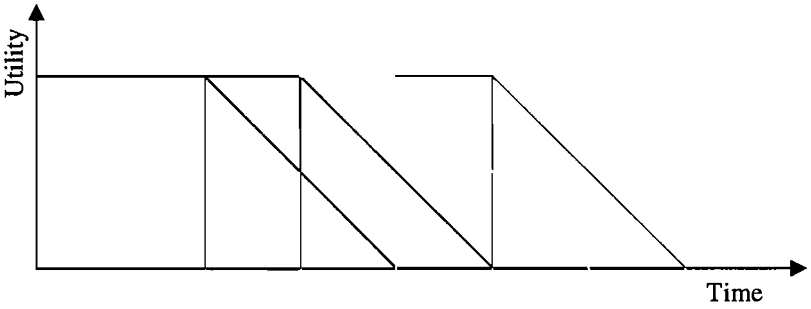


Figure 8: QOS graph for this experiment.

QOS graphs are specified by threshold and interval. The QOS for this experiment has the threshold of 100, 150, 200 and 250. The intervals are fixed at 50 apart. The utility maximum is 100%, and decreases when reaching the threshold with the slope of -1. The minimum utility is 0%, and not any further. The threshold of each QOS is also the deadline for each application.

applications that are doing extremely badly and the rest of the applications are doing acceptable well.

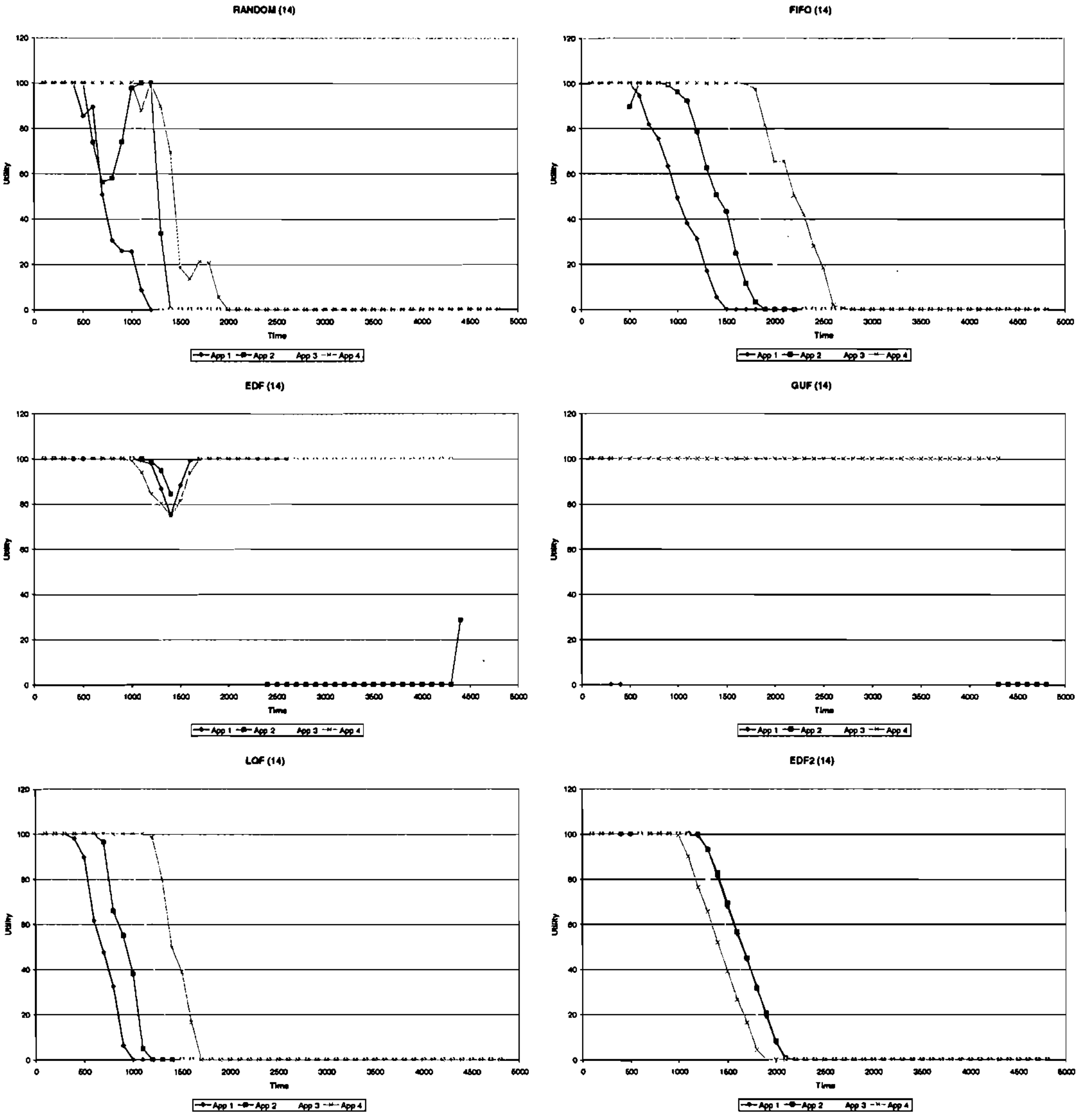


Figure 10: Utility of inter-arrival time at 14.

These graphs are the utility obtained by the latency result from previous figures. Most of the utilities drop to 0 as time goes on. The exceptions are EDF and GUF, where one application drops to zero all of a sudden, and the others are kept alive because the sacrifice of one application.

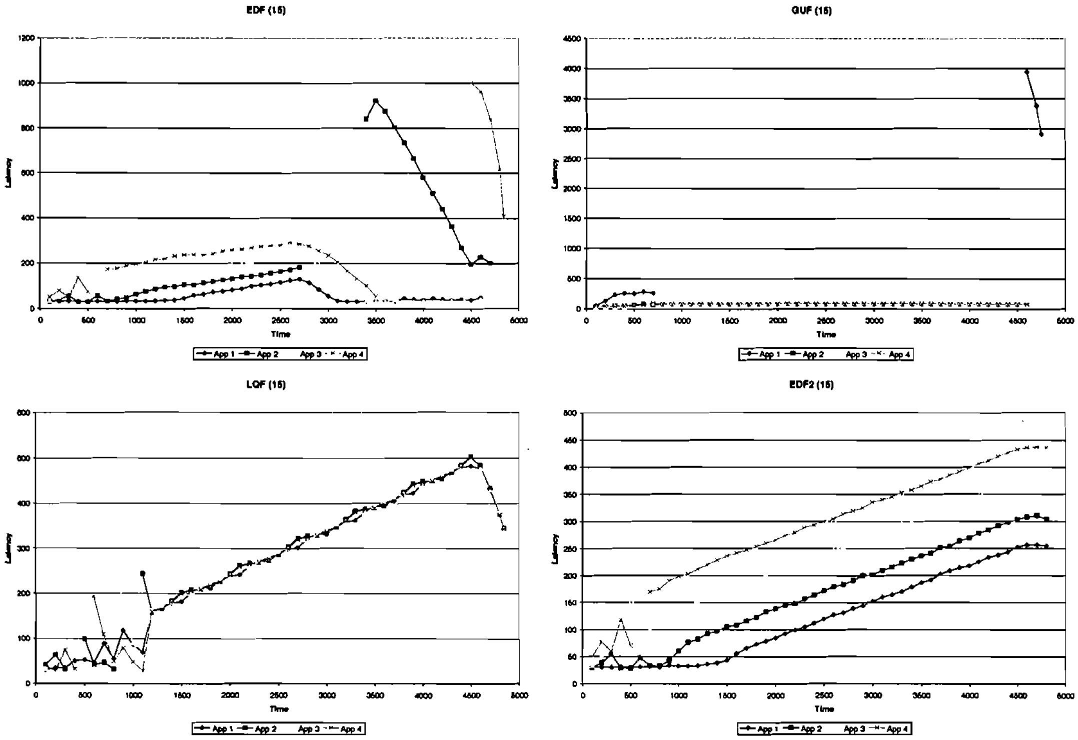
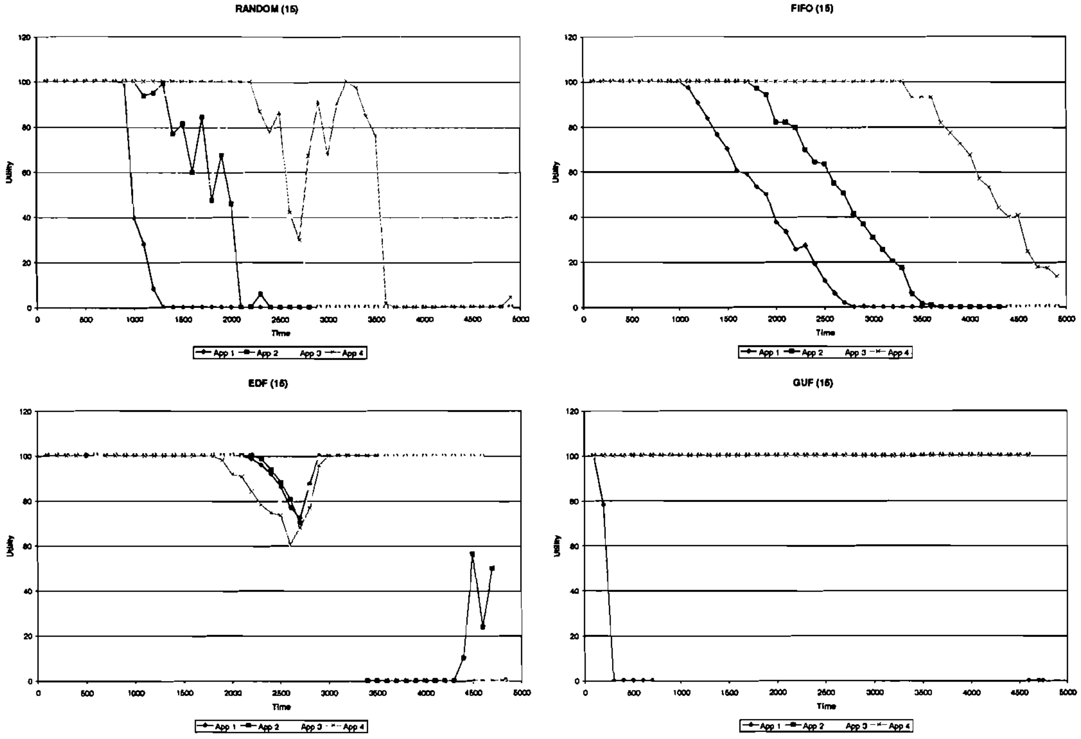


Figure 12: Latency of inter-arrival time at 15.

The graphs are similar to inter-arrival time of 14, except the latency is lower at the respective time index at 14.



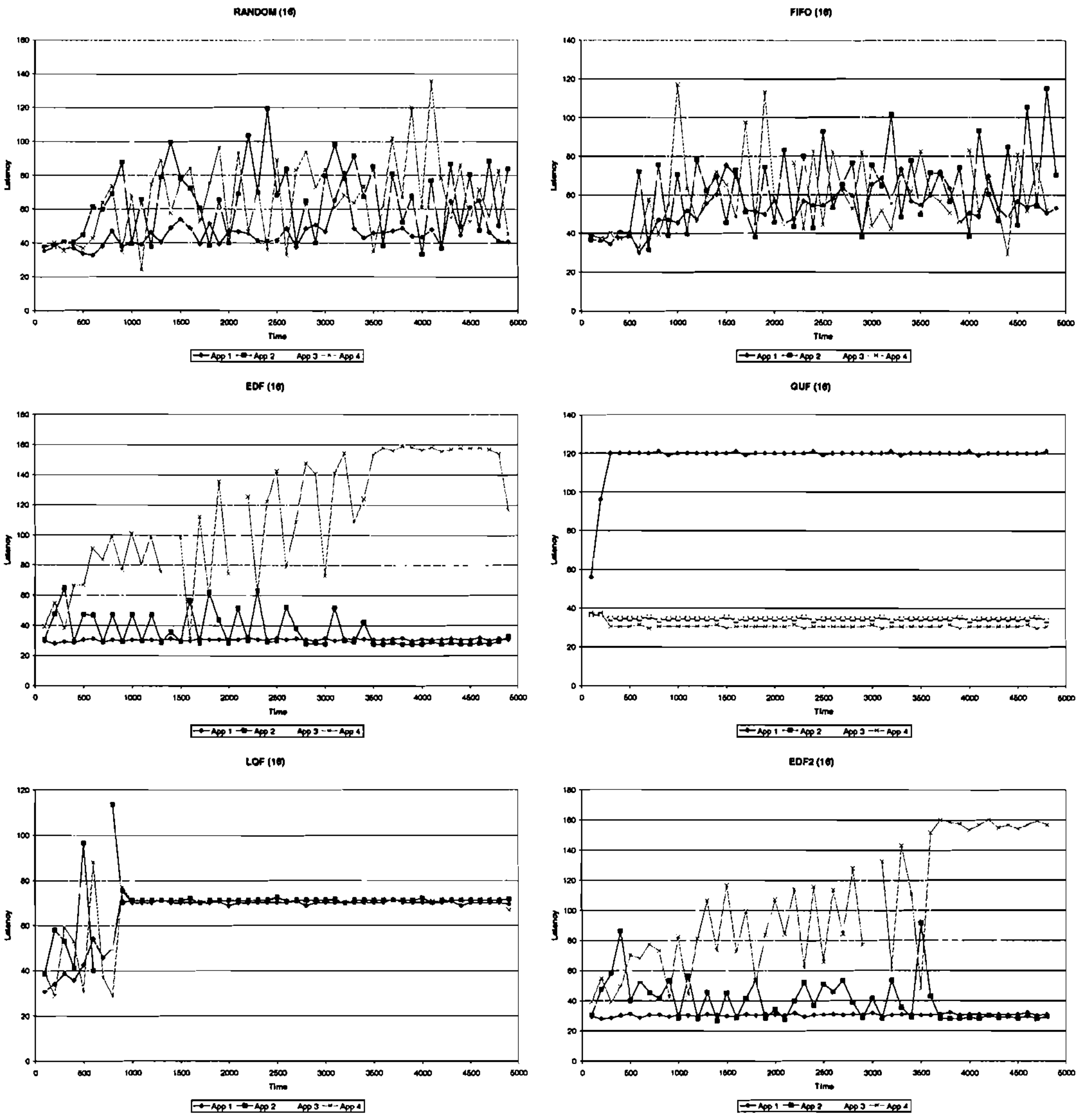


Figure 15: Latency of inter-arrival time at 16.

The latency graph seems to be fluctuation a lot. However, they are actually fluctuating in a very small scale. What's going on is that when the worker thread is doing other boxes at that interval of time, the latency increases. The latency decreases when the worker thread is working on that particular box. Some algorithm such as EDF, GUF, LQF actually converge to a steady latency when the simulation is run long enough.

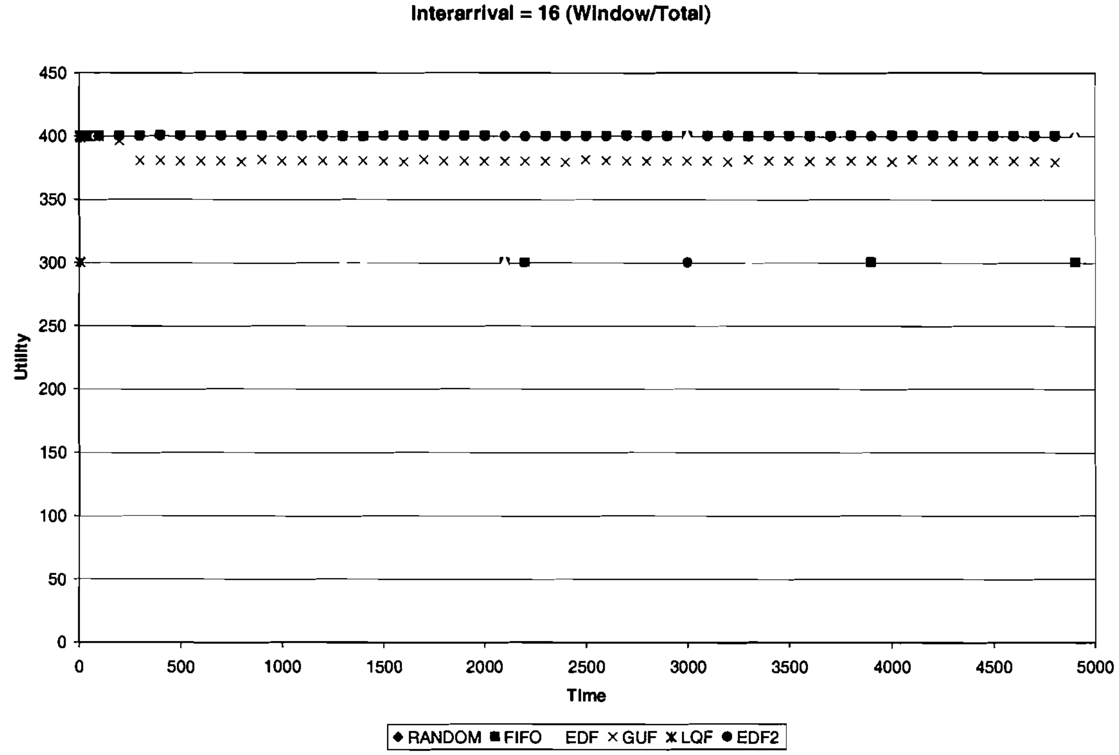


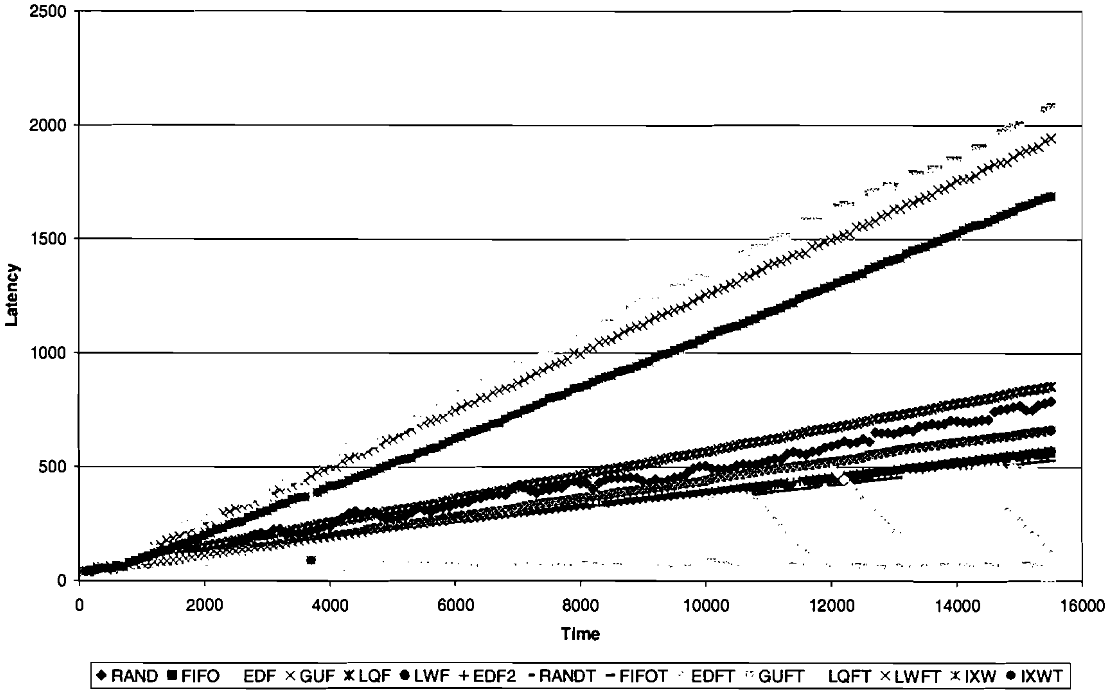
Figure 17: Total Utility of all applications with respective algorithms at 16.

Almost everyone is happy here, except the GUF, which has the problem discussed previously. Some random FIFO and EDF drop to 300. This is caused by the scheduler not scheduling one particular application at that certain time interval. In other word, no output was produced at that period, and there for it gives a false drop of the total utility.

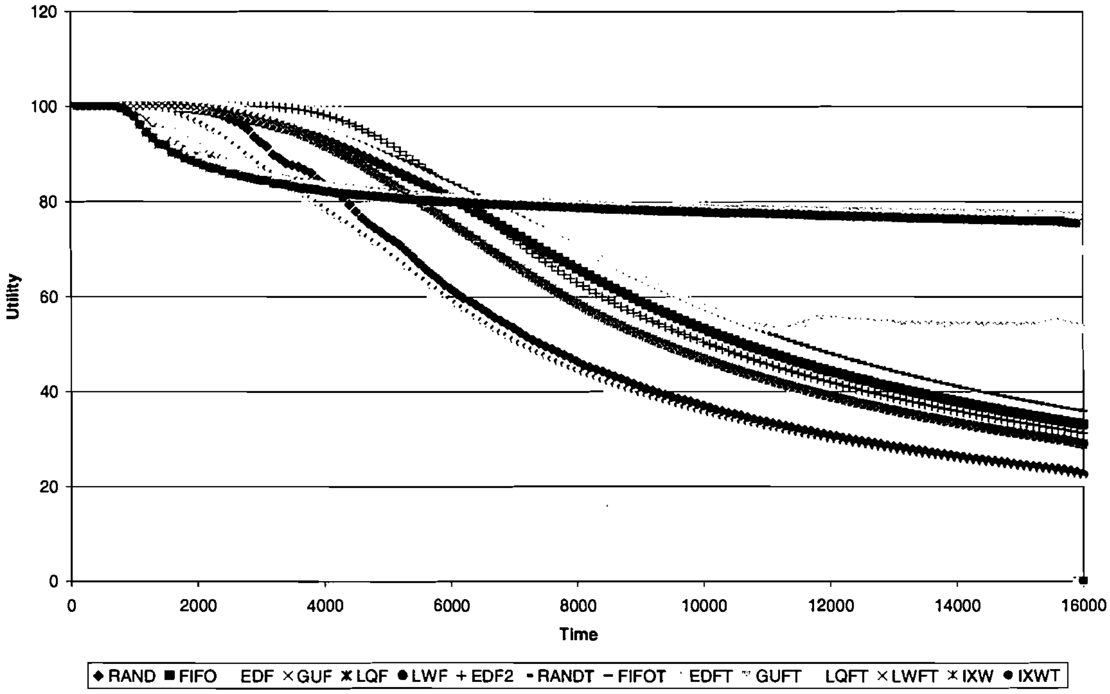
4.3.4 Observations:

1. The best performing algorithms seems to be EDF and GUF, because they are the only two that doesn't die.
2. However, how is "best performance" being defined? EDF and GUF sacrificed one application to save the other three. Is sacrificing consider a good way to schedule?
3. Once the application is sacrificed, it's not considered in future scheduling unless scheduler has nothing else more important to do (i.e. toward the end of simulation while every other application is finished)
4. In all algorithms, when 2 comparing value are equal, one is selected randomly. For example, if 2 tuples have same utility, one is selected randomly to be executed. This result a not very smart GUF algorithm where it might not pick the tuples amount the same utility ones that's closes to the deadline.
5. Perhaps a combined algorithm can be developed. But what is the priority of each.
6. What does an empty slot mean? (occurred during windowed period where no tuple was processed)

15.5 interarrival/1000msg/average latency



15.5 interarrival/1000msg/cumulative/average utility



4.5 Load Shedder

4.5.1 Purpose:

This experiment compares result with and without the load shedder.

4.5.2 Setup:

A simple load shedder is implemented to help the scheduler handle the network while being overloaded. The goal was to shed some of the message tuples that has no utility (garbage) which would cause blocking of the subsequent message if they were kept in the message queue.

The load shedder is called every time scheduler is about to schedule a box. The load shedder looks through the network and finds the *schedulable* boxes. Within these boxes, the load shedder removes the messages that has 0 utility and re-verify if the box is schedulable after removing those messages. The total number of messages is recorded and output at the end of simulation

The experiment is run with similar fixed parameters. The network size is 4 by 4. It's assumed that there is only one worker thread, and the buffer is unlimited. There are no fan-outs in this network. Storage manager is not a factor in this experiment. There are 1000 messages.

4.5.3 Results:

The graph is plotted from the beginning of the simulation till all 1000 messages are processed. Inter-arrival time is 15.5. A "message shredded" graph is plotted as well.

4.5.4 Observations:

1. Killing the 1st message will cause the 2nd message not able to be executed and will eventually get killed as well if no other messages arrive before its utility reaches 0.
2. GUF(s) and IXW(s) are the ones that play with utility the most, and thus causing the most messages to be shredded.
3. Again, what is the definition of "best performer?" Here an additional variable, message shed, has added yet another extra level of complexity.
4. When messages are not coming fast enough, most of the messages that can't make it will be killed. This results very good performance (because a lot of messages are dropped.)

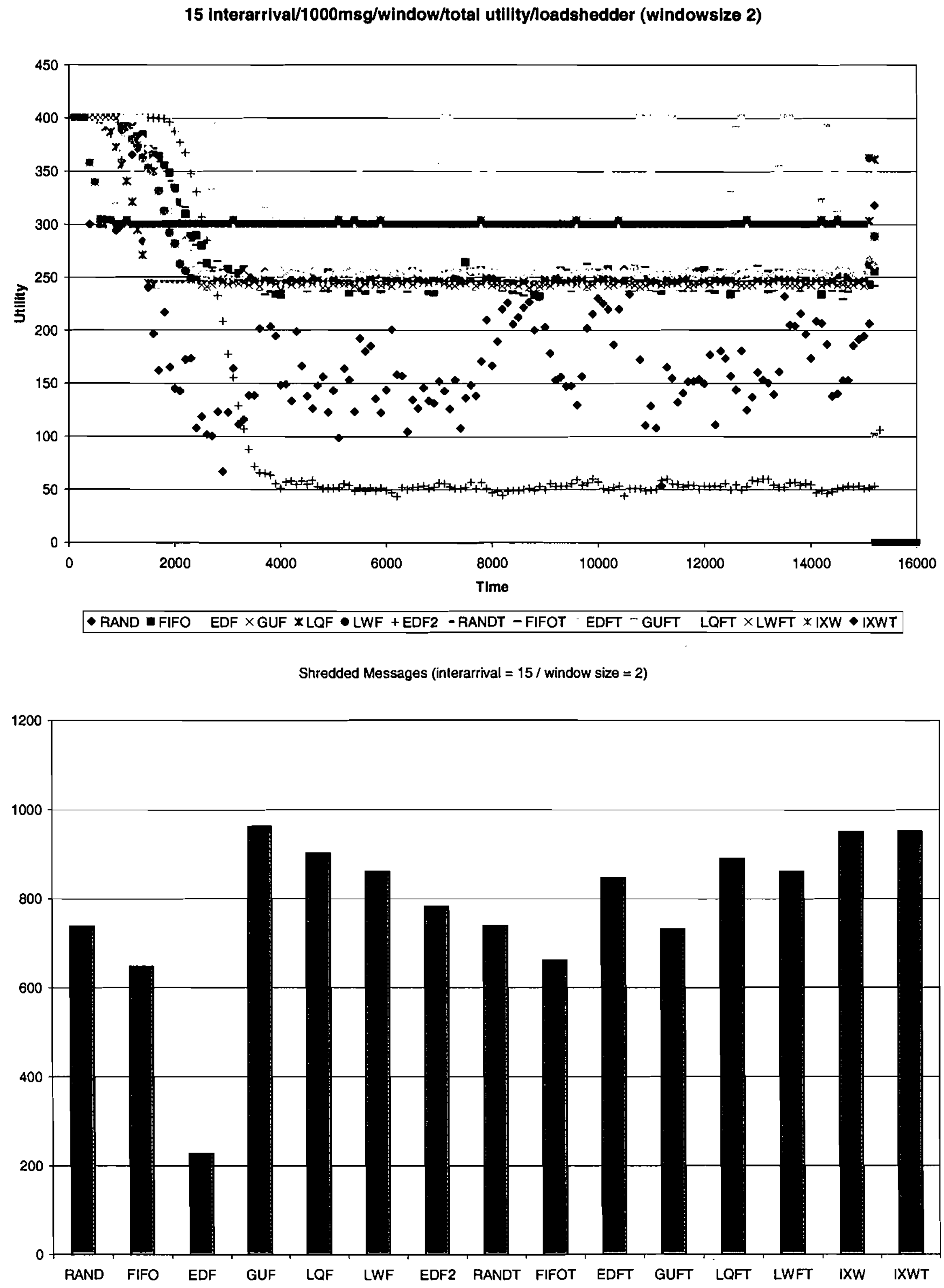


Figure 21: Inter arrival time of 15 with load shedder.

EDF seems the best algorithm here because it shed the least load while maintaining some utility throughout the simulation.

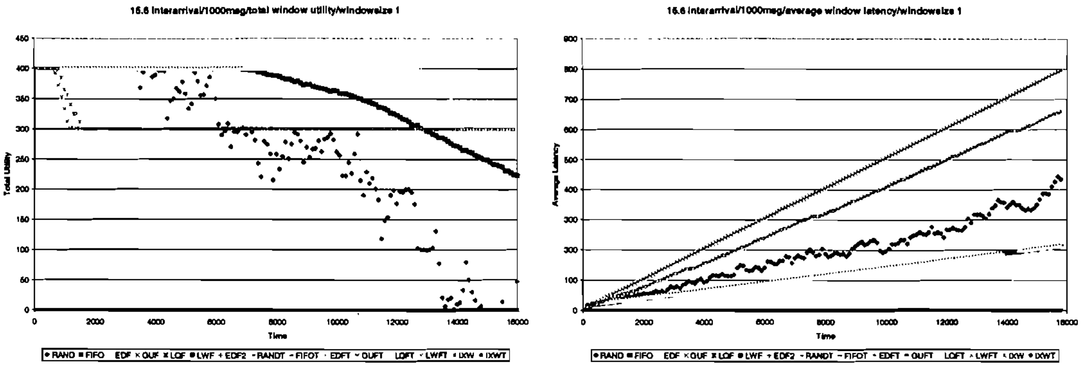


Figure 24: Inter-arrival time at 15.8.

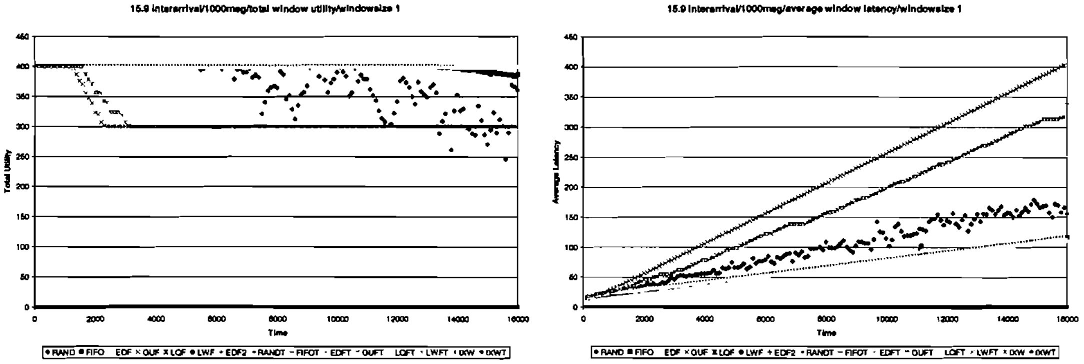


Figure 25: Inter-arrival time at 15.9.

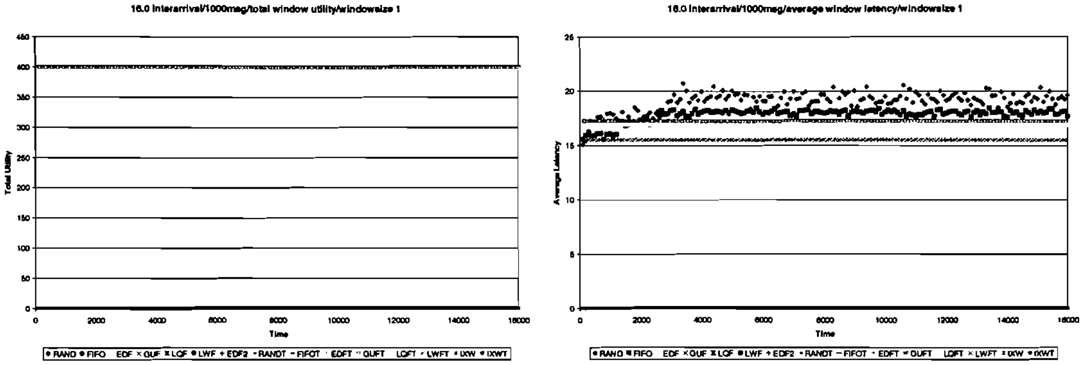


Figure 26: Inter-arrival time at 16.0.

4.6.4 Observations:

1. Various algorithm performance starting to degrade at different point, however, it would seem they will all eventually go down when the network is overloaded.
2. The original assumption from experiment 2 still stands. The time needed for any algorithm to make to the output is 16. Anything less than that will cause the message to pile up in the queue and eventually degrade the network performance.

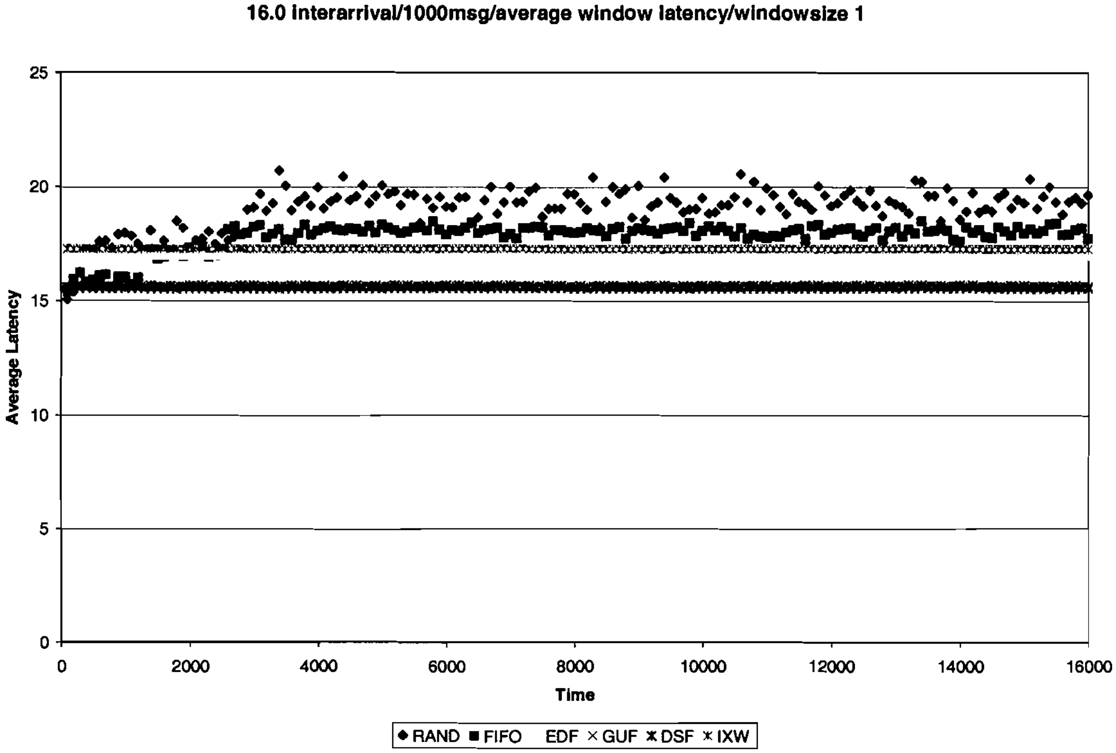


Figure 28: Latency for 16

Although the utility at inter-arrival time of 16 are 100% for every one, the latency are different. Some messages are block at the most down stream box, and wasting unnecessary time before being put thru. In this latency graph the advantage of DSF can be clearly shown. These little places to save time at the end could make significant difference when the network scales up.

4.7.4 Observations:

1. DSF is believed to cause the least blocking time by pushing the most down stream box out to the output.
2. At optimal inter-arrival rate, DSF performs almost as well as GUF, yet slightly slower (latency). But significantly better than any other algorithm.
3. At near optimal inter-arrival rate, DSF perform between GUF and EDF.
4. DSF still has the problem of sacrificing one entire application output. However, it does so a lot latter than GUF/IXW, yet earlier than EDF

5.4 Load Shedding

It is important for the scheduler to operate near the optimal load. It is also important to prevent useless messages to block the entire message queue. A way to satisfy these is to take away some messages by force. From the experiment of simple load shedder, we found that it does give good performance in terms of latency. However, the other dimension of problem which didn't exist before, the quality, or rather, the accuracy of results, was not able to be measured from this simulation model.

5.5 Minimizing Wait Time

From the beginning we know that the scheduler can't make the CPU faster. One place where unnecessary wait time can be eliminated is the message queue at the last box prior to leaving the network. By holding on the last box and process other upstream boxes will not cut any time for the upstream messages to arrive earlier. However, by processing the most downstream box first will reduce the latency of those messages and consequently improve their utility as well.

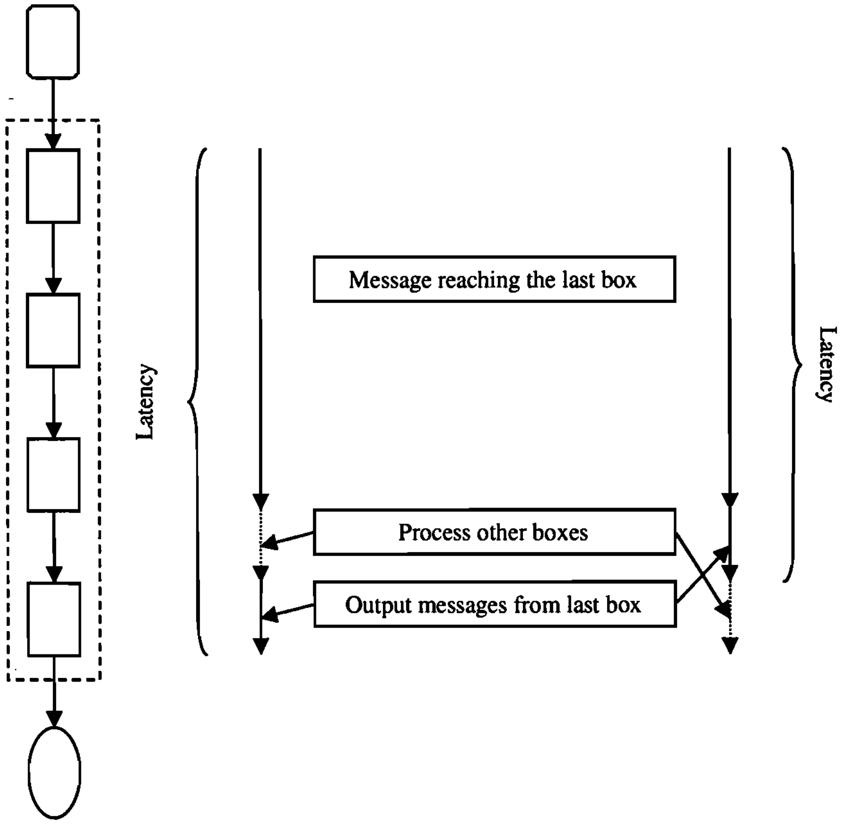


Figure 29: Save time with DSF

By push out the message as soon as message arrive the last box, the time from waiting other boxes to be processed can be saved.

7.2 Better ways to specify utility and deadline

Quality of service is and important clue for scheduling. However, looking at only the current utility of current tuple is not enough. The scheduler needs to be able to predict the future utility needed for reaching the output further downstream.

A more formal way of specifying quality of service might be needed here. The definition of a negative slope and the obtainment of utility were not very clear. In addition, the scheduler might want to provide some suggested region where it's capable to make some differences. After all, the users do not know the capacity of the network. If the QOS is specified too strictly, there will be very little room for the scheduler to work with.

Another way of defining quality of service might be to have multiple deadlines. So the QOS graph looks like steps instead of slopes. As a result, the scheduler could work with a second deadline should it miss the first one.

7.3 Optimized scheduling plan and dynamic adjustments

Another dimension to explore is dynamic scheduling plan. An initial guess of scheduling plan can be made when the network starts up. During run-time, depending on how well the initial plan is doing, the scheduler can dynamically adjust its plan of scheduling based on the feedback of the QOS monitor.

However, how expansive this scheduling algorithm would be is unknown. If Aurora is eventually going to be handling 10^6 boxes, looking through the entire network and recomputing new dynamic optimization plans each time might not be feasible.

Consequently, perhaps an intelligent QOS feedback monitor could communicate with the scheduler regarding any existing urgency and whether the original plan needs to be modified or not, since the static guess at compile-time couldn't take factors such as load variation into account. The QOS monitor should also be able to help adjust the optimization plan to the extent where the scheduler would not need to recalculate the entire network for new scheduling plans too frequently.