Scheduling in Distributed Stream Processing Systems

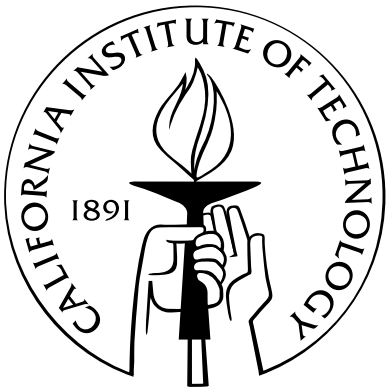
Thesis by

Andrey Khorlin

In Partial Fulfillment of the Requirements

for the Degree of

Master of Science



California Institute of Technology

Pasadena, California

2006

(Submitted May 24, 2006)

Contents

Acknowledgements

I would like to thank Professor K. Mani Chandy for providing me with great help, support and guidance. Through our work on this thesis, I was able to gain a lot of valuable experience. I am also grateful to other members of the Infospheres Lab including Agostino Capponi, Lu Tian, and Daniel Zimmerman for creating a great working environment and providing me with very valuable feedback that helped me in my work on this thesis.

Also, I would like to acknowledge IBM's SMILE Team, including Chitra Dorai, Gerry Buttner, Roman Ginis, Jianren Li, Rob Strom, and Yuanyuan Zhao, with whom I spent a year working as an intern. They provided me with great working experience and exposed me to the area of stream processing, which helped me to arrive at many ideas presented in this thesis.

Last, but not least, I must thank my lovely wife, Olga, for enduring our separation for the past three years and helping us to keep our relationship as strong as the day we got married.

This work was funded in part by the National Science Foundation under ITR grant, CCR-0312778 and also by the Lee Center on Networking at the California Institute of Technology.

Abstract

Stream processing systems receive continuous streams of messages with relatively raw information and produce streams of messages with processed information. The utility of a stream-processing system depends, in part, on the accuracy and timeliness of the output. Streams in complex event processing systems are processed on distributed systems; several steps are taken on different processors to process each incoming message, and messages may be enqueued between steps. This work explores the problem of distributed dynamic control of streams to optimize the total utility provided by the system. A system can be controlled using central control or distributed control. In the former case a single central controller maintains the state of the entire system and controls the operation of all processors. In distributed control systems, each processor controls itself based on its state and information from other processors. A challenge of distributed control is that timeliness of output depends only on the total end-to-end time and is otherwise independent of the delays at each separate processor whereas the controller for each processor takes action to control only the steps on that processor and cannot directly control the entire network. In this work, we discuss a framework for design and analysis of the control-based scheduling algorithms for a distributed stream processing system and illustrate our framework with two concrete scheduling algorithms.

Chapter 1

Introduction

1.1 Background

Message-driven or *event-driven* applications are very well known and used in computer science . The model for such applications is that application components exchange information via sending and receiving messages (or events), which travel along the channels connecting them. This model is very general and is therefore applicable to many practical systems. The model is useful not only for distributed applications, but also in the design of user interfaces, hardware architectures and other domains. In fact, the message-based model describes well the mode of communication among different entities in the real world such as departments, companies and people. Thus, any extensions to this model are applicable outside of computer science and distributed systems.

1.2 Streams and Stream Processing

Streaming applications can be viewed as specialized extensions of message-driven applications. In this context, a *stream* is defined as sequence of messages emitted by a single source with every message having the same schema. Therefore, *stream processing* is a class of computations that involves receiving one or more streams of messages, executing a program against incoming streams, and producing one or more output streams. The computation can also change, but is assumed to change less frequently than other system parameters. A streaming application is a message-driven application that performs stream processing on several input streams. This definition encompasses many applications. To narrow the definition for the purposes of this discussion, the following key attributes of a streaming application are assumed:

- **Statefulness:** The tasks may have state. This is different from filtering or sampling applications that perform computation on one message at a time.
- **Rapid Input Rate:** The message arrival rate is high.

- **Dynamism:**
  - Arrival and service rates for each stream may change rapidly.
  - The number of streams produced and consumed by the application may change at run time.
  - The computation performed by the application may change, but much less frequently than other system attributes.
- **Performance Requirements:** Clients represent performance requirements in the form of *Quality of Service (QoS)* functions, which define penalties for the delay incurred in producing output.
- **Multiple Streams:** There may be several streams of information flowing through the applications that may or may not share some computation.
- **Multiple Users:** The results are delivered to multiple application users.

Not all stream processing systems exhibit all these features. Only the first two properties are truly necessary for a system to be a stream processing system. In fact, there are several stream processing systems described in the literature that lack one or more elements of the above list [4].

1.3 QoS-based Scheduling in Distributed Stream Processing Systems

The requirements described in the previous section put new demands on system design. At the very least, a stream processing system has to process huge numbers of messages while potentially maintaining a large amount of state. Moreover, if the extended requirements for dynamism and performance are necessary, the system must be aware of them too. Since a centralized system is often not capable of meeting such requirements, a distributed implementation of the stream processing system is required. In a distributed stream processing application, computation is broken up into several pieces that are placed onto separate machines.

In such settings, we explore the design of scheduling algorithms that are needed on each machine of a distributed stream processing system. Our novel approach has two parts. First, we realize that in a distributed stream processing system queuing delays will be a large part of total end-to-end delay. This view is different from those approaches proposed in Carney *et al.* [11] and Babcock *et al.* [3]. Scheduling should be designed to alleviate these queuing delays. Second, we treat scheduling as a control problem, which is similar to the approach taken in Paganini *et al.* [39]. Scheduling at each server can be controlled using feedback information about global conditions of the system,

Parameter	Description
S	The set of servers in a topology
C	The matrix of link capacities between the servers. $C_{ij} = 0$ if there is no link between server i and j. $C_{ij} = c > 0$ if there is a link with distribution of transmission delays c between server i and j, where $i, j \in [0, \|S\|]$ *Note: The matrix is symmetric because links between servers are bidirectional.*

Table 1.1: Physical Topology Parameters

allowing each server to converge to an optimal scheduling policy. Thus, we can use the theory of control design to analyze the speed of convergence and stability of our algorithms. We believe that this approach will yield a scheduling algorithm with greater robustness to uncertainty. Similar to Carney *et al.* [11], we define a QoS function for each flow, which quantifies the cost of delay for every message in every stream. These costs impact the order in which messages should be processed at each server to improve application performance.

1.4 Formal Problem Definition

In this section, we provide a formal definition of the problem. We start by defining server topology and stream computation, and conclude with a definition of scheduling as an optimization problem.

1.4.1 Server Topology

A server topology represents a distributed set of servers that are connected by a set of links. Each server has a certain computational capacity. This capacity is determined by the server's architecture, *i.e.*, CPU speed, cache size, memory size, *etc.* Similarly, each link has an associated capacity. For the purpose of defining an abstract framework, we use the distribution of service times needed to process each stream's messages to represent the computational capacity of a server. Link capacity is represented as the distribution of time delays for messages traveling on the link (see Table 1.1).

1.4.2 Computation

A *streaming computation* is a directed acyclic graph (DAG) in which each node represents an indivisible unit of a computation and each link represents data flow. A node with zero in-degree is called a *source*, and a node with zero out-degree is called a *sink*. All data produced by sinks represents the total result of a streaming computation. A *distributed stream processing system* consists of a set of such streaming computations. Each computation is distributed among $|S|$ servers available in the topology. The mapping of computation on this topology is outside of the scope of this work,

Parameter	Description
F	Set of streaming flows where each flow f_i is a directed acyclic graph.
f_k	Single streaming flow defined as a tuple $< \tilde{C}_k, \tilde{F}_k >$, where $\tilde{C}_k$ is a set of computations and $\tilde{F}_k$ is a connectivity matrix such that $\tilde{F}_{k_{ij}} = 1$ if $\tilde{c}_i \in \tilde{C}_k$ sends output to $\tilde{c}_j \in \tilde{C}_k$ and $\tilde{F}_{k_{ij}} = 0$ otherwise.
$m : \mathbf{R}^2 \rightarrow \mathbf{R}$	Mapping function $m(k,i) = j : \tilde{c}_i \in \tilde{C}_k$ such that $f_k \in F$ is mapped to server $s_j \in S$
Y_{ij}	Set of computations from flow j placed on server i, *i.e.*, $\forall \tilde{c}_k \in Y_{ij}, \tilde{c}_k \in \tilde{C}_j \wedge f_j \in F \wedge s_i \in S \wedge m(k,i) = j$

Table 1.2: Streaming Computation Parameters

Parameter	Description
QoS Function	$q : \mathbf{R} \rightarrow \mathbf{R}$ maps delay to measure of cost. Measure of cost is comparable across streams.
Delay computation	$\forall \tilde{c}, \tilde{c}(m_1, m_2, ..., m_n) = \grave{m}$ $time - stamp(\grave{m}) = min_{\forall i \in [0,n]}(time - stamp(m_i))$ where $time - stamp(m)$ returns time when messages used to generate m or m itself has entered the system.

Table 1.3: Quality of Service Function

therefore we assume that a mapping function, m, is given to us (see Table 1.2). There are several mapping strategies proposed in the literature [6, 55, 59].

1.4.3 Distributed Scheduling as Optimization over a Queuing Network

Now, we define a notion of *quality of service* and formulate the distributed scheduling problem as an optimization problem over a queueing network. Each flow $f_i \in F$ is said to have a quality of service function, $q : \mathbf{R} \rightarrow \mathbf{R}$. This non-decreasing function maps the delay for every message produced by every $\tilde{c}_k \in \tilde{C}_i$ with $\tilde{F}_{i_{km}} = 0$, for all m (*i.e.*, each node with zero out-degree).

The delay for each message is computed as follows. Every message entering the system is time-stamped. Each operator in a data flow takes several input messages and produces zero or more messages [1, 4, 29]. An output message carries the highest time-stamp from all messages used to create it. When a final output message arrives at a sink, the difference between the current time and the time-stamp is taken. This difference is called *message delay*, d, and $q(d)$ is the cost of the delay (see Table 1.3).

A set of streaming flows distributed over a network of servers is viewed as a queuing network, where each server has a set of queues corresponding to each flow's input (see Definition 1.4.1). The queues are connected if there is data flowing between two servers as defined by the structure of the flows and the mapping function. The scheduling at each server is defined as an optimal selection policy for enqueued messages at each local server such that a global cost-based system performance

metric is minimized. Currently, our optimization metric is defined as the average cost for all flows in the system, where the cost of a flow is defined as the average cost of all messages produced by the flow so far (see Definition 1.4.2).

Definition 1.4.1. *Let Q_{mj} be a queue on server s_i that corresponds to a sub-flow Y_{mj}, and let $[Q_{m1}, ..., Q_{mn}]$ be a state of all queues on server s_i, and ω be some feedback information at time t. Then the* scheduling function, *sched : $(s_m, Q_{m1}, ..., Q_{mn}, t, \omega) \rightarrow \tilde{j}_k$, determines which job $\tilde{j}_k \in Q_{mk}$ should be executed in the next Δt.*

Definition 1.4.2. *The* Distributed Scheduling Problem *is to find a* scheduling function, *sched, that minimizes $\frac{1}{|F|} \lim_{t\rightarrow\infty} \frac{\sum_{i=0}^{|F|} \sum_{t=0}^{N_t^i} c(d_t^i)}{N_t^i}$ where N_t^i is the number of events received by stream i before time t.*

Note: Definition 1.4.1 does not force the algorithm to use any specific feedback information. In the next chapter, we discuss the concrete assumptions behind this definition that are used in this work.

Chapter 2

Problem Space and Proposed Scheduling Algorithms

2.1 Single Server: Static Mean-Based Scheduling

In this chapter, we present two approaches to solve the distributed scheduling problem. The first approach is a static mean-based scheduling (SMBS) algorithm. The second approach is a dynamic queue-control (DQC) algorithm. We design both approaches in a single server environment and then outline the extensions needed to move these algorithms to a complete queueing network. In addition, we discuss in detail the current assumptions and limitations of both approaches. A summary of our framework is presented in Figure 2.1.

Our first approach is based on a process sharing algorithm that splits a server's processing capacity among all streams allocated on the server. The process sharing works dynamically such that the capacity is divided among non-idle streams. For this process sharing scheme, we use stream statistics to determine each stream's expected delay under a given share of the total capacity. Then, we use this analysis iteratively to determine the share that minimizes the metric introduced in Section 1.4.3. Since the analysis uses expected queue sizes and delays, we call this approach *static mean-based scheduling* (SMBS).

2.1.1 Process Sharing Algorithm

At the core of the SMBS approach is the dynamic process sharing algorithm. This algorithm splits the processing capacity needed to execute jobs for each local stream according to the stream's priorities. However, unlike classical static process sharing, which divides the capacity permanently among different jobs, the division in this algorithm only occurs when two or more local streams are competing for resources.

The dynamic process sharing algorithm works as follows. For each flow f_j on server s_i there is a queue of messages Q_{ij}. The algorithm serves the message from the head of the queue. Thus, no

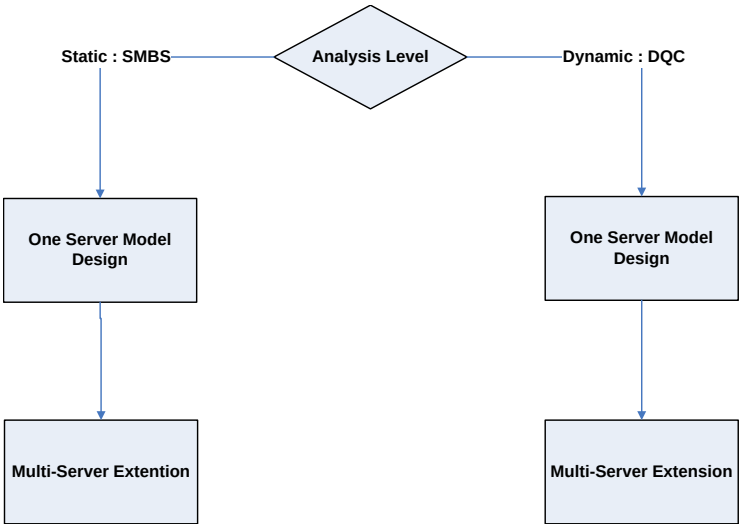


Figure 2.1: Analytical Framework

message reordering is done. The server processes each message until the local stream's computation, Y_{ij}, produces zero or more messages. Therefore, scheduling decisions are not made on a per operator basis and intermediate results of the computation are not persistent under this scheduling technique, unlike that of Babcock *et al.* [3]. The scheduling algorithm performs process sharing. Each queue has a priority number assigned to it. If there is more than one non-empty queue, the scheduler gives a share of the local processing capacity based on the priority number of each queue (see Algorithm 1).

Algorithm 1 is executed whenever the current job in stream j finishes processing or a new job arrives into an empty queue. Later we will see that the priorities $\{p_1, ..., p_n\}$ can be adjusted to regulate how much of the capacity share each stream should get. This will be at the core of determining a local scheduling policy such that the overall global objective function is minimized. Moreover, we will prove that this mode of sharing is always more efficient than the classical static process sharing scheme in which the shares are not adjusted at run-time. This result is presented in Section 3.2.

2.1.2 Process Sharing Algorithm Model

In order to determine priorities for the process sharing algorithm, we need to understand its behavior. We use queuing theory the same way it is used to derive expected queue sizes for other types of scheduling such as FIFO. We represent our scheduling process as a Markov Chain for which the invariant probabilities are computed. From the invariant probabilities, the expected queue size for each local stream is derived. Then, by application of Little's Law, the average delay is determined.

For the purposes of defining a Markov chain we assume that for each local stream Y_{ij}, arrival

Algorithm 1 *schedule*($\{Q_{i1}, ..., Q_{in}\}$, $\{m_{i1}, ..., m_{in}\}$, $\{p_1, ..., p_n\}$,k)

Require: $\exists Q_{ij}$ such that $|Q_{ij}| > 0$
Require: $\exists m_{ij}$ such that $m_{ij} \neq \emptyset$
Require: $\sum_{j=1}^{n} p_j = 1$
Require: $k \in [1, n]$
Require: $\{Q_{i1}, ..., Q_{in}\}$: set of queues corresponding to local flows
Require: $\{m_{i1}, ..., m_{in}\}$: set of currently executing jobs
Require: $\{p_1, ..., p_n\}$: set of priorities
Ensure: $\sum_{j=1}^{n} \tilde{p}_j = 1$
1: **if** *done*(m_{ik}) **then**
2: send(m_{ik})
3: **end if**
4: **if** *size*(Q_{ik}) > 0 **then**
5: $m_{ik} \leftarrow$ *dequeue*(Q_{ik})
6: **end if**
7: **for all** $m_{ij} \in \{m_{i1}, ..., m_{in}\}$ **do**
8: *totalUsedWeight* $\leftarrow$ *totalUsedWeight* $+ p_j$ {Aggregate shares for all executing jobs}
9: **end for**
10: **for all** $m_{ij} \in \{m_{i1}, ..., m_{in}\}$ **do**
11: $\tilde{p_j} \leftarrow \frac{p_j}{totalUsedWeight}$
12: *execute*(m_{ij}, $\tilde{p_j}$) {continue job execution with given fraction of the CPU}
13: **end for**

and service rates are distributed exponentially with means λ_j and μ_j respectively. Therefore, we can construct a continuous Markov chain (CMC) in which each state is a vector of queue sizes for each local stream Y_{ij} (the size of a queue includes the job that is currently being served). An example of such a Markov chain for two queues with maximum size of two messages is shown in Figure 2.2.

The outgoing transitions from each state depend on queue size. If the queue size is zero, then stream share is distributed among the non-empty queues. The following is the definition of the transition function t that determines the rate of transition from state i to state j.

Definition 2.1.1. *Given a scheduling algorithm on n queues and given two states $S_i = [s_{i1}, ..., s_{in}]$ and $S_j = [s_{j1}, ..., s_{jn}]$, the* transition function $t : \{S_i, S_j\} \rightarrow \boldsymbol{R}$ *specifies the transition rate from state S_i to state S_j and is defined as*

$$t(S_i, S_j) = \begin{cases} \lambda_i & \text{if } \exists s_{ik} \in S_i, \exists s_{jm} \in S_j :: (s_{jm} - s_{ik}) = 1 \wedge \forall \acute{k} \neq k, \acute{m} \neq m \ s_{j\acute{m}} = s_{i\acute{k}}; \\ \omega_i \mu_i & \text{if } \exists s_{ik} \in S_i, \exists s_{jm} \in S_j :: (s_{jm} - s_{ik}) = -1 \wedge \forall \acute{k} \neq k, \acute{m} \neq m \ s_{j\acute{m}} = s_{i\acute{k}}; \\ 0 & \text{otherwise}; \end{cases}$$

where $\omega_i = \frac{p_i}{\sum k : s_{ik} \neq 0 : p_k}$.

Proposition 1. *Let M be a continious ergodic Markov chain with transition function t and invariant distribution* Π. *The total residence time for jobs in local flow Y_{ij} is*

$$\overline{W}_i = \frac{\sum_{j=1}^{\infty} s_{ij} \pi_i}{\lambda_i}, \text{ where } \pi_i \in \Pi.$$

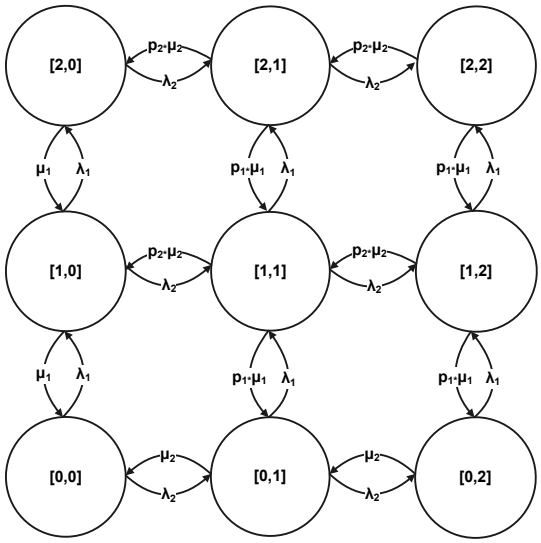


Figure 2.2: Example of continuous Markov chain for two queues

Proof. $\overline{W}_i = \frac{\overline{q}_i}{\lambda_i}$ by Little's Law, where $\overline{q}_i$ is expected queue size. Then, since M is ergodic, $\overline{q}_i = \sum_{j=1}^{\infty} s_{ij}\pi_i$. □

The Markov chain presented here could be viewed as a multi-dimensional grid of states where the number of dimensions is equal to the number of queues being analyzed. For example, two queues yield a two-dimensional mesh in the first quadrant, while three queues yield a cube in the first quadrant. We will use this representation in our further discussion.

2.1.3 Process Sharing Algorithm Model Approximation

Before we can use the abstraction introduced in the previous section, we need to determine a concise way of making predictions with it. Definition 2.1.1 involves an infinite Markov chain, which lacks both global and local balance. One way to attack the complexity of the chain is to determine the average amount of sharing that takes place among the streams. If the amount of sharing is known, we can break down the chain into the interior case, when all queues are non-empty, and a set of boundary cases. Then, the model can be disaggregated into subsets that can be analyzed with standard techniques. However, approximating the sharing factor is very complex. The complexity can be illustrated by the case of two streams. We can write down the following equations from the assumption that, in a steady state, the arrival rate equals the service rate:

$$\begin{aligned}
\lambda_1 &= p_b^1\mu_1 + p_i p_1 \mu_1 \quad \text{where } p_b^1 = Pr(|Q_{i2}| = 0 \wedge |Q_{i1}| > 0),\ p_b^2 = Pr(|Q_{i1}| = 0 \wedge |Q_{i2}| > 0) \\
\lambda_2 &= p_b^2\mu_2 + p_i p_2 \mu_2 \quad \text{where } p_i = Pr(|Q_{i2}| > 0 \wedge |Q_{i1}| > 0)
\end{aligned}$$

Now, we have two equations and three unknowns, p_b^1, p_b^2 and p_i. We cannot use an obvious third equation, $p_b^1 + p_b^2 + p_i = 1$, because it can be derived from the other two equations. A ratio between interior and boundary probabilities, $\frac{p_i}{p_b^1+p_b^2}$, could be used as a third equation. However, approximating this ratio is as complex as the original goal.

In the absence of a closed-form solution, we use a numerical approximation of the CMC model. We can approximate the infinite chain with a finite chain assuming that queues may not exceed a certain maximum size, q_{max}. If q_{max} is greater than or equal to the true maximum possible size, then the finite model approximates the infinite model perfectly.

Once the model is converted to a finite chain, we can use standard matrix solution to find invariant probabilities: $\pi Q = 0$, where Q is the transition matrix [38] (see Definition 2.1.2). However, since Q is singular, we need to perform either equation elimination or equation replacement to find invariant probabilities. More importantly, matrix Q is very large and its size grows exponentially with the number of queues. For example, with a five queue model where each queue does not exceed ten jobs, the total number of cells in Q is $(10^5)^2$, which clearly cannot fit in memory or be processed in reasonable time.

Definition 2.1.2. ***(Q-Matrix)*** *Given continuous Markov chain M for scheduling process over $\{Q_{i1}, ..., Q_{in}\}$, where $Q_{ij} \in [0, n_j]$, and corresponding transition function t, let* projection function *$s : \boldsymbol{R} \to \boldsymbol{R}^n$ project a column or row index of Q to a state $S_j \in M$:*

$$s(k) = S_j, \text{ where } S_j = [s_{j1}, ..., s_{jn}] \text{ and } s_{jm} = R(m)\%(n_m + 1)$$

$$R(m-1) = \frac{R(m)}{(n_m + 1)} + s_{jm} \ , \ R(n) = k \wedge m \in [1, n]$$

Define Q *as*

$$Q_{ij} = \begin{cases} t(s(i), s(j)) & \text{if } i \neq j; \\ \sum_{m=1}^{rank(Q)} t(s(i), s(m)) & \text{if } i = j; \end{cases}$$

$$rank(Q) = \prod_{i=1}^{n} n_i$$

Since direct evaluation of invariant probabilities is not feasible, we present an optimized algorithm for finding an approximate invariant distribution. It is based on converting Q into transition probability matrix P, which can be represented concisely using Definition 2.1.2. Thus there is no extra memory needed to look up values of P, since they are obtained by formula evaluation. The invariant probability π can be found iteratively (see Algorithm 2):

$$\pi_t^T = P\pi_{t-1}^T.$$

Furthermore, the vector π_{t-1}^T can be represented concisely by rounding numbers in π_{t-1}^T that are within ϵ of zero. The vector π is sparse, with non-zero entries occurring in consecutive regions. Therefore, it can be represented as a segment tree that stores information about non-zero segments and allows $O(log(|S|)$ time lookup of entries in π, where $|S|$ is the number of segments. Usually, the number of segments is small because the state's probability is centered around the origin.

With concise representations of P and π, we perform iteration much faster with much less memory than using a standard matrix approach. The questions of what the initial value of π should be and how we determine termination of the algorithm efficiently remain. To seed vector π we use the following heuristic. From FIFO M/M/1 queue analysis, we know that the probability that the queue has size m is

$$Pr(|Q_{ij}| = m) = \rho_j^m(1 - \rho_j) \text{ where } \rho_j = \frac{\lambda_j}{\mu_j}.$$

We compute the probability for n up to the maximum size of the queue and then distribute the probability in a uniform fashion among the states whose total queue sizes are the same (see Algorithm 3).

Algorithm 2 $predict(P, N_{iter}, \epsilon) \rightarrow [\overline{q}_1, ..., \overline{q}_n]$

$\pi \leftarrow initialize()$
$[\tilde{q}_1, ..., \tilde{q}_n] \leftarrow [0, ..., 0]$ {expected queue sizes from previous iteration}
$[\overline{q}_1, ..., \overline{q}_n] \leftarrow [0, ..., 0]$
$iteration \leftarrow 0$
$sum \leftarrow 0$
while $!done([\tilde{q}_1, ..., \tilde{q}_n], [\overline{q}_1, ..., \overline{q}_n], \epsilon) \wedge iteration < N_{iter}$ **do**
  for all $i \in [1, rank(Q)]$ **do**
    $nonZeroEntries \leftarrow getNonZeroIndices(i)$
    for all $j \in nonZeroEntries$ **do**
      if $\pi_j \neq 0$ **then**
        $\tilde{\pi}_j = \tilde{\pi}_j + \pi_j t(i, j)$
      end if
    end for
  end for
  $sum \leftarrow \frac{-1}{t(i,i)}\tilde{\pi}_j$ {weigh next state probability by the holding time}
  $iteration \leftarrow iteration + 1$
  for all $i \in [1, rank(Q)]$ **do**
    $state \leftarrow s(i)$
    $normProb \leftarrow \frac{-t(i,i)^{-1}\pi_j}{sum}$
    $[\tilde{q}_1, ..., \tilde{q}_n] \leftarrow [\tilde{q}_1, ..., \tilde{q}_n] + normProb * state$
  end for
end while
$\pi = \tilde{\pi}$
return $[\overline{q}_1, ..., \overline{q}_n]$

In order to determine if the termination condition is reached, the following strategy is used. During evaluation of π, expected queue sizes are computed. Then they are compared to expected queue sizes from the previous iteration. If the difference is less than ϵ, the algorithm is terminated.

Algorithm 3 $initialize() \rightarrow \pi$

Require: $f(n,q) \rightarrow \mathbf{R}, f(n,q) = \sum_{k=1}^{n+1} f(k-1,q-1) \wedge f(n,2) = n+1$: computes total number of states with total size n over q local queues

```
for i = 0 to n do
    size ← Σ_{i=1}^{n} s(i)
    π_i ← (ρ_i^{size}(1 − ρ_i))/f(n, q)
end for
return π
```

In addition, a maximum number of iterations can be set after which termination is mandatory. Both of these termination conditions are used in our approximation algorithm (see Algorithm 4).

Algorithm 4 $done([\overline{q}_1, ..., \overline{q}_n], [\tilde{q}_1, ..., \tilde{q}_n], \epsilon)$

```
for i = 0 to n do
    if |q̄_i − q̃_i| > ε then
        return false;
    end if
end for
return true
```

2.1.4 Optimum Process Sharing Parameters

Once we have a scheduling algorithm whose behavior we can predict, we can use this prediction method to find the optimal parameters $[p_1, ..., p_n]$ to minimize the cost of average delay incurred by each stream. At this point several optimization techniques can be employed. The most general approach based on Genetic Algorithms is utilized to allow the most general family of quality of service functions to be used. The genome is just a set of integers. When they are normalized, they yield $[p_1, ..., p_n]$. The evaluation function is the prediction algorithm that determines average delay given $[p_1, ..., p_n]$ (see Algorithms 5 and 6).

Algorithm 5 $findOptimalPartition(N) \rightarrow [p_1, ..., p_n]$

```
population ← getInitial()
for i = 0 to N do
    population ← evolve(evaluator)
end for
[p_1, ..., p_n] ← normalize(getFittest(population))
return [p_1, ..., p_n]
```

2.2 Single Server: Dynamic Queue Control Scheduling

In this section, we propose another algorithm that allows marginal cost-based scheduling and achieves provably optimal outcome in the case of one server. In a later section, we discuss how this algorithm

Algorithm 6 $evaluator(P, N_{iter}, \epsilon, [p_1, ..., p_n]) \rightarrow cost$

1: $delay \leftarrow predict(P_{[p_1,...,p_n]}, N_{iter}, \epsilon)$
2: $cost \leftarrow q_i(delay)$;
3: **return** $cost$

can be extended to a distributed multi-server environment. Unlike the algorithm proposed before, this algorithm makes control decisions at every Δt (for discretized time scale, we assume $\Delta t = 1$). Assuming that there is no cost of switching between tasks, the algorithm selects the most profitable piece of work from all available unfinished jobs in all queues to be executed in the next Δt. The algorithm schedules the job whose service rate scaled by the first derivative of the QoS function at the point equal to the total amount of time spent in the system is the greatest (see Algorithm 7).

Algorithm 7 $dynamicQueueControl([m_1^k, ..., m_N^k])$

Require: $k \in [0, n]$ where n is number of local flows
Require: m_i^k is a message from local flow k
Require: $q_k(x)$ is a QoS function for local flow k
Require: $T(m_i^k) = time - stamp(m_i^k)$ is time spent in the system since entry
1: $maxCost \leftarrow 0$
2: $j \leftarrow 0$
3: **for** $i = 0$ to N **do**
4: $\quad marginalCost \leftarrow \mu_k \left.\frac{dq_k(x)}{dx}\right|_{x=T(m_i^k)}$
5: $\quad$ **if** $marginalCost > maxCost$ **then**
6: $\quad\quad j \leftarrow i$
7: $\quad\quad maxCost \leftarrow marginalCost$
8: $\quad$ **end if**
9: **end for**
10: $\tilde{m}_j^k \leftarrow execute(m_j^k, \Delta t)$ {run job m_j^k for Δt units of time; if execute produces output, the job is done and its results are sent downstream}
11: **if** $\tilde{m}_j^k \neq \emptyset$ **then**
12: $\quad end(\tilde{m}_j^k)$
13: **end if**

This scheduling approach allows messages to be processed out of order. For some streaming applications reordering messages may be infeasible. We discuss the issues of message reordering more in Section 2.3. However, if message reordering is inappropriate, the algorithm can be modified to select only among the messages at the heads of the local queues instead of the total set of all waiting messages. Below, we propose a modified version of Algorithm 7 for the case when message reordering is not permitted. We prove optimality of this algorithm in Section 3.5.

The key modification is the addition of queue size into the metric used to select the next message for processing. If the message that is going to be processed, m_1^i, in the next Δt finishes, then not only is the cost of further delay for this message avoided as in Algorithm 7, but the waiting cost for all messages in that queue is also reduced. For each waiting message, the impact of the queue size on the cost, $costRedDueToQueueSize$, is computed by taking the mean service time ${\mu_i}^{-1}$ and

multiplying it by the number of messages that are in front of each message in the queue. If there is a model that can predict service time for each message, then the average ${\mu_i}^{-1}$ can be replaced by the prediction from the model $\tilde{\mu}_i^{-1}$.

$$costRedDueToQueueSize = \left(\sum_{j=1}^{n_i} \left. \frac{dq_i(x)}{dx} \right|_{x=T(m_j^i)+j{\mu_i}^{-1}} - \sum_{j=2}^{n_i} \left. \frac{dq_i(x)}{dx} \right|_{x=T(m_j^i)+j{\mu_i}^{-1}} \right)$$

We call the modified algorithm DQC-NR where "NR" stands for "No Reordering" (see Algorithm 8).

Algorithm 8 $dynamicQueueControlNR(s_i, F, Q, \tilde{Q})$

Require: $s_i \in S$ is a server
Require: F is a set of flows allocated on s
Require: Q is a set of quality of service functions where $q_i(x)$ is the QoS function for flow $f_i \in F$
Require: $\tilde{Q}$ is a set of queues for each flow f_i such that $\tilde{q}_i \in \tilde{Q} : \{m_1^i, ..., m_{n_i}^i\}$, where m_1^i is at the head of the queue
Require: $T(m_i^k) = time - stamp(m_i^k)$ is time spent in the system since entry
1: $maxCost \leftarrow 0$
2: $j \leftarrow 0$
3: **for** $i = 0$ to N **do**
4: $\quad costRedDueToQueueSize \leftarrow \left(\sum_{j=1}^{n_i} \left. \frac{dq_i(x)}{dx} \right|_{x=T(m_j^i)+j{\mu_i}^{-1}} - \sum_{j=2}^{n_i} \left. \frac{dq_i(x)}{dx} \right|_{x=T(m_j^i)+j{\mu_i}^{-1}} \right)$
5: $\quad marginalCost \leftarrow \mu_i \left(\left. \frac{dq_i(x)}{dx} \right|_{x=T(m_1^i)} + costRedDueToQueueSize \right)$
6: $\quad$ **if** $marginalCost > maxCost$ **then**
7: $\quad\quad j \leftarrow i$
8: $\quad\quad maxCost \leftarrow marginalCost$
9: $\quad$ **end if**
10: **end for**
11: $\tilde{m}_1^j \leftarrow execute(m_1^j, \Delta t)$ {run job m_1^j for Δt units of time; if execute produces output, the job is done and its results are sent downstream}
12: **if** $\tilde{m}_1^j \neq nil$ **then**
13: $\quad send(\tilde{m}_1^j)$
14: **end if**

2.3 Message Reordering

Message reordering semantics can be non-trivial in a stream processing system. Typically, streaming computations are stateful, which means that the output of a computation depends on the history of messages. For example, suppose the system computes the moving average price for stock *IBM* between 1:00 PM and 2:00 PM. If one of the messages for this time frame arrives after 2:00 PM due to out-of-order arrival, then the operator will not be able to compute the final value of the result. Thus, message reordering may cause increased delay. The SMBS approach doesn't result in message

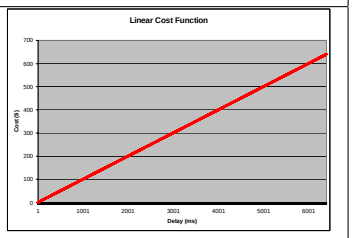
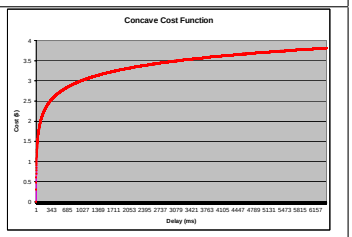
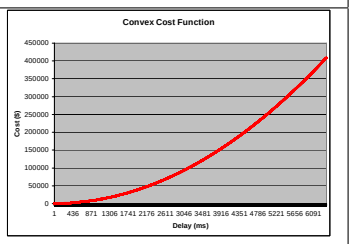
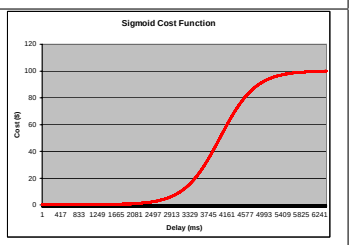
Function Name	General Form	Illustration
Linear	$cost = a * delay + b$	
Concave	$cost = log_b(a * delay)$	
Convex	$cost = delay^a$	
Sigmoid	$cost = \frac{w}{1+e^{(a-\frac{delay}{b})}}$	

Table 2.1: Types of QoS Functions used

reordering, but DQC performs message reordering depending on the QoS function. We introduced a non-message reordering DQC algorithm. However, it is unclear how to combine DQC and DQC-NR within the same system if only some parts of the system are sensitive to message reordering. In the future, the incorporation of message reordering into the feedback control needs to be studied more extensively. Further issues related to message reordering are discussed by Abadi *et al.* [1].

2.4 QoS Functions

In the current analysis, four types of Quality of Service functions (linear, concave, convex and sigmoid) are used. The general function forms are presented in Table 2.1.

All functions represent interesting real-life scenarios. The linear function represents a scenario where the cost of delay is proportional to delay. Therefore, minimizing the cost results in proportional

minimization of the delay for all messages in the stream. The concave function represents the case where only timely messages are valued. The marginal cost of delay diminishes more the longer a message is delayed. The convex function represents streams whose output messages are uniformly valuable. Thus, increasing the delay of any message results in cost blow-up and stiff penalties. The sigmoid function is a continuous equivalent of a step function that represents the case when messages with delays up to x are not penalized. However, if a message passes a certain deadline, the penalty becomes substantial.

2.5 Cost of Average *vs.* Average Cost

In Section 1.3, the SMBS algorithm is described in terms of trying to minimize the cost of average delay. This is different from the objective function introduced in Section 1.4.3, which uses average cost instead of cost of average delay. Depending on the quality of service function, this substitution may result in the optimization algorithm being tricked into considering costs that are different from the true objective. In this section, the issue of cost of average delay *versus* average cost of delay is addressed.

For a linear function, it is always the case that both quantities are equal by the property of expected value:

$$E(cX) = cE(X).$$

Thus, in case of linear quality of service functions, there is no issue with the SMBS algorithm.

For convex functions $delay^a$, the cost of average will be higher than the average cost. This can be easily proven for the case when $a = 2$:

$$V(X) = E(X^2) - E(X)^2 \Rightarrow E(X)^2 = E(X^2) - V(X) \Rightarrow E(X)^2 \geq E(X^2).$$

Moreover, the amount by which cost of average overestimates average cost depends on variance, which is determined by our scheduling algorithm. Unfortunately, we have no good model for estimating the variance produced by our algorithm.

For concave functions $log_b(a * delay)$, the cost of average delay underestimates average cost. This can be intuitively seen from the basic properties of the logarithm:

$$log(a + b) \leq log(a) + log(b) \Rightarrow log(a + b) \leq log(ab)$$

$$log(E(X)) \leq E(log(X)).$$

For concave and convex functions, the issue of under- and over-estimation only comes into play when scheduling several streams whose QoS functions intersect, because the error in estimation

effectively shifts the point of intersection. The point of intersection is important because, near the point of intersection, the change in priorities yields the most effect in terms of the impact on the objective function.

For sigmoid functions the cost of average could either overestimate or underestimate average cost depending on where on the sigmoid QoS curve the average falls. A sigmoid function has the property that the amount of error is bounded by w. However, it also has the most potential for producing erroneous prediction. For example, assume half of the delays yield a cost near zero (in the first flat part of the sigmoid curve) and half of the delays yield a cost near w. Then the average cost could be around $w/2$. On the other hand, the cost of average delay could be around 0, because average delay falls on the "cheap" part of the QoS curve. This case has the potential to "trick" the optimization algorithm into assuming that giving little or no priority to a stream results in virtually no cost, but in reality the average cost could be quite high.

At this point, we only have a few heuristics that can help the optimization algorithm to yield better predictions. In future work, the relation between average cost and cost of average should be explored more rigorously. Meanwhile, we notice that the distribution of delay under SMBS remains exponential-like. In several experiments the variance change was observed to be roughly similar to the change of the mean, *i.e.*, if the mean and variance under share p_1 are x and v, and the mean under share p_2 is $c * x$, then the variance undre share p_2 is $c * v$. This simple heuristic can improve the optimization portion of our SMBS algorithm. We illustrate these approaches in Section 4.8.

2.6 Distribution of Service Times

Assumptions about the distribution of service times are important in our analysis. The dynamic queue control (DQC) algorithm doesn't rely on any particular distribution of service times. However, the static algorithm (SMBS) relies on distribution assumptions for the Markov model used to predict scheduling algorithm behavior and to perform adjustments to mitigate cost of average predictions. Currently, the assumption is that the service and arrival rates are distributed exponentially. For an exponential distribution, the variance is equal to the mean. The alternative distributions could be hyper-exponential or hypo-exponential. The former represents the case of increased variance, while the latter represents the case of reduced variance. In the future, for the SMBS scheme, a new model must be derived to fit these distributions. For the DQC scheme more experimentation must be performed to determine whether variance affects algorithm efficacy.

2.7 Multiple Server Case

In Section 2.1.1 and 2.2, the queue control algorithms that work in a single server environment were defined. Now, a series of extensions to these algorithms for multiple server environments is discussed. The extensions to the SMBS algorithm are covered first, followed by discussion of DQC and DQC-NR in multiple server environments.

2.7.1 SMBS

The straightforward way to extend SMBS scheduling is to extend our genetic algorithm approach from Section 2.1.4. The global solution for the whole network can be represented as a two-dimensional genome of priorities, $\tilde{P}$, where $\tilde{P_{ij}}$ is the priority for stream j on server i. Then, the same structure of the genetic algorithm is used to find a near optimal partition on each server. The evaluation function for this approach looks at end-to-end delay for each stream induced by all local priorities. The genome with the lowest total cost is the fittest. The system also monitors the mean arrival and service rates of each stream on each server and, when these means change, invokes the algorithm to find new partitions.

Algorithm 9 $globalSMBS(N) \rightarrow P$

Require: N : Number of evolution iterations
Ensure: $\forall i$, $sum_{j=1}^{n_i} P_{ij} = 1$
1: $population \leftarrow getInitial()$
2: **for** $i = 0$ to N **do**
3: $\quad population \leftarrow evolve(globalEvaluator)$
4: **end for**
5: $P \leftarrow getFittest(population)$
6: **return** P

Algorithm 10 $globalEvaluator(P, N_{iter}) \rightarrow cost$

1: $cost \leftarrow 0$
2: **for all** $f_j \in F$ **do**
3: $\quad delay \leftarrow 0$
4: $\quad$ **for all** $s_i \in S \wedge \exists k, m(k, j) = i$ **do**
5: $\quad\quad delay \leftarrow delay + predict([P_{i,1}, ..., P_{i,n_i}], N_{iter})$
6: $\quad$ **end for**
7: $\quad cost \leftarrow cost + q_j(delay)$;
8: **end for**
9: **return** $cost$

Since the algorithm is centralized, it has certain limitations. One set of limitations comes from the nature of genetic algorithms; they are slow and have a tendency to converge to sub-optimal solutions. This is not the biggest issue, since the mean rates usually do not change frequently. The hardest part is genome evaluation, since it involves computing the Markov chain approximation for each server in the topology. The approximation is time consuming and, as we will see in the

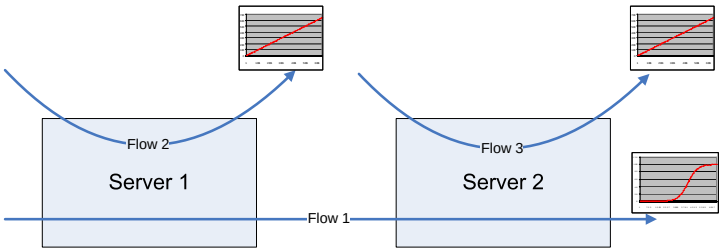


Figure 2.3: Two Server/Three Flow Example

experimental section, cannot effectively be evaluated for more than five streams on each server. Nonetheless, for small networks, this approach may be sufficient and simple enough to use, and for larger networks the presented algorithm can be easily parallelized to provide limited scalability. Alternately, a distributed algorithm could be designed to achieve even better scalability.

2.7.2 DQC

For the dynamic queue control algorithm, the extension to a multiple server environment is complex. The simplest approach is to use DQC and DQC-NR algorithms in a multiple server environment with simple heuristic modifications. A complete control algorithm that yields provable, optimal or near-optimal solutions is yet to be developed.

Consider a simple two server/three stream example where just executing a local DQC algorithm without any modifications fails to achieve optimality. The mapping of streams to servers is presented below (Figure 2.3).

For simplicity, assume that all three flows have the same arrival and service rates. The quality of service function for each flow is depicted in Figure 2.3. Flow 1 has a sigmoid quality of service function $q_i(d) = 10000/(1 + e^{(20-d/110)})$, and flows 2 and 3 have linear quality of service function $q_i(d) = d$. On the first server, our single server algorithm defers scheduling jobs in flow 1 until the point where $\frac{dq_1(t)}{d(t)}$ becomes greater than $\frac{dq_2(t)}{d(t)}$, which happens near the inflection point where the derivative of $q_1(t)$ starts to grow rapidly. Then, on the second server, the delay for messages in flow 1 will be at least equal to service time, which causes the total cost to be 10000. If the local algorithm "knew" that delaying messages for flow 1 would result in ultimately higher cost, it would have scheduled these messages ahead of flow 2's messages, even though the metric introduced by Algorithm 7 tells it otherwise.

Now, we introduce a simple extension to our local algorithm that mitigates some of its deficiencies. The extension to Algorithm 7 is simple. We offset the time $T(m_i^k)$ each message spends in the system

by the expected total service delay. Total service delay is the aggregation of all service times until a local message reaches the user. The new formula for the modified Algorithm 7 computes marginal cost as:

$$marginalCost = \mu_k \left. \frac{dq_k(x)}{dx} \right|_{x=(T(m_i^k)+\omega)}$$

$$where \omega = \sum_{i=0}^{D_k} \mu_{ik} \text{ and } D_k \text{ is the set of downstream servers on which flow } k \text{ is located .}$$

The average service time, μ_{ij}, is tracked on each server for each flow and is fed back upstream to be used for marginal cost computation.

However, this simple change doesn't work well in all circumstances. True service time experienced on servers downstream may greatly differ from the expected service time. This deviation case is impossible to predict in general. In addition, there is queuing delay downstream, which is even harder to estimate than the deviation from the mean. We explore these issues experimentally in Chapter 4.

2.8 Internet Traffic, Flow Intensities and Birth-Death Rates

As mentioned in the introduction, the problem of distributed scheduling and queuing delays is very similar to the rate control problem from the Internet domain. For example, FAST TCP [28] formulates the problem of adjusting the rate of data transmission over the Internet as a control problem and uses well developed control theory tools to prove the stability of a novel TCP algorithm. We are trying to apply a similar approach to streaming systems; these have a more complex structure of data flows than Internet traffic because they involve varying service times, flow splits and joins and quality of service functions that assign cost to timeliness of intermediate results rather than utility based on average throughput.

Nonetheless, the study of Internet traffic provides valuable insights. The flows on the Internet backbone are usually classified into two groups: long-lived, intensive flows ("elephants") and short-lived, light flows ("mice"). As much as eighty percent of total traffic consists of "elephant" flows. Therefore, the analysis in Paganini *et al.* [39] only looks at long-lived processes. This is important because any control algorithm needs time to react to changes in the system; if changes are fast enough due to a large number of short-lived flows, no control algorithm can react appropriately.

Although there are no industrial strength stream processing systems at this time, we conjecture that a similar situation would hold. Thus, our analysis is only concerned with long-lived streaming flows. The streaming computations are continuous and persistent, so it is very likely that most flows will exist for long periods of time. If not, the system design could still maintain our focus on only

long-lived streams, because rapid birth and death of flows would affect the variance of service rates for the long-lived flows. Thus, by adjusting the statistics of long-lived streams, the system could take into account a greater number of short-lived flows.

Chapter 3

Theoretical Results

3.1 Introduction

In this chapter, several theoretical results are introduced. Optimality proofs of the DQC and DQC-NR algorithms are presented. In addition, the advantage of the dynamic process sharing scheme used by the SMBS algorithm over the static process sharing scheme is proved.

3.2 Static Sharing Analysis

We start with a proof that the dynamic process sharing algorithm has an advantage over the classical static process sharing algorithm. First, we define an *edge* (or *boundary*) as a set of MC states such that each state has one or more queues empty. For example, such a a set of states for a two queue Markov chain could be all states with the first queue empty. Second, we define the static process sharing algorithm. Dynamic process sharing was defined in Section 2.1.1. With these two definitions, we can proove that the dynamic process sharing is superior. The core of the proof is to show that, due to edge states, the static scheme would yield only partial processing capacity to a flow when full capacity is available and there is a temporary state of non-contention.

Definition 3.2.1. *Given a continuous Markov process M with state space S, transition rates defined by the dynamic process sharing algorithm from Section 2.1.1, and a set $K = \{1, \ldots, n\}$. The* edge *(or* boundary*), E_K, is a set consisting of all states $\tilde{s_i} = \{s_1, ...s_n\} \in S$ such that $(\forall k : k \in K : s_k = 0)$.*

Definition 3.2.2. *Given a server s with processing capacity $\tilde{C_s}$, a set of flows F on s where $|F| = n$, and a set of priorities $\{p_1, ..., p_n\}$ such that $\sum_{i=1}^{n} p_i = 1$, the* static process sharing scheme *is an algorithm that provisions $p_i\tilde{C_s}$ to flow $f_i \in F$ at all times.*

Theorem 3.2.1. (Dynamic Process Sharing Theorem) *For any server s and set of flows F where*

μ_i is the expected service rate and λ_i is the expected arrival rate for flow $f_i \in F$,

$$\forall f_i : f_i \in F : E(W^d_{f_i}) < E(W^s_{f_i})$$

where $E(W^d_{f_i})$ is the expected wait time for flow f_i under dynamic process sharing and $W^s_{f_i}$ is the expected wait time for flow f_i under static process sharing.

Proof. Let $K = \{1\}$ and $Pr(E_K)$ be the non-zero probability that all queues except for flow f_i are empty. Without loss of generality, under dynamic sharing, in any state $s_i \in E_K$, the service rate for f_i under dynamic sharing is μ_i while the service rate under static sharing is $p_i\mu_i$. Since there exists time when $\mu_i > p_i\mu_i$, $E(q^d_{f_i})$ is less than $E(q^s_{f_i})$ where $E(q^s_{f_i})$ is expected queue size under static sharing and $E(q^d_{f_i})$ is expected queue size under dynamic sharing. Thus, by Little's Law:

$$\forall f_i : f_i \in F : E(W^d_{f_i}) < E(W^s_{f_i}).$$

The proof depends on the fact that $\exists K | Pr(E_K) > 0$. This is very easy to show. Let state $s_0 \in S$ such that $\forall \tilde{s}_j : \tilde{s}_j \in s_0 : \tilde{s}_j = 0$. State s_0 corresponds to the case when all queues are empty and no messages are being processed. The steady state probability for this state is $1 - \rho$.

$$Pr(s_0) = 1 - \sum_{\forall f_i in F} \lambda_{f_i}/\mu_{f_i} > 0$$

Without loss of generality, we also know that the transition probability of moving from state s_0 to state $s_1 = [\tilde{s}_0, ..., \tilde{s}_n] \in S$ such that s_1 is a boundary state is

$$Pr(s_0 \rightarrow s_1) = \lambda_{f_i} / \sum_{\forall f_i in F} \lambda_{f_i} > 0.$$

Therefore, $Pr(s_1) > 0$, which shows that there exists K, *i.e.*, $K = \{j\}$, for which $Pr(E_K) > 0$. □

3.3 Optimality of DQC in a Single Server Environment

Now, we prove that the dynamic queue control algorithm minimizes average cost when executed in a single server environment. In this proof, the expected cost of making incremental decisions at each Δt is determined. Then, the expression that minimizes this expected cost is derived and is shown to be the same as the one used by Algorithm 7.

Theorem 3.3.1. (DQC Optimality Theorem) *Given server s and a set of flows F, where each flow has a quality of service function q_i that maps message delay to a measure of cost, the scheduling*

algorithm DQC minimizes the total cost function

$$\frac{1}{|F|} \lim_{t \to \infty} \frac{\sum_{i=0}^{|F|} \sum_{t=0}^{N_t^i} c(d_t^i)}{N_t^i}$$

where N_t^i is the number of events received by stream i before time t.

Proof. After Δt of time, the current job being executed is either finished or needs more time to finish. The probability that the job finishes in Δt for stream i is $\mu_i \Delta t$. Let M be a set of all unfinished jobs at s; then the total cost incurred is the delay for all unprocessed jobs in the queue:

$$\forall m_i^k : m_i^k \in M : Cost = \sum_{i=1}^{N} (\frac{dq_k(x)}{dx}\Big|_{x=T(m_i^k)} \Delta t) - \frac{dq_k(x)}{dx}\Big|_{x=T(m_i^k)}.$$

$Cost$ is expressed as the summation of the costs for all the jobs minus the cost of the job that actually finishes.

The other case is that the job being executed does not finish. This will happen with probability $1 - \mu_i \Delta t$. In this case, the total cost of the decision will be

$$\forall m_i^k : m_i^k \in M : Cost = \sum_{i=1}^{N} (\frac{dq_k(x)}{dx}\Big|_{x=T(m_i^k)} \Delta t).$$

Thus, the expected cost of processing a job for Δt of time is

$$E(Cost) = (\mu_i \Delta t) \sum_{i=1}^{N} (\frac{dq_k(x)}{dx}\Big|_{x=T(m_i^k)} \Delta t) - \frac{dq_k(x)}{dx}\Big|_{x=T(m_i^k)} + (1 - \mu_i \Delta t) \sum_{i=1}^{N} (\frac{dq_k(x)}{dx}\Big|_{x=T(m_i^k)} \Delta t)$$

$\equiv$

$$E(Cost) = \sum_{i=1}^{N} (\frac{dq_k(x)}{dx}\Big|_{x=T(m_i^k)} - \mu_i \frac{dq_k(x)}{dx}\Big|_{x=T(m_i^k)} \Delta t^2$$

Therefore, in order to minimize expected cost, the algorithm DQC has to maximize

$$\mu_i \frac{dq_k(x)}{dx}\Big|_{x=T(m_i^k)}$$

□

3.4 Scheduling Policies Induced by QoS Functions

A very interesting set of properties of the DQC scheduling algorithm involves its behavior for different types of Quality of Service functions. Under a given type of quality of service function (*i.e.*, linear,

convex, concave or sigmoid), DQC degenerates into a well-known scheduling discipline such as first-come, first-serve (FCFS), last-come, first-served (LCFS) or a combination of these two.

Theorem 3.4.1. (QoS-induced Scheduling Disciplines) *Given a set of jobs $J = \{j_1, ..., j_n\}$, a quality of service function q, and a scheduling algorithm DQC:*

- *DQC degenerates to FCFS if $\frac{d^2q(x)}{dx} \geq 0$*
- *DQC degenerates to LCFS if $\frac{d^2q(x)}{dx} < 0$*

Proof. Without loss of generality, assume jobs $j_1, j_2 \in J$ have arrival times t_1 and t_2, where $t_1 < t_2$. For the purposes of this proof, we assume that if $t_1 < t_2$, then the arrival time for j_1 is less than for j_2. Moreover, let w be time the job j_1 has spent waiting. Then $w - (t_2 - t_1)$ is the amount of work needed for j_2. If $\frac{d^2q(x)}{dx} \geq 0$, then

$$\text{as } w \to \infty \text{ , } (\mu_i \left.\frac{dq(x)}{dx}\right|_{x=w-(t_2-t_1)}) < (\mu_i \left.\frac{dq(x)}{dx}\right|_{x=w}).$$

Thus j_1 will be served before j_2, which corresponds to FCFS discipline, because $\left.\frac{dq(x)}{dx}\right|_{x=w} \to \infty$ as $w \to \infty$.

If $\frac{d^2q(x)}{dx} < 0$, then

$$\text{as } w \to \infty \text{ , } (\mu_i \left.\frac{dq(x)}{dx}\right|_{x=w-(t_2-t_1)}) > (\mu_i \left.\frac{dq(x)}{dx}\right|_{x=w}).$$

Thus j_2 will be served before j_1, which corresponds to LCFS discipline, because $\left.\frac{dq(x)}{dx}\right|_{x=w} \to 0$ as $w \to \infty$. □

Using Theorem 3.4.1, we can conclude that under linear and convex functions DQC degenerates to FCFS, and under a concave function it degenerates to LCFS. Under a sigmoid function, the messages are served FCFS until the total time in the system reaches the inflection point at $\frac{d^2q(x)}{dx} = 0$, and then the messages are served LCFS.

3.5 Optimality of DQC-NR

Now, we show that DQC-NR also achieves optimality if message reordering is not allowed. Generally, the structure of the proof is the same as for DQC. However, an assumption is made that the service time for messages residing in the queue is not known. Expected service time is assumed to be a "good enough" metric of true service time.

Theorem 3.5.1. (DQC-NR Optimality Theorem) *Given a server s, a set of flows F where each flow has a quality of service function q_i that maps message delay to a measure of cost, and a set of*

queues $\tilde{Q}$ *where* $\tilde{q}_i \in \tilde{Q}$ *corresponds to flow* $f_i \in F$*, then scheduling algorithm DQC-NR minimizes the total cost function*

$$\frac{1}{|F|} \lim_{t \to \infty} \frac{\sum_{i=0}^{|F|} \sum_{t=0}^{N_t^i} c(d_t^i)}{N_t^i},$$

where N_t^i *is the number of events received by stream* i *before time* t*,*

under constraint (message reordering prohibited)

$$G = \forall i \in [0, |F|], T(m_k^i) < T(m_n^i) \Rightarrow T_c(m_k^i) < T_c(m_n^i),$$

where $T_c(m_k^i)$ *is the completion time of message* m_k^i*.*

Proof. Similar to Theorem 3.3.1, we define the marginal cost $m_c(t)$ of delaying a message m_k^i for Δt of time to be

$$m_c(t') = \left. \frac{dq_i(x)}{dx} \right|_{x=t'} \Delta t,$$

where, as in Theorem 3.3.1, $t' = T(m_1^i)$. Define a function $\delta(s, i)$ as

$$\delta(s, i) = \sum_{j=s}^{n_i} m_c(T(m_j^i) + j{\mu_i}^{-}1),$$

where i is an index of the i^{th} flow, n_i is the total number of messages in $\tilde{q}_i$ and s is the s^{th} message from the head in queue $\tilde{q}_i$. We note that $\delta(1, i)$ defines the total marginal cost for queue flow f_i and $\delta(2, i)$ defines the total marginal cost for queue flow f_i if the first message completes after Δt of time. When we select a message m_1^i at the head of the queue $\tilde{q}_i$ for processing for the next Δt it will finish with probability $\mu_i \Delta t$ and will remain unfinished with probability $(1 - \mu_i \Delta t)$. If the message doesn't finish then the total marginal cost for all the queues on this server is

$$\delta_t(i) = \sum_{j=1}^{n} \delta(1, j).$$

However, if the message finishes, the cost is

$$\delta_t(i) = \sum_{j=1}^{n} \delta(1, j) - \gamma - \phi, \text{ where}$$

$$\gamma = m_c(T(m_1^i)) \text{ and}$$

$$\phi = \delta(1, i) - \delta(2, i).$$

We note that the cost if the message finishes is different because the message at the head of the queue is removed and the queue becomes shorter, changing the marginal cost for all messages in the

queue.

Thus, the expected cost of selecting message m_1^i for processing in the next Δt is

$$E(C_i) = (1 - \mu_i \Delta t)(\sum_{j=1}^{n} \delta(1,j)) + (\mu_i \Delta t)(\sum_{j=1}^{n} \delta(1,j) - \gamma - \phi)$$

$\equiv$

$$E(C_i) = \sum_{j=1}^{n} \delta(1,j) - \mu_i \Delta t \sum_{j=1}^{n} \delta(1,j) + \mu_i \Delta t \sum_{j=1}^{n} \delta(1,j) - \mu_i \Delta t(\gamma + \phi)$$

$\equiv$

$$E(C_i) = \sum_{j=1}^{n} \delta(1,j) - \mu_i \Delta t(\gamma + \phi).$$

So, to minimize the expected cost, we need to maximize $\mu_i \Delta t(\gamma + \phi)$. This is what the DQC-NR algorithm does (see Algorithm 8). □

Theorem 3.5.1 shows that Algorithm 8 achieves optimality in the single server environment.

3.6 Bounds on DQC-NR

If message reordering is not allowed, it is interesting to consider how much worse the performance is than when message reordering is permitted. Now, we show the best case and worst case bounds on DQC-NR as compared to DQC. This should provide an intuition about how much is lost by prohibiting message reordering.

Theorem 3.6.1. (Bounds on DQC-NR) *Given a server s, a set of flows F with quality of service functions $q_i : R \to R$, and algorithms DQC and DQC-NR. Let G_{DQC} be the optimum achieved under DQC and G_{DQC-NR} be the optimum achieved under DQC-NR. On average,*

$$0 \leq G_{DQC-NR} - G_{DQC} \leq q_i(\sum_{i=1}^{|F|} \frac{\overline{q}_i}{\lambda_i}),$$

where $\overline{q}_i$ is the expected queue size for flow i.

Proof. The lower bound can be easily derived from Theorem 3.4.1; given a convex quality of service function, DQC degenerates into FCFS scheduling, which in fact corresponds to DQC-NR. The upper bound can also be derived by using Theorem 3.4.1. We note that, in the worst case, DQC-NR performs FCFS while DQC degenerates into LCFS. Therefore, the message that arrives at the end of queue $\tilde{q}_i$ may need to wait until all messages in all queues that arrived before it are processed.

Thus, on average the wait time is

$$\overline{W} = \sum_{i=1}^{|F|} \frac{\overline{q}_i}{\lambda_i}.$$

The additional cost incurred is therefore $q_i(\overline{W})$ for some flow f_i. □

As Theorem 3.6.1 shows, under some conditions DQC-NR can degenerate into DQC. However, in the case when DQC induces LCFS scheduling, DQC-NR can perform much worse than DQC.

Chapter 4

Experimental Results: Single Server

4.1 Experimental Settings and Ptolemy

This section describes the experimental environment used in our work. All code for the experiments is developed in Java. Whenever experiments are aimed at describing performance of an algorithm, these experiments are performed on IBM ThinkPad T43p with 2 GHz CPU, 1 GB RAM and Sun's JDK $1.4.2_06 unless otherwise noted.$

Multi-machine, wide-area experiments are simulated using the Ptolemy discrete event simulator [9] with custom extensions. A discrete event simulation consists of actors that exchange information using messages on a virtual discretized time scale. Ptolemy allows easy addition of custom actors to the simulation engine. In this case, custom actors include servers that implement the described scheduling and control policies (SMBS and DQC), channels that introduce optional delay on message transfer, cost calculators that compute the cost of ultimate delay using provided QoS functions, and data sources that generate messages whose payloads are used to compute simulated service times on each server. Message payloads and interarrival times are distributed exponentially.

An example of a Ptolemy model is shown below for the two streams/ one server example. The links going from data sources to the server and then to cost calculators carry data messages. The links going in the opposite direction carry feedback messages (see Figure 4.1).

We use the simulation as an accurate representation of real-life conditions and utilize simulation results as a baseline for evaluation of our predictive and optimization algorithms.

4.2 Need for Non-trivial Scheduling

We begin by illustrating the need for smart scheduling. The need for non-trivial scheduling can be demonstrated by performing comparison of our scheduling approaches to a naïve scheduling policy.

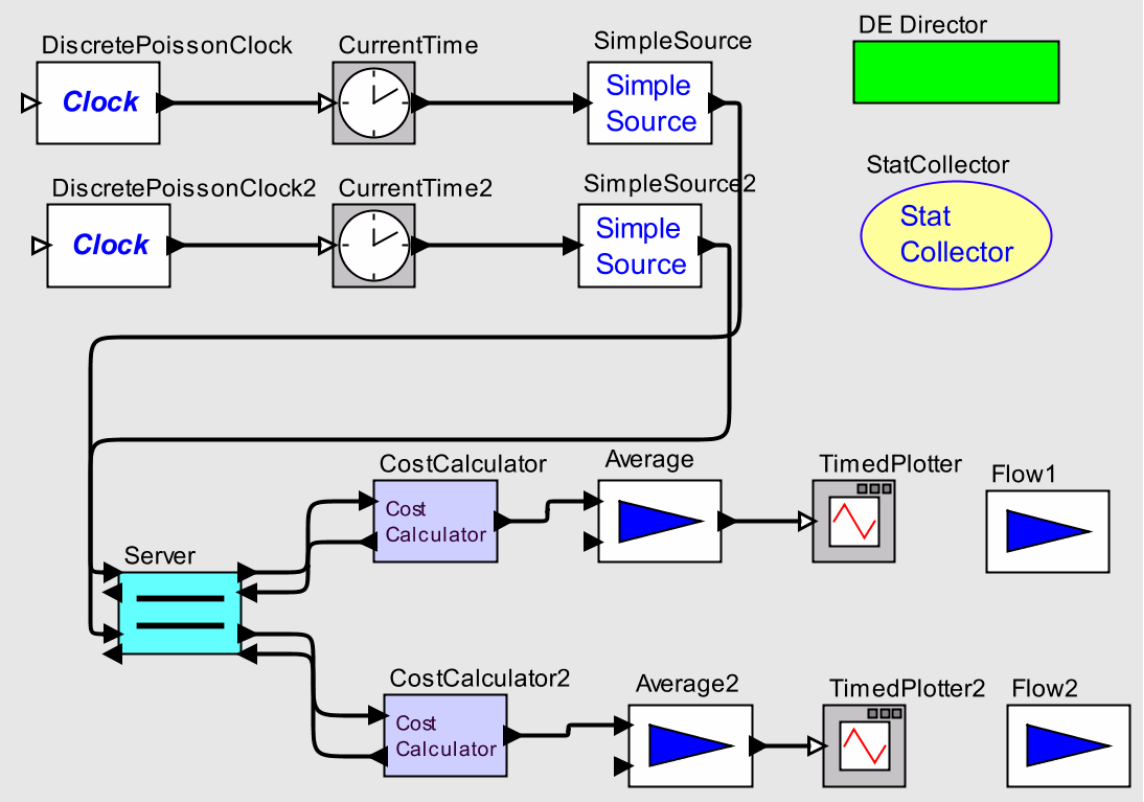


Figure 4.1: Example of Ptolemy Simulation Model

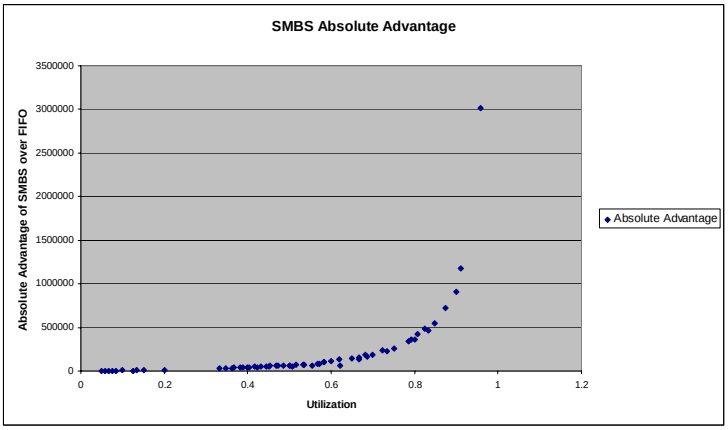


Figure 4.2: SMBS vs FIFO on Two Streams with Linear QoS (Absolute Difference)

Parameters	**Descriptions**
Algorithms	SMBS, DQC and DQC-NR
Number of Streams	2
Stream 1 expected interarrival times	Between 160 and 4000
Stream 2 expected interarrival times	Between 160 and 4000
Stream 1 expected service times	100
Stream 2 expected service times	100
Stream 1 QoS	$cost = delay$
Stream 2 QoS	$cost = 2000 * delay$

Table 4.1: Experiment Parameters

As an example of of a naïve scheduling policy, a simple FIFO algorithm is chosen. The detailed comparison of SMBS, DQC and DQC-NR against FIFO is performed in Section 4.8.

To show the advantages of smart scheduling, the algorithms are run against FIFO at different utilization levels. The parameters for each experiment are listed in Table 4.1. For each algorithm, average cost under linear QoS is generated. The second stream's QoS function weighs delay at 2000 times that of the first stream. Different utilizations are achieved by changing arrival rates for the streams. The average cost is compared to the average cost under FIFO scheduling. The results for all three algorithms are very similar because, under a linear QoS function, all three algorithms degenerate into similar scheduling policies.

All algorithms—SMBS, DQC, and DQC-NR—outperform FIFO. The absolute and percentage differences between our scheduling algorithm and FIFO are depicted in Figures 4.2–4.7. The absolute difference grows exponentially after utilization of 60%, which corresponds to the point at which the queue size begins to grow exponentially. The percentage difference grows approximately quadratically, which is also due to the fact that the larger the queues, the more effect the smart scheduling has.

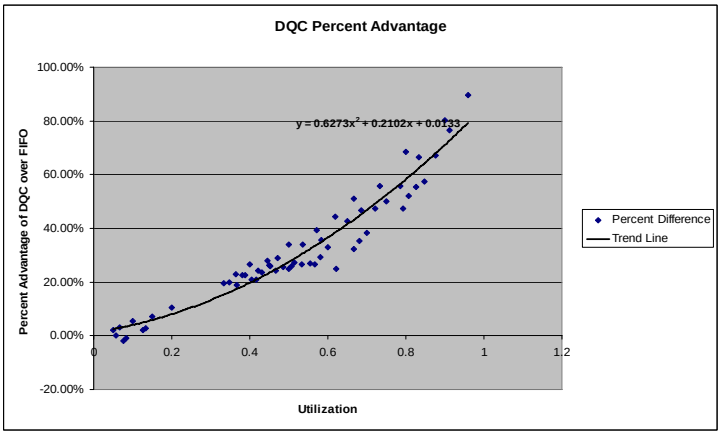


Figure 4.3: DQC vs. FIFO on Two Streams with Linear QoS (Percent Difference)

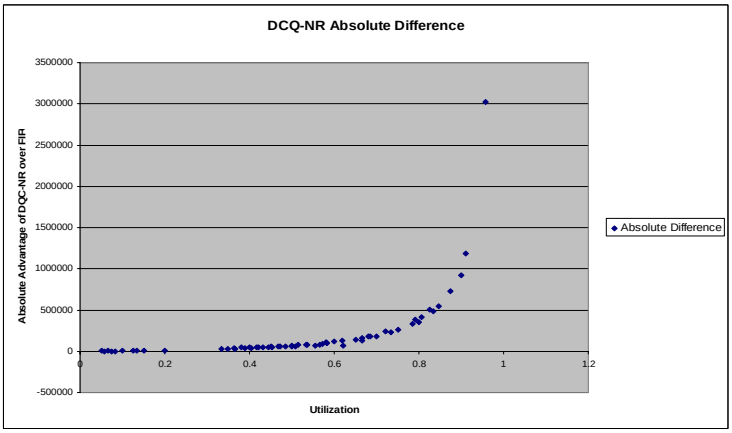


Figure 4.4: DQC-NR vs. FIFO on Two Streams with Linear QoS (Absolute Difference)

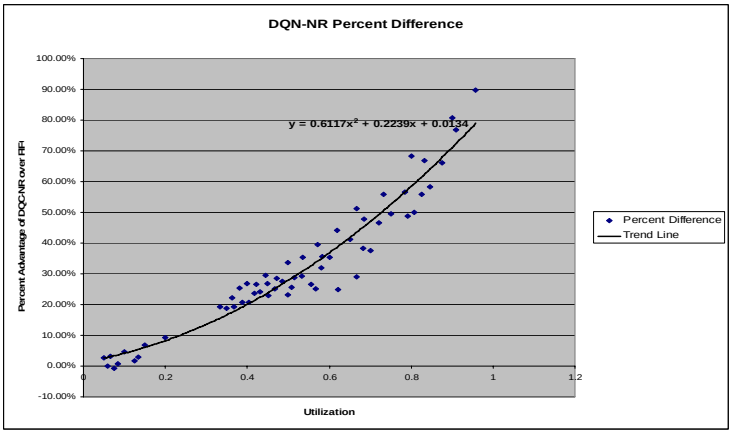


Figure 4.5: DQC-NR vs. FIFO on Two Streams with Linear QoS (Percent Difference)

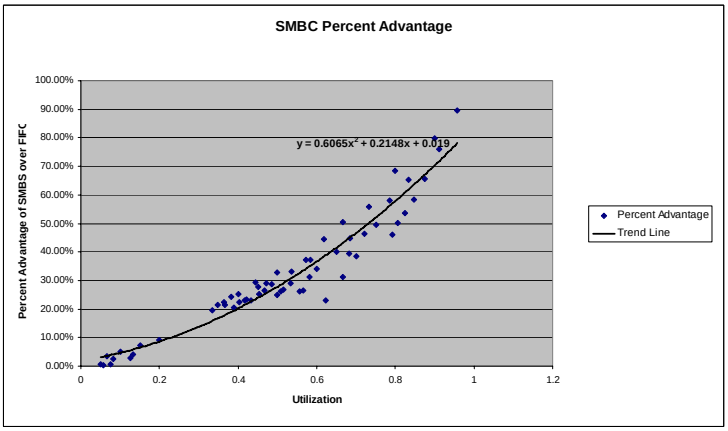


Figure 4.6: SMBS vs. FIFO on Two Streams with Linear QoS (Percent Difference)

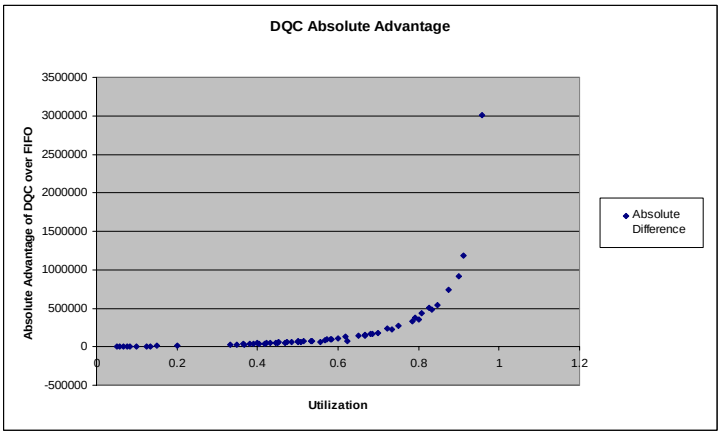


Figure 4.7: DQC vs. FIFO on Two Streams with Linear QoS (Absolute Difference)

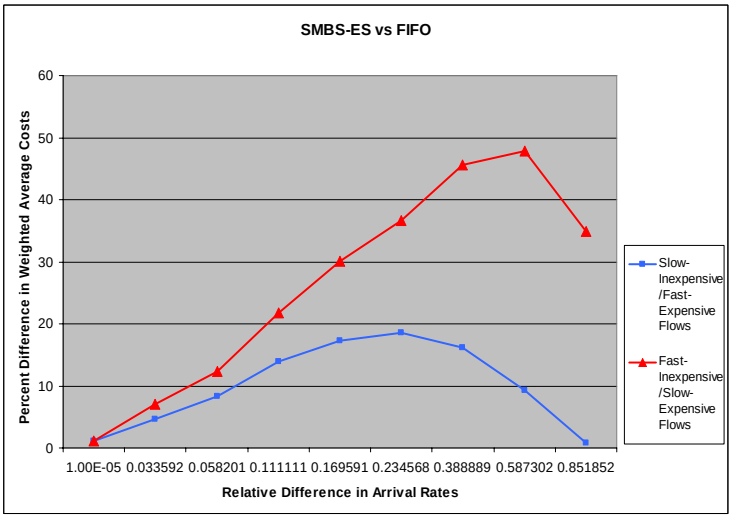


Figure 4.8: SMBS-ES *vs.* FIFO

In the previous set of experiments, the scheduling algorithms were compared to FIFO. An interesting, related question arises from our comparison. Is FIFO equivalent to the fifty-fifty dynamic process sharing scheme? This question is important for further evaluation of our algorithms. If SMBS with equal shares (SMBS-ES) degenerates to FIFO, it could be used as an alternative baseline for further experiments. To test this hypothesis, FIFO is run against SMBS-ES in a one server, two flows environment with one flow's arrival rate made progressively faster while maintaining the overall utilization at 90% for all runs. Then, the percentage difference in stream rates is plotted against the difference in weighted average cost between FIFO and SMBS-ES. This set of experiments is repeated twice, making the fast stream 10 times more expensive than the slow stream and vice versa. The results are depicted in Figure 4.8.

In Figure 4.8, it can be seen that the difference between FIFO and SMBS-ES grows as the arrival rates diverge. However, if the streams have equal parameters (*i.e.* arrival and service rates), SMBS-ES does very closely mimic FIFO behavior.

From experiments on SMBS-ES it can be seen that, aside from utilization, the difference in streams' rates also impacts the efficacy of the smart scheduling algorithms. When there are two streams with one low-cost stream having rare, computationally non-intensive events and the other high-cost stream having frequent, computationally intensive events, changing priorities to prefer the second stream does not change the weighted average cost greatly. In Figures 4.9 and 4.10, the decrease in advantage of SMBS over SMBS-ES and FIFO is shown. The utilization for all the experiments is kept at 90%. The difference is plotted against relative streams' rate difference, which is computed as $\frac{\lambda_1 - \lambda_2}{\lambda_1}$. The rest of the parameters are kept the same as in Table 4.1.

In summary, our scheduling algorithms produce significant (up to twofold) improvements over

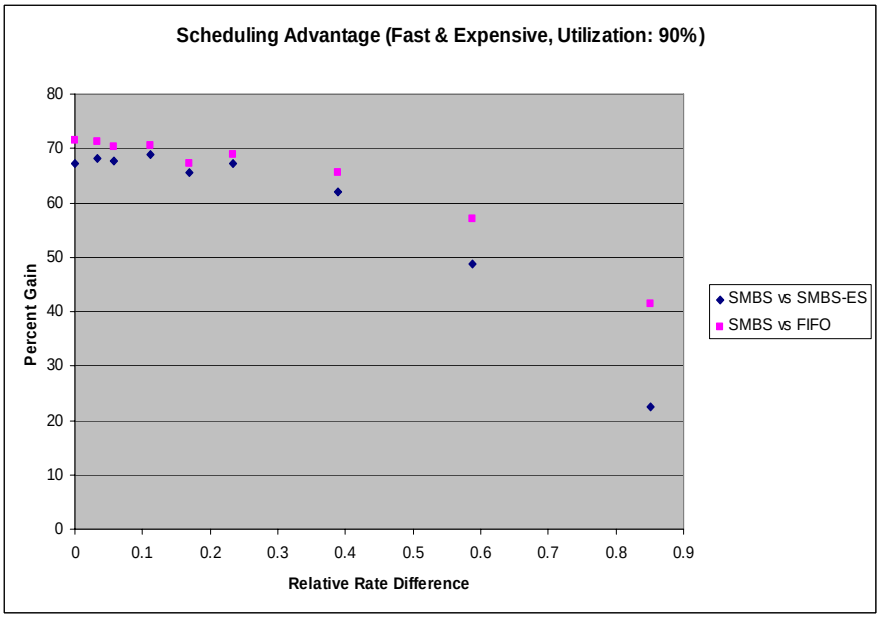


Figure 4.9: Fast Expensive Stream/Slow Inexpensive Stream Advantage Reduction

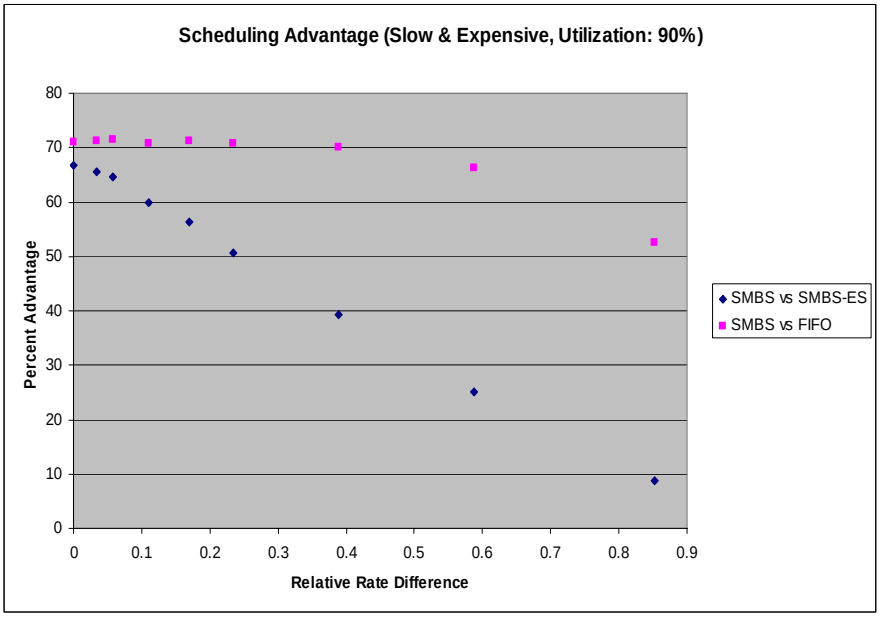


Figure 4.10: Slow and Expensive Stream/Fast Inexpensive Stream Advantage Reduction

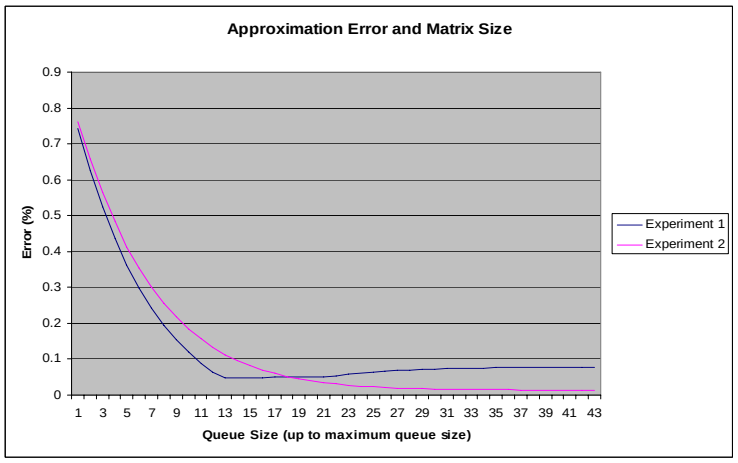


Figure 4.11: Example of Ptolemy Simulation Model

the naïve scheduling algorithm even using simple linear QoS functions. Moreover, the most important fact that affects the amount of improvement is system utilization. On the other hand, the improvement decreases as the difference in rates between the streams grows.

4.3 Queue Sizes and Prediction Accuracy

We start with a detailed experimental analysis of the SMBS scheduling algorithm. The experiments on the SMBS algorithm are split into a series of steps. First, in this section, the amount of error introduced by substituting the finite Markov Chain for an infinite Markov chain is analyzed. Second, the run-time and memory advantages of the fast approximation used by SMBS are shown. Third, the accuracy of the fast approximation is determined. Finally, a comparison among DQC, DQC-NR, SMBS and FIFO is illustrated.

Since our model is based on treating an infinite Markov chain as finite, the first key factor that is important to our approximation is the relationship between the size of the finite chain and the accuracy of the approximation. In other words, we want to know how much accuracy is sacrificed when the model size is reduced by assuming that queues do not exceed certain thresholds. The two sets of experiments below answer this question.

In the above experiments, two flows located on one server are parameterized as stated in Table 4.2. The approximation is performed assuming a certain limit on queue size for these two flows. With each experiment iteration, the limit is increased. One set of experiments is performed on an equal pair of streams. The other has high and low frequency streams. The approximation error is plotted in relation to the maximum queue size assumed. Both sets of experiments show that

	Experiment 1		Experiment 2	
	Steam 1	**Steam 2**	**Steam 1**	**Steam 2**
Mean inter-arrival time	250	250	190	250
Mean service time	100	100	100	70
Priority	1	10	4	7
Maximum queue size (from simulation)	44	11	33	13

Table 4.2: Experiment Parameters

State Size	**Number of Queues**
≤ 2500	2
≤ 15625	3
≤ 160000	4
≤ 400000	5

Table 4.3: Model Size - Number of Queues Correspondence

approximation becomes accurate quite quickly and, after a queue size of 12 (half of the maximum queue size), the error is within 10%.

4.4 Fast MC Model Approximation Performance Evaluation

Now, we examine the performance of our fast model approximation and compare it to the classical LU decomposition approach on non-sparse matrices and the Conjugate Gradients method (CGS) in conjunction with the Quasi-Minimal Residual method (QMR) on sparse matrices [10, 23]. The experiments are run in the following settings. For LU decomposition, the JAMA matrix package is used [21]. For sparse matrices, a sparse matrix toolkit (SMT) is used [25]. The reason that CGS is used in combination with QMR is that CGS sometimes may not converge to a solution. In this case, QMR is used as a fallback solver. In our experience, such a fallback is rarely needed, while using CGS generally results in faster execution.

For each experiment, the model is specified as a set of flows with maximum queue sizes, expected service rates, expected arrival rates and relative priorities. The number of states reported in the figure corresponds to the number of states in the Markov chain generated from a model given its specification. Table 4.3 relates the number of states in a chain to number of queues in a model. For each model, the experiment is repeated several times (the standard deviation never exceeds 4% of the mean).

Note: The size of Q is the number of states squared. For example, a 387072 state model results in a 1.49825E+11 cell Q

The experimental results are presented in Figure 4.12. It can be clearly seen that our fast algorithm has the same exponential complexity as CGS/QSR and LU decomposition. However, the

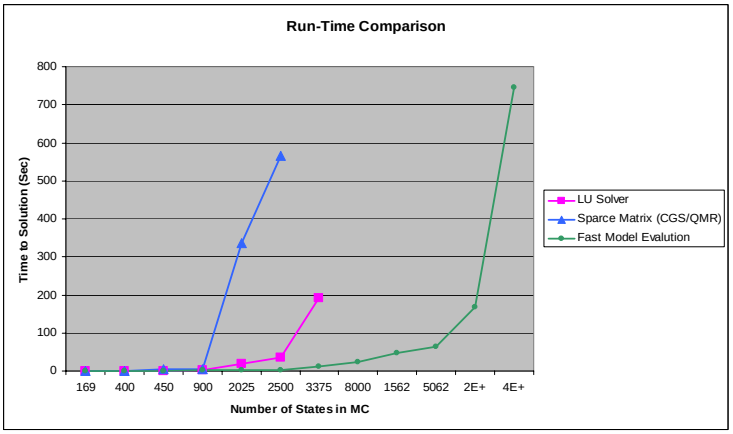


Figure 4.12: Run-Time Comparison

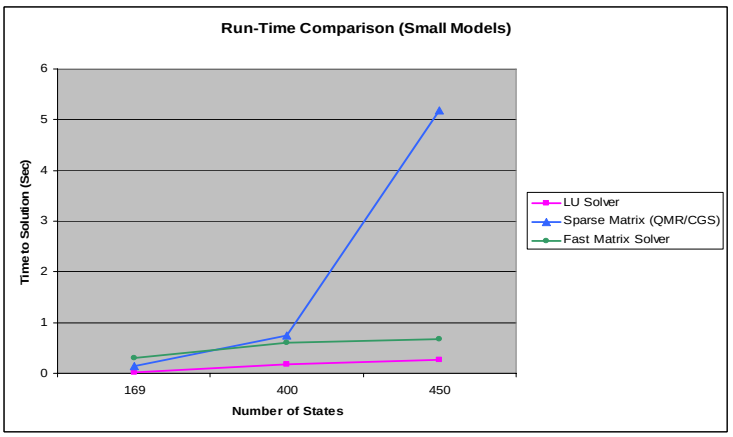


Figure 4.13: Run-Time Comparison (Small Models)

exponential blowup occurs in much larger models. This fact lets our fast approximation process models defined over 5 queues. In addition, our fast model approximation can be easily parallelized, which would increase its ability to process bigger models (the question of parallelization is deferred to future studies).

An interesting point to be noted is that, for small models, the LU decomposition solver outperforms the CGS/QMR and fast matrix approximation algorithms (see Figure 4.13). This is not surprising, since both algorithms are iteration-based and, on small matrices, LU decomposition performs better than iteration-based algorithms. However, the maximum size of matrices for which this is the case is very small, less than 200 states (two queues of maximum 14 messages each).

In terms of memory used by these three algorithms, the picture is very similar to that for execution time (see Figure 4.14). The fast matrix approximation algorithm doesn't eliminate exponential blowup in memory used, but it "postpones" the increases until the model size reaches a large

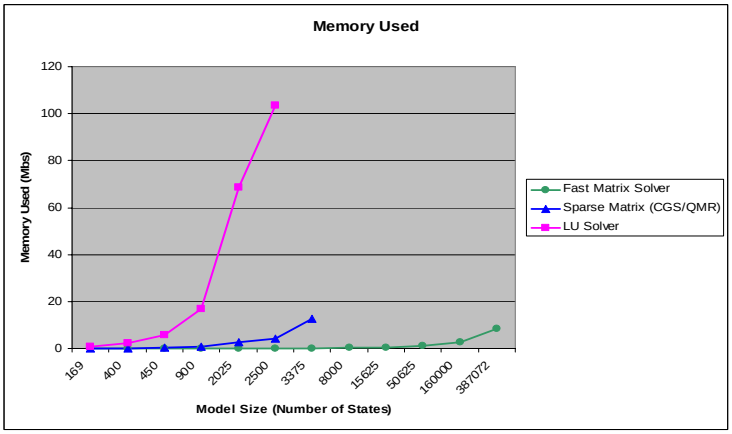


Figure 4.14: Memory Comparison

number of states. It is not surprising that the sparse matrix approach performs in between LU decomposition and fast approximation, because it still represents Q explicitly (although in a sparse manner). Moreover, CQS and QSR require some intermediate matrices that increase their memory usage.

In summary, the fast approximation algorithm drastically reduces memory usage and speeds up calculation of steady-state probabilities for large Markov chain models. With the introduction of multi-core processors, by parallelizing fast approximation, the performance could be increased even further. In the next section, the issue of accuracy of the fast approximation algorithm is discussed.

4.5 Accuracy of Fast Approximation Algorithm

We explore how the accuracy of our approximation method depends on various parameters, including the number of iterations, amount of diffusion in the initial vector π_0, difference between streams' rates, round-off error and number of streams.

We start by looking at the number of iterations. For the purpose of studying the impact of number of iterations, the experiments are run at 90% utilization with two, three and four streams. For each experiment, the number of iterations is increased and the approximation result is compared to the simulation result to determine approximation error. The results are shown in Figure 4.15. Not surprisingly, the approximation becomes more accurate as the number of iterations increases. Also, it can be seen in Figure 4.15 that the more streams there are, the longer it takes to converge to steady-state. The good part is that the difference in number of iterations between two streams and three streams, and between three streams and four streams, is only two-fold. Thus, the number of iterations does not grow exponentially with the number of streams.

The approximation results also exhibit oscillation in the convergence process. In Figures 4.16

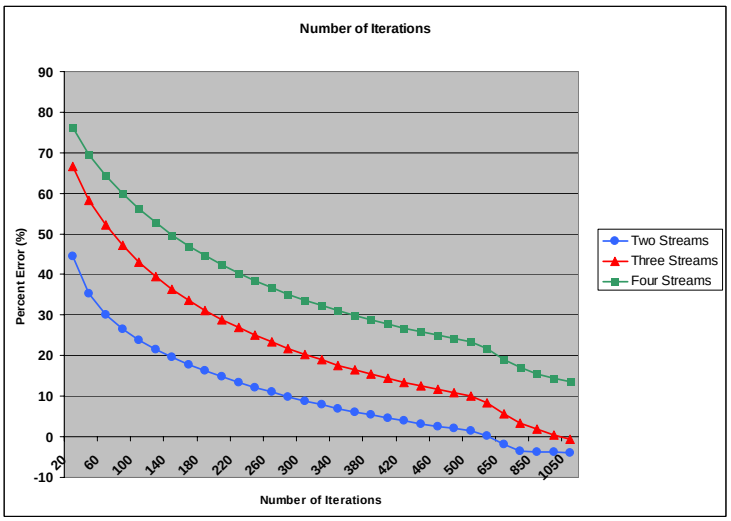


Figure 4.15: Number of Iterations and Accuracy of Fast Approximation

and 4.17, the convergence of the two stream approximation is depicted and we can see that the approximation error fluctuates at each iteration.

Another parameter that affects accuracy is precision of the sparse vector, π. The precision is a number, ϵ, such that for all $\pi_i < \epsilon$, π_i is treated as 0. The greater the ϵ, the less memory is needed to store π and the greater the approximation error is. To test the impact of precision, the number of iterations is fixed while the precision is slowly increased. In Figure 4.18 it can be seen that approximation error decreases as precision increases. Moreover, it is not surprising that the four stream approximation requires higher precision, because of greater probability dispersion.

Figure 4.19 shows the memory savings resulting from decreased precision. By allowing 10% error in the approximation, approximately 50% saving in space is achieved. This rule quantifies the benefit of rounding.

Now, we examine the impact of differences in stream rates on approximation error. The same set of experiments on two streams as before is repeated. This time, the arrival rate is varied such that the total utilization is kept constant at 90%. Figure 4.20 depicts approximation error in relation to the difference in arrival rates. The red line shows the case when the stream with the faster arrival rate is ten times more expensive. The yellow line shows the reverse case. As Figure 4.20 shows, there is no relation between rate difference and approximation error. The error oscillation results from variance in simulation results. To prove this, for each experiment, the expected queue sizes from approximation and simulation are shown. Figure 4.21 shows simulation results to have much more variance than the predictions.

The final parameter studied is the quality of initial condition for the approximation. The algorithm for setting the initial probabilities is described in Section 2.1.3. For each set of states whose

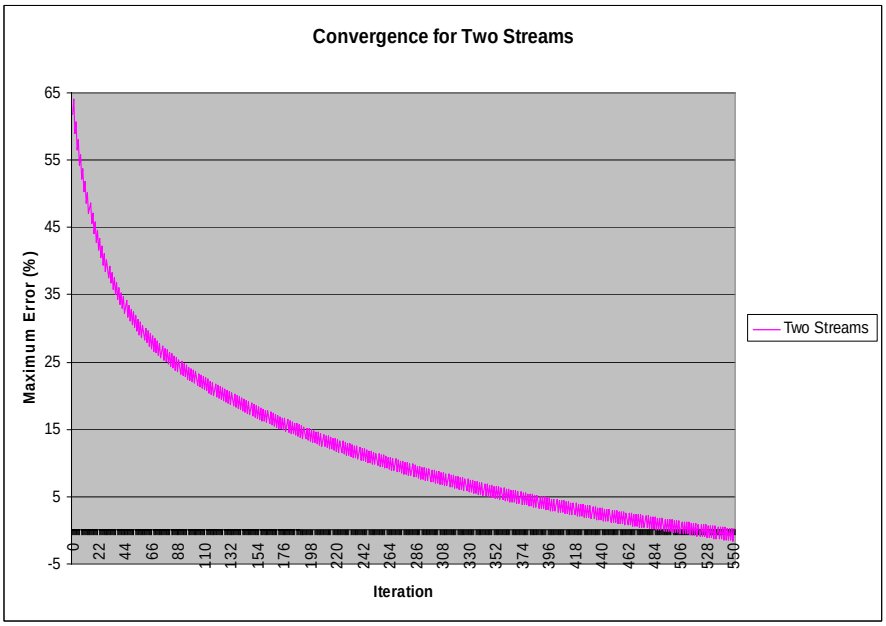


Figure 4.16: Rate of Convergence

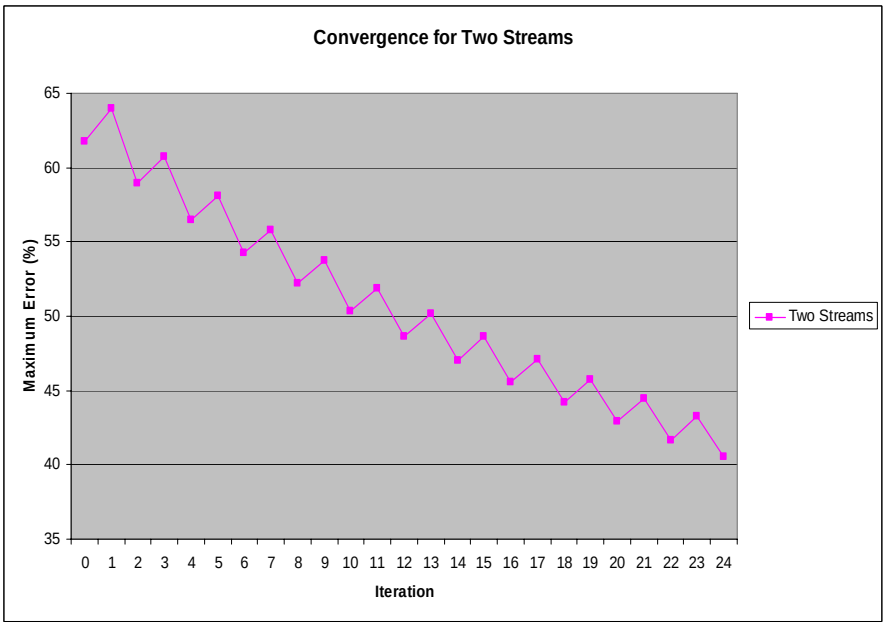


Figure 4.17: Rate of Convergence (Detailed)

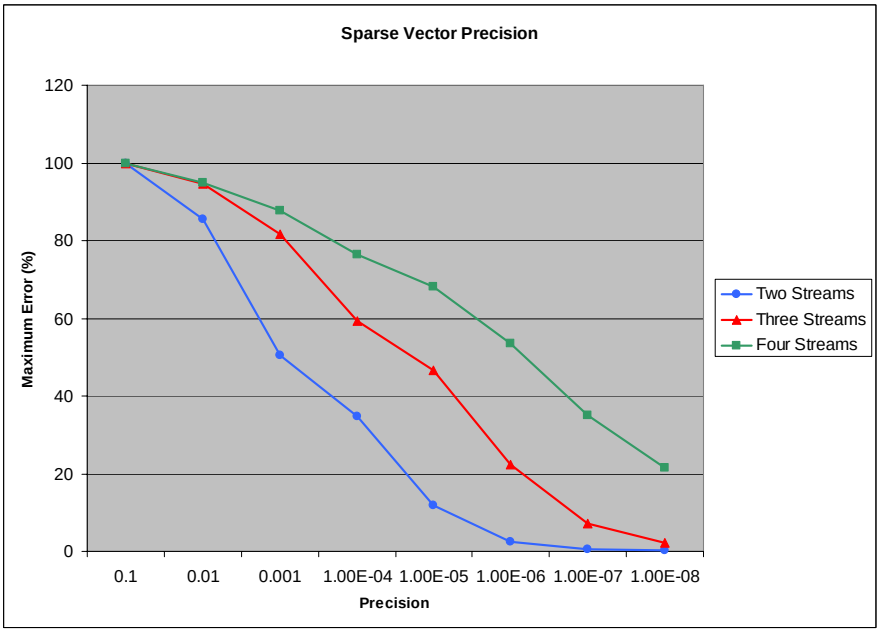


Figure 4.18: Impact of Sparse Vector Precision on Approximation Error

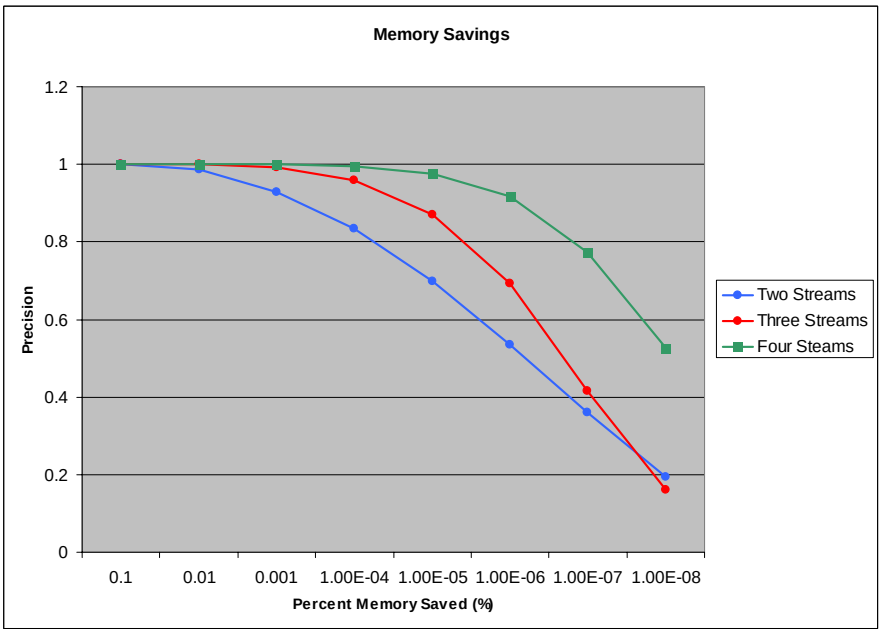


Figure 4.19: Impact of Sparse Vector Precision on Memory Savings

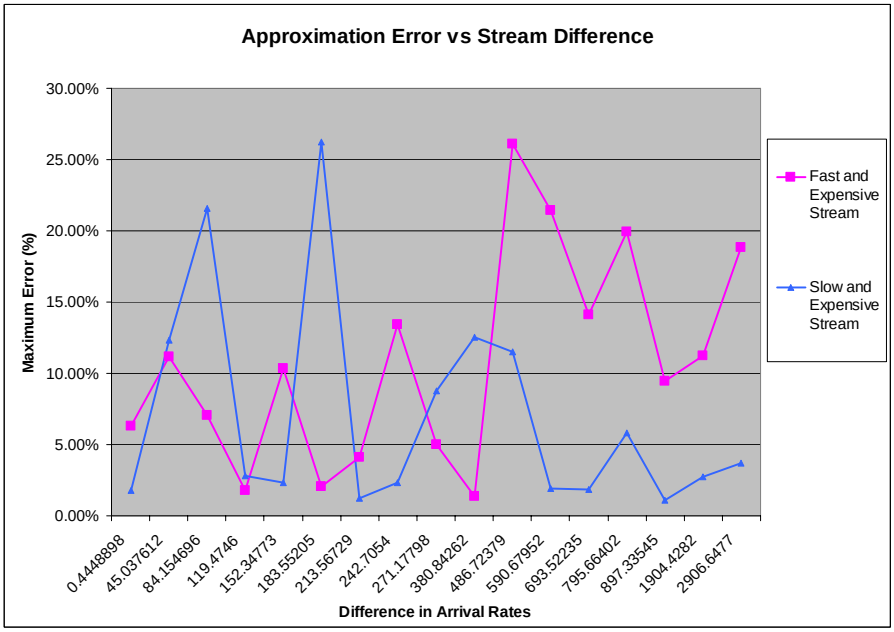


Figure 4.20: Impact of Stream Difference on Approximation Error

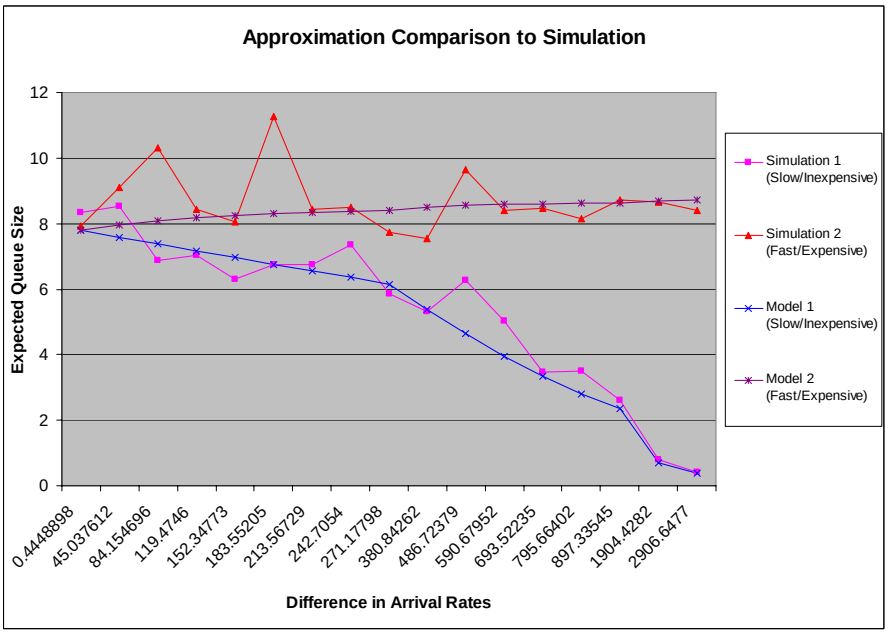


Figure 4.21: Comparison to simulation

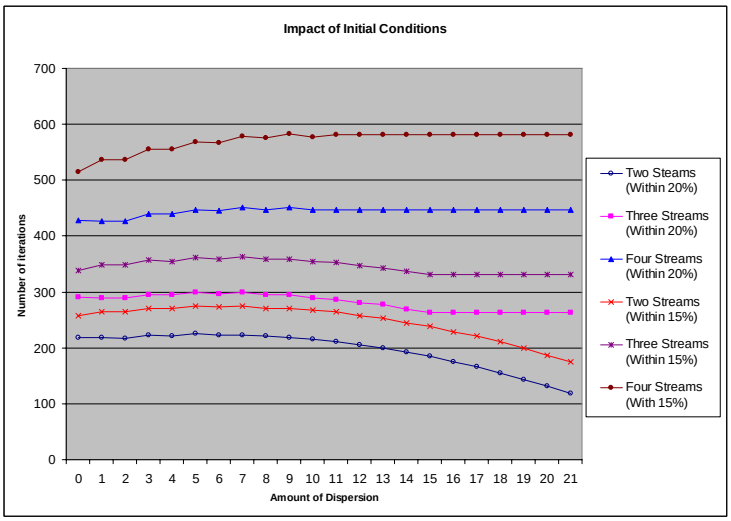


Figure 4.22: Impact of Initial Conditions on Approximation Error

total queue size is n, the probability from the M/M/1 model is computed. The result is divided equally among all states in the set. For the experiments here, the maximum n for which to perform the initialization is picked. The results are presented in Figure 4.22. The graph shows how the greater amount of dispersion (*i.e.*, greater n) impacts the number of iterations needed to obtain an approximation within 20% of the correct result. Except for two streams, the greater amount of dispersion has no considerable positive effect on the speed of convergence. For two streams, there is a speedup of around 40%. The reason for this phenomenon is that, for the two streams, the steady state probabilities are distributed closer to the initial condition. Nonetheless, since the two stream approximation already executes very fast, the conclusion is that the initial condition selection proposed in Section 2.1.3 has no significant, positive impact on the speed of our approximation algorithm.

4.6 Static Sharing *vs.* Dynamic Sharing

In Section 3.2, it was proven that static process sharing always results in greater residence times than the dynamic process sharing used by the SMBS algorithm. Now, a few interesting experiments that verify this theoretical result are shown and discussed. To compare the static sharing scheme to the dynamic sharing scheme, several rounds of experiments simulating the behavior of both of these schemes are run on a single server with two flows. The parameters for the two flows are presented in Table 4.1. However, unlike previous experiments, the cost function for each of the flows is just the value of total end-to-end delay (*i.e.*, $cost = delay$).

Figure 4.23 depicts the ratio between weighted average costs under static and dynamic process

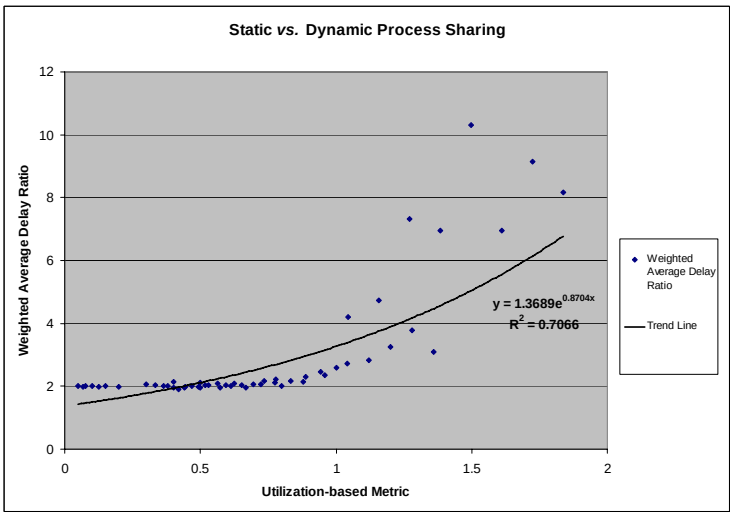


Figure 4.23: Dynamic vs Static Process Sharing

sharing. The weighted average cost, as before, is computed by taking the cost of delay for each flow and weighing using the corresponding arrival rates. The figure plots this ratio against the utilization metric, which is

$$U_{metric} = (\frac{\lambda_1}{\lambda_2})\rho, \text{ where } \lambda_1 \geq \lambda_2.$$

In Figure 4.23, the cost ratio between static and dynamic process sharing grows exponentially with U_{metric}. This result may look surprising; one might expect that with increased utilization the static sharing behavior should approach dynamic process sharing, because at high utilization both streams share the processor most of the time (Figure 4.24).

However, under static sharing, each flow experiences exponentially increasing queuing delays due to the fact that each share gets only 50% of the CPU. Not only do the jobs get half of the processing capacity when the other half is vacant, but future jobs have to wait in queue while the current job finishes. Thus, as utilization grows, the queuing delays for each share increase exponentially forcing the difference between static and dynamic process sharing to increase. This behavior also explains why ρ is scaled by $\frac{\lambda_1}{\lambda_2}$ in U_{metric}. Consider two scenarios with equal utilization. One scenario has two streams with equal arrival rates and the other has a fast and a slow arriving stream. The greater the gap in arrival rates, the more benefit dynamic process sharing brings, because of the queuing delays for the fast stream. In short, dynamic process sharing outperforms static process sharing as was proven in Section 3.2. Moreover, the advantage of the dynamic scheme increases with utilization due to growing queuing delays on each share in the static process sharing algorithm.

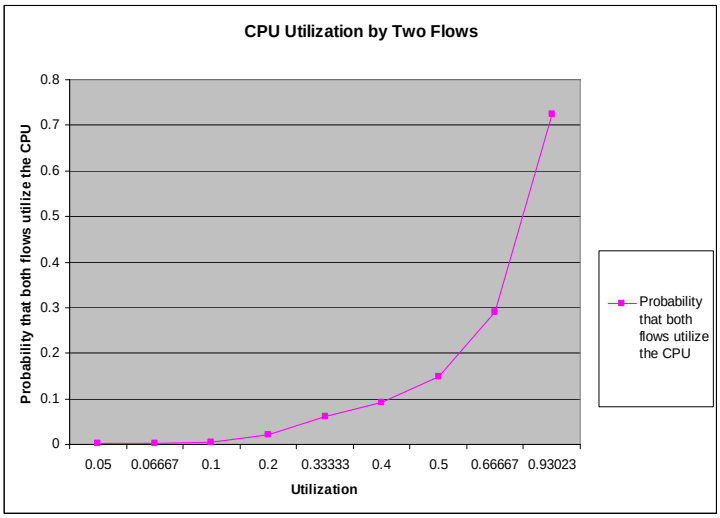


Figure 4.24: Probability that both streams use the CPU under the dynamic process sharing scheme

4.7 Cost of Average *vs.* Average Cost

In the previous section, issues of speed and accuracy of the SMBS approach were addressed. Now, we demonstrate the effect that the substitution of the cost of average delay metric for the average cost metric has on the scheduling. The experiments are run on two servers with different quality of service functions. For each experiment the cost of average is compared to average cost. There are three batches of experiments. In the first batch, the two streams have equal rates. In the second batch, one stream has faster arrival and greater service rates and is more expensive than the other. In the third batch, the stream with faster arrival and greater service rates is cheaper than the other. These experiments are summarized in Tables 4.4–4.6.

	Stream Parameters		**q(E(x))**	**E(q(x))**
	$\lambda_1 = 220$ $\mu_1 = 100$ Algorithm = SMBS-ES	$\lambda_2 = 220$ $\mu_2 = 100$ Algorithm = SMBS		
Experiment 1	x	$10x$	6014	6014
Experiment 2	$ln(5x + 500)$	$log_5(10000x + 500)$	8.93	9.59
Experiment 3	$2x^2$	$\frac{x^3}{200}$	$3.3e7$	$4.8e6$
Experiment 4	$\frac{100}{1+e^{10-x/110}}$	$\frac{100}{1+e^{3-x/550}}$	30.3	29.97
Experiment 5	$\frac{x^2}{200000}$	$log_5(10000x + 500)$	11.18	7.705
Experiment 6	$log_5(10000x + 500)$	$\frac{100}{1+e^{12-x/110}}$	7.27	5.25

Table 4.4: Cost of Average *vs.* Average Cost Experiment Parameters (Experiments 1–6)

The difference between cost of average and average cost is depicted in Figure 4.25. This result corresponds to theoretical analysis carried out in Section 2.5. For linear QoS functions, both metrics match as expected (Experiments 1, 7, and 13). For concave QoS functions, the average cost is below

the cost of average delay (Experiments 2, 8, and 14). However, the difference is small due to the nature of the log function. For convex QoS functions, the average cost is above the cost of average delay (Experiments 3, 9, and 15) and the magnitude of the difference is consistently amplified by the polynomial cost function. For the case of sigmoid QoS functions, there is both over-estimation (Experiments 10 and 16) and under-estimation (Experiment 4). Moreover, the magnitude fluctuates highly from experiment to experiment, which shows that sigmoid functions have the most potential for "tricking" the SMBS scheduling algorithm.

	Stream Parameters			
	$\lambda_1 = 190$ $\mu_1 = 170$ Algorithm = SMBS-ES	$\lambda_2 = 400$ $\mu_2 = 10$ Algorithm = SMBS	**q(E(x))**	**E(q(x))**
Experiment 7	x	$10x$	1401	1401
Experiment 8	$ln(5x + 500)$	$log_5(10000x + 500)$	8.32	8.75
Experiment 9	$2x^2$	$\frac{x^3}{200}$	$1.22e7$	$6.39e6$
Experiment 10	$\frac{100}{1+e^{10-x/110}}$	$\frac{100}{1+e^{3-x/550}}$	44.75	69.56
Experiment 11	$\frac{x^2}{200000}$	$log_5(10000x + 500)$	29.2	16.87
Experiment 12	$log_5(10000x + 500)$	$\frac{100}{1+e^{12-x/110}}$	6.87	7.07

Table 4.5: Cost of Average Experiment Parameters (Experiments 7–12)

	Stream Parameters			
	$\lambda_1 = 190$ $\mu_1 = 170$ Algorithm = SMBS-ES	$\lambda_2 = 400$ $\mu_2 = 10$ Algorithm = SMBS	**q(E(x))**	**E(q(x))**
Experiment 13	$10x$	x	14981	14981
Experiment 14	$log_5(10000x + 500)$	$ln(5x + 500)$	8.97	9.24
Experiment 15	$\frac{x^3}{200}$	$2x^2$	$1.08e8$	$2.2e7$
Experiment 16	$\frac{100}{1+e^{3-x/550}}$	$\frac{100}{1+e^{10-x/110}}$	33.4	49.4
Experiment 17	$log_5(10000x + 500)$	$\frac{x^2}{200000}$	6.83	7.05
Experiment 18	$\frac{100}{1+e^{12-x/110}}$	$log_5(10000x + 500)$	9.84	16.02

Table 4.6: Cost of Average Experiment Parameters (Experiments 13–18)

In Section 2.5, it was mentioned that cost of average delay could be adjusted such that its difference from average cost of delay is minimized. Now, two strategies are presented to achieve this goal. The first strategy is based on constructing histograms of true residence time on each server and using the histograms to compute approximate average cost. The histogram approach greatly reduces the error between cost of average and average cost, but it is impractical to construct such histograms in real life, because the system needs to evaluate several different possible partitions very quickly during the optimization process. The second strategy is model-based. Experimentally, it can be seen that the distribution of residence times under SMBS resembles an exponential distribution. Thus, when the mean residence time is obtained from the Markov model, it can be used to build an exponential distribution and compute average cost. A continuous exponential distribution can be

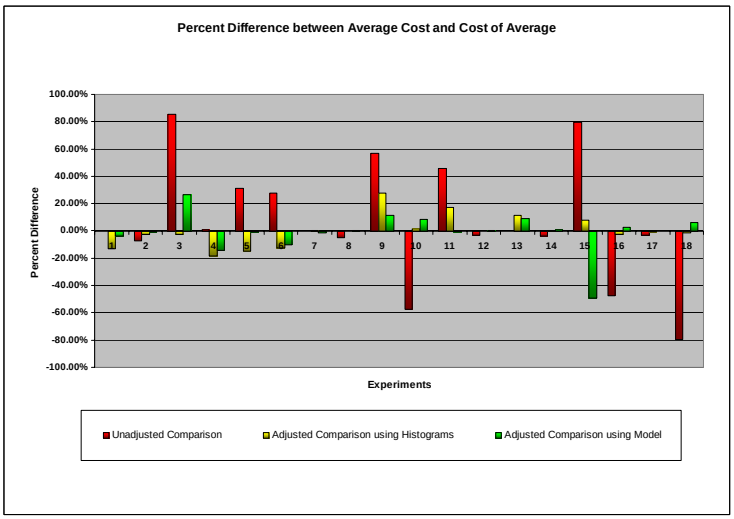


Figure 4.25: Cost of Average *vs.* Average Cost (Experiments 1–18)

discretized to obtain average cost as follows:

$$\overline{Cost}_i = \sum_{i=0}^{c\overline{D}} re^{-ri} q_i(i), \text{ where } r = \frac{1}{\overline{D}} \text{ and } \overline{D} \text{ is average delay from the Markov model.}$$

Unfortunately, at this time, we have no rigorous error bounds for our model-based approach.

In Figure 4.25, the errors between simulated average cost and average cost from two strategies are shown together with the cost of average delay. Red bars depict the errors from Experiments 1–18. Yellow bars show the value of average cost of delay computed by generation histogram. Each histogram bucket is set to 20% of standard deviation. If the buckets were made small enough, the error would reduce to zero, because the histogram would represent the true distribution of residence times. Green bars in Figure 4.25 depict the difference between the model-based approach and the true average cost obtained from simulation. This difference is generally much smaller than the difference from the first experiment with unadjusted SMBS. There are still noticeable discrepancies for convex QoS functions due to error being amplified by the cubic cost function. The error in Experiment 9 is less than in Experiment 15, because in Experiment 9 the cubic cost function is applied to the slow stream, which has a lower-magnitude error than the faster stream in Experiment 15.

In short, the model-based adjustment approach to the SMBS scheme results in narrowing the gap between the cost of average delay and the average cost of delay, making SMBS a viable competitor to DQC and DQC-NR.

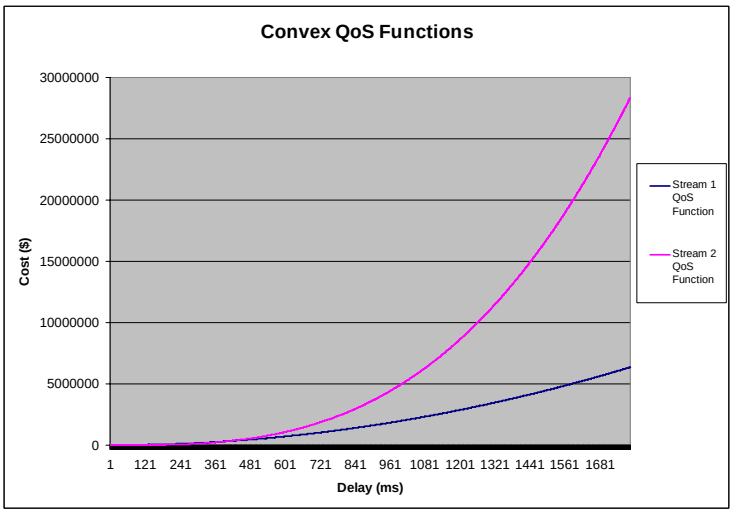


Figure 4.26: Convex QoS Functions

4.8 Smart Scheduling Effectiveness

Finally, we compare the effectiveness of the proposed scheduling algorithms, SMBS, DQC and DQC-NR, under various conditions. In particular, we show how these algorithms behave depending on utilization, quality of service functions, and number of streams. FIFO is used as a baseline for the comparison. These factors are studied in a single server environment. The experiments over a queueing network are discussed in Chapter 5.

We start by looking at the behavior of SMBS, DQC and DQC-NR under convex quality of service functions. Figure 4.8 depicts quality of service functions for two streams used in the experiment and the advantages our three algorithms exhibit under different utilizations. For these experiments, both streams have similar arrival and service rates. As seen on the graph, all three algorithms perform similarly. This is the case because two QoS functions are increasing non-intersecting functions. Thus, all three algorithms always give priority to the most costly stream. The advantage diminishes with utilization, because marginal cost decreases as the delay is reduced. The greater the utilization, the more impact our algorithms have on delay reduction .

The behavior of SMBS, DQC and DQC-NR under concave QoS functions is depicted in Figure 4.8. Unlike the convex case, the marginal advantage increases with utilization. This occurs for exactly the same reason as in the convex case, but concave functions have increasing marginal cost as delay decreases. In addition, DQC outperforms SMBS and DQC-NR here, because DQC allows message reordering. DQC degenerates into LCFS under concave QoS functions. Thus, neither SMBS nor DQC-NR can achieve the same results because no message reordering is performed.

The sigmoid case, presented in Figure 4.8, illustrates the failure of the SMBS algorithm due to

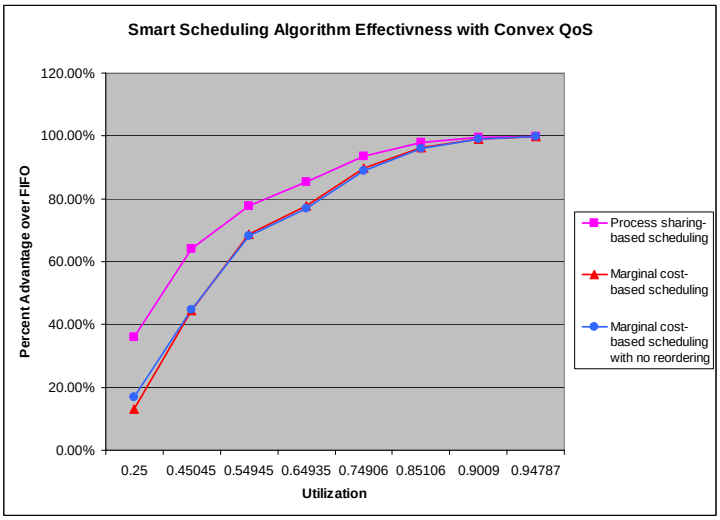


Figure 4.27: Effectiveness under Convex QoS Functions

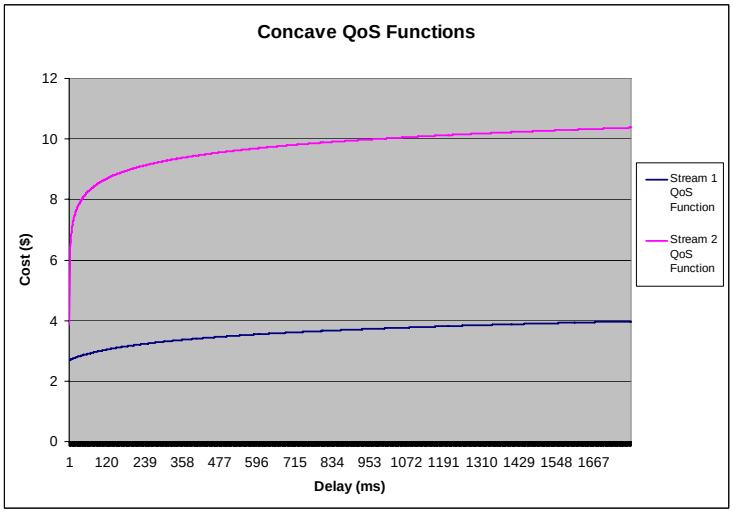


Figure 4.28: Concave QoS Functions

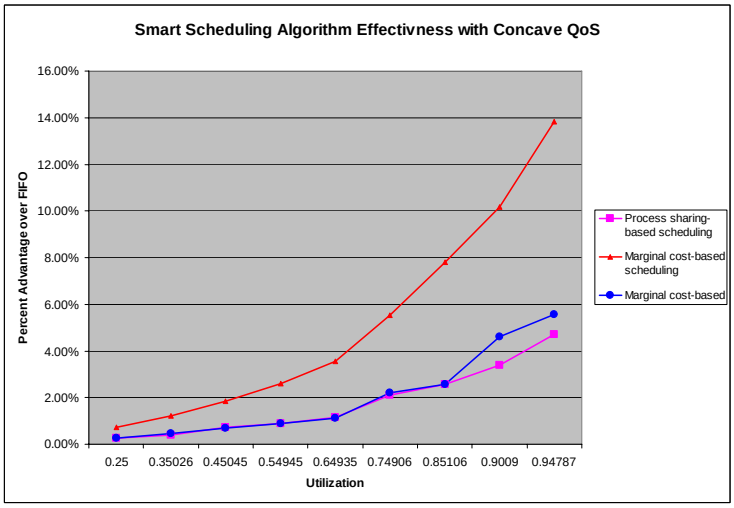


Figure 4.29: Effectiveness under Convex QoS Functions

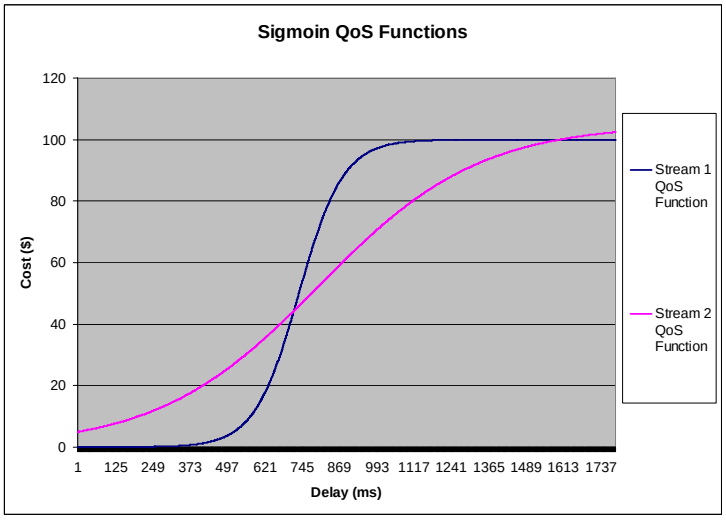


Figure 4.30: Sigmoid QoS Functions

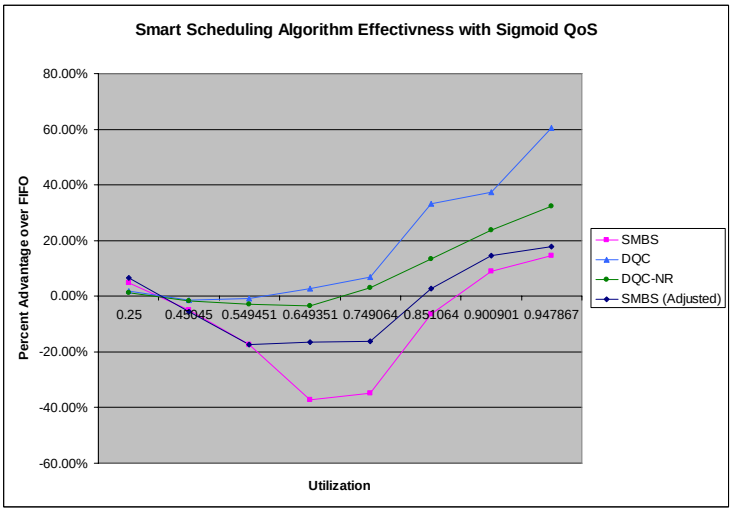


Figure 4.31: Effectiveness under Sigmoid QoS Functions

mismatch between average cost and cost of average. Thus, the adjustment presented in Section 4.7 is added to compare it with plain SMBS behavior. Adjusted SMBS performs better than plain SMBS, but still not as well as plain FIFO. The reason for the failure of adjusted SMBS is the disparity described earlier between 50/50 sharing and FIFO. For utilization between 40 and 60 percent, Adjusted SMBS picks optimal sharing priorities around 50/50. However, in this case, FIFO outperforms equal sharing. The growth of advantage of DQC, DQC-NR and to a lesser extent adjusted SMBS with increased utilization illustrates the nature of sigmoid curves; because no cost is assigned for delays below a threshold, a certain amount of queueing delay should be present to make our algorithm outperform FIFO.

To complete the analysis of QoS impact on the efficacy of scheduling, a combination of quality of service functions is used such that one stream has a sigmoidal function and the other has a linear one. The outcome of this experiment is shown in Figure 4.8 and is quite similar to that of the previous experiment, because the linear function has a segment at which one stream is more expensive than the other and a segment where priorities switch. This is similar to the two sigmoid functions presented earlier.

In the next set of experiments, we see how smart scheduling effectiveness depends on the number of streams deployed on the server. In all the experiments, the total utilization on the server is 90%. The arrival and service rates for all streams are equal. In the first round, we run DQC and DQC-NR on a single server, increasing the number of streams with each iteration. Each new stream has a linear QoS such that the cost is equal to $i * delay$, where i is the total number of streams. Thus, each additional stream has a higher cost than the last stream added. The results are depicted in Figure 4.34. The gain of DQC and DQC-NR over FIFO increases linearly with the number of streams.

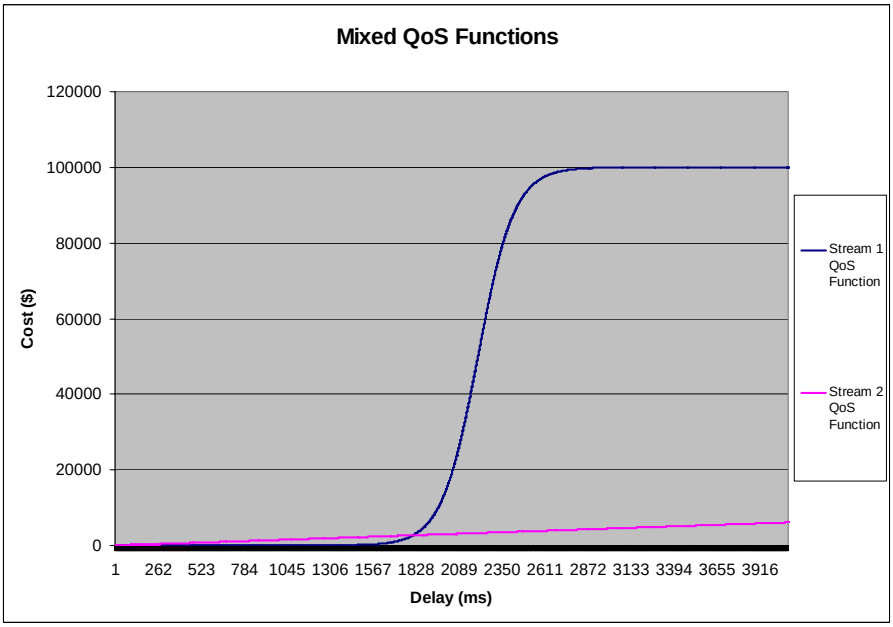


Figure 4.32: Mixed QoS Functions

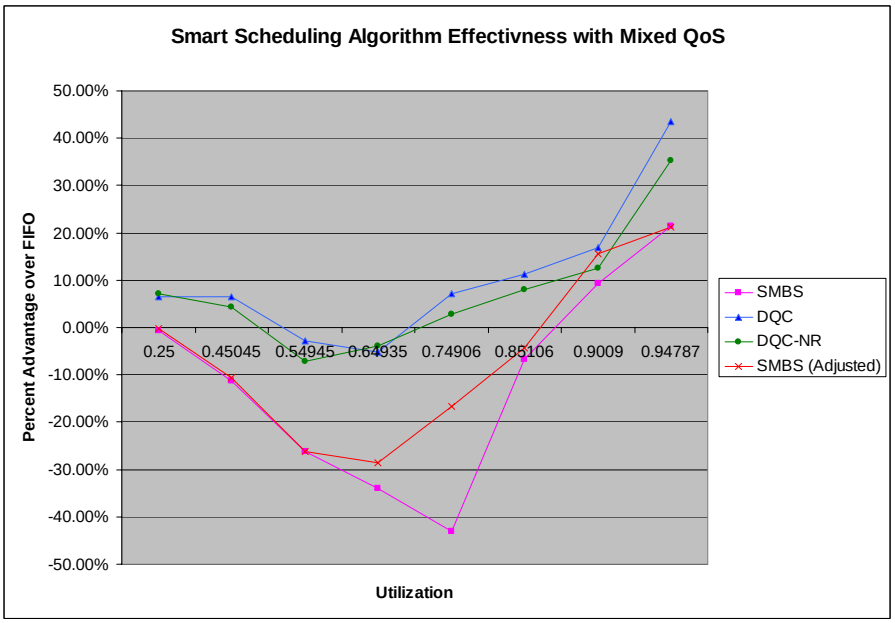


Figure 4.33: Effectiveness under Mixed QoS Functions

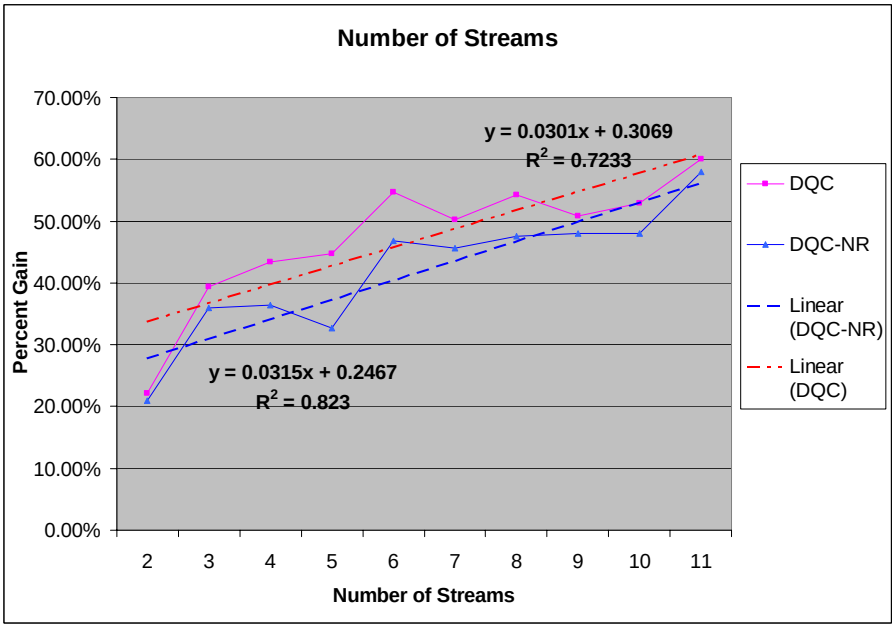


Figure 4.34: Number of Streams

However, this result doesn't prove that the gain we obtain is due to the increased number of streams on the server because the marginal cost between the cheapest and the most expensive stream also increases with the number of streams. To remove this factor, in the next round, we repeat the same experiments making only half of the deployed streams have QoS function $q_i(d) = d$, and the other half have QoS function $q_i(d) = 2 * d$. Thus, the difference between the cheapest and the most expensive streams remains constant from iteration to iteration. The results are shown in Figure 4.35. There is no additional gain from DQC and DQC-NR with the increased number of streams. This can be explained by the fact that high cost streams are competing for the same share of the CPU. The total share of the CPU given to all high cost stream increases, but there are more streams competing for this share.

As we have seen from the experiments presented in this section, the effectiveness of smart scheduling algorithms is determined primarily by the nature of quality of service functions and by the ratio of rates between streams as was shown in Section 4.2, as opposed to other factors such as the total number of streams deployed on the server.

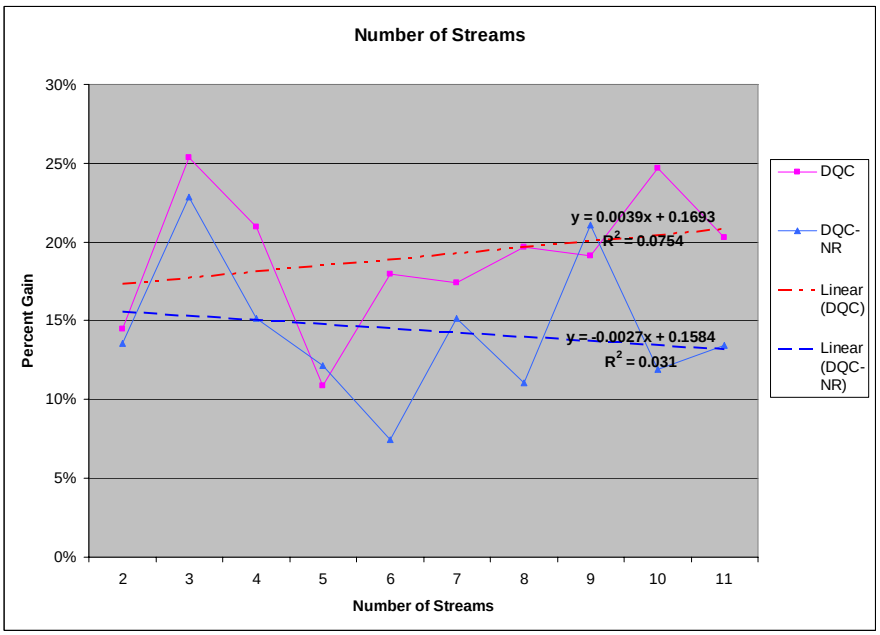


Figure 4.35: Number of Streams

Chapter 5

Experimental Results: Queuing Network

In this chapter, we examine the behavior of our algorithms in a multi-server environment. Since there are no provably optimal algorithms, the chapter introduces only limited examples that try to show that even simple heuristics can outperform a simple first-in, first-out scheduling algorithm.

The simplest multi-server environment consists of two servers. There are three flows deployed on these servers: two flows on each server, with one flow being executed on both servers. Figure 5.1 shows the flows' deployment. In the first round of experiments, the utilization on both servers is maintained at 90%. The quality of service for Streams 2 and 3 is linear with equal slope, and Stream 1 has a sigmoidal QoS. At each iteration of this round, the slope of the linear QoS function for Streams 2 and 3 is increased. The goal of the experiment is to show that, as the slope is increased, the efficacy of smart scheduling with feedback decreases. The reason for this decrease lies in the fact that feedback should force scheduling to prefer Stream 1 to Stream 2 on Server 1, although Stream 2 should be scheduled ahead of Stream 1. This happens because if Stream 1 is scheduled as usual, it encounters a huge delay on Server 2 due to the sigmoidal QoS. As the slope for Stream 2 increases, prioritization of Stream 1 becomes more costly. In this round, the smart algorithms with feedback are compared to the same algorithm with no feedback.

In Figure 5.2, the results of the first round are shown. First, it should be noted that the efficacy of the DQC algorithm with and without feedback drops as both Stream 1 and Stream 2 become more expensive. This drop occurs because, as slope increases, the streams on each server converge. As

	Stream 1		**Stream 2**	**Stream 3**
QoS function	$\frac{100000}{1+e^{20-x/110}}$		ax where $a \in [1, 25]$	ax where $a \in [1, 25]$
	Server 1	*Server 2*	*Server 1*	*Server 2*
Mean inter-arrival time	220	220	220	220
Mean service time	100	100	100	100

Table 5.1: Queuing Network: Experiment Parameters (Round One)

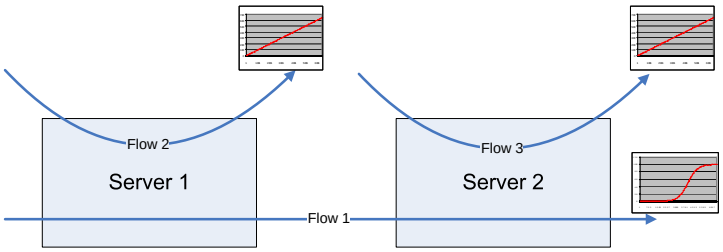


Figure 5.1: Two Server/Three Stream Example

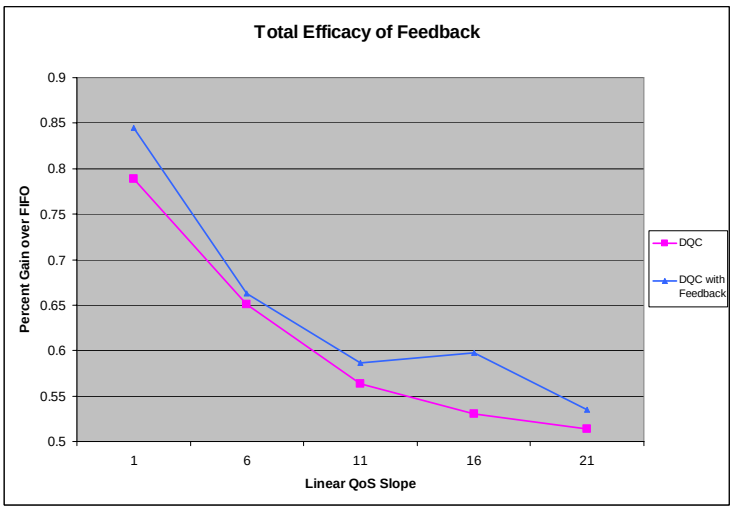


Figure 5.2: Total Efficacy of Feedback

the costs of the streams equalize, the benefits of the smart scheduling decrease. Second, as the slope of the linear QoS increases, the difference between DQC with and without feedback decreases. The total advantage of the algorithm over FIFO is computed based on average cost for all three streams. Thus, as the costs of Streams 2 and 3 increase, the benefit gained for Stream 1 becomes diluted. To verify this hypothesis, the cost for Stream 1 alone is depicted in Figure 5.3. The advantage of DQC with feedback over DQC for Stream 1 alone remains constant no matter what slope is chosen for Streams 2 and 3. In the next round, we show how the current parameters can be changed to achieve greater advantage.

In the first round of experiments, the gap between DQC scheduling with and without feedback is very narrow. In the next round of experiments, we will show that the gap can be made arbitrarily large. To make this gap wider, we replace the quality of service functions from Table 5.1 with the functions listed in Table 5.2.

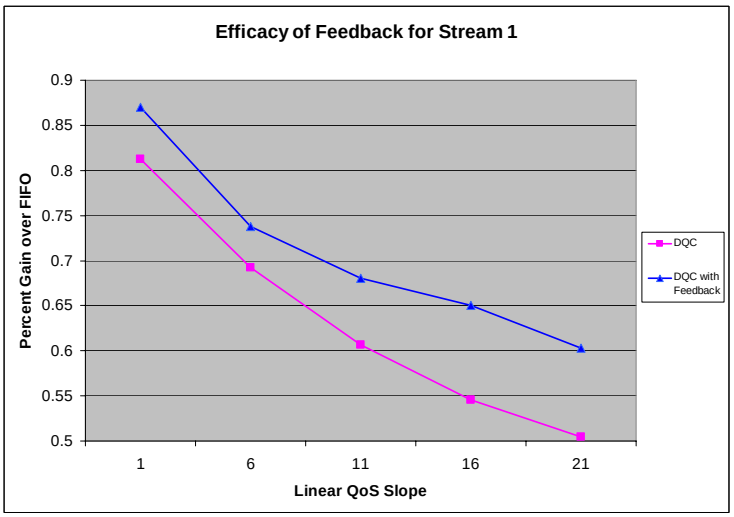


Figure 5.3: Efficacy of Feedback for Stream 1

Stream 1	$\frac{100000}{1+e^{15-x/10}}$
Stream 2	$ax + 100$ where $a \in [1, 21]$
Stream 3	$a'x + 100$ where $a' \in [1, 21]$ and $a = a'$

Table 5.2: Queuing Network: Experiment Parameters (Round Two)

In this round, the quality of service function for Stream 1 is made much steeper than in the first round. The new sigmoidal QoS is also shifted to the left. Moreover, for linear QoS, a minimum cost of 100 is introduced to make sure that the linear QoS intersects the sigmoidal curve in at most two places (*i.e.*, it removes the case when, for small delays, linear QoS assigns less cost than sigmoidal). The change in sigmoidal QoS is depicted in Figure 5.4.

The results of the second round are depicted in Figure 5.6. The gap between DQC with and without feedback is bigger than before. With the new sigmoidal QoS, the cost of not offseting the delay on the first server results in much greater penalties than before, because the new sigmoidal QoS assigns maximum cost for lesser delays. Thus, if the delays are not offset, the average cost is around 100000, since the vast majority of the events hit the maximum cost for Stream 1.

It should be noted that the overall percent advantage of DQC with and without feedback over FIFO is less than before, because the new quality of service function assigns greater cost for message delay in Stream 1. Smart scheduling under these conditions cannot greatly decrease cost, because it cannot physically prioritize messages in Stream 1 enough to avoid the cost ceiling. In other words, no matter what scheduling does for Stream 1, most of the messages will be delayed by more than 100 units of time.

Another interesting fact that can be observed from Figure 5.6 is that the advantage of DQC increases with the slope of the linear QoS functions. This is different from the first round of exper-

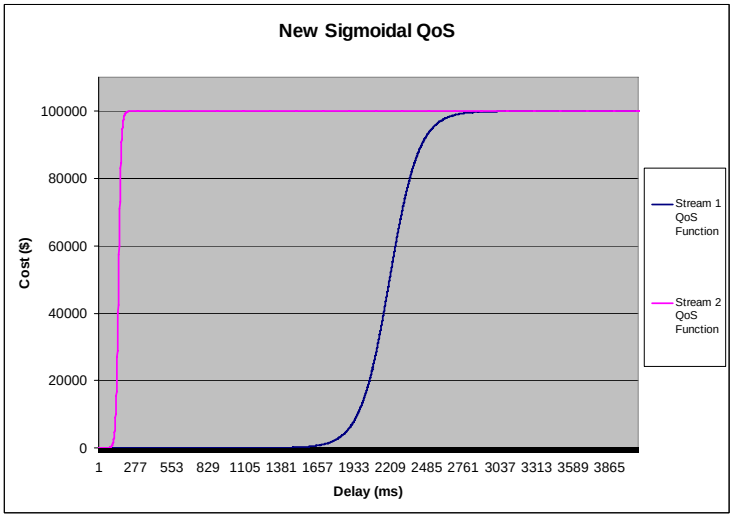


Figure 5.4: Change in Sigmoidal QoS

iments. DQC improves the average cost for all three streams. However, Stream 1's cost is vastly bigger than the cost for other streams. The differences decrease as the slope of linear QoS increases. Thus, the cost saving for Streams 1 and 2 impacts the average cost of the three stream greater as the slope of linear QoS increases. The overall gain over FIFO increases. This can be confirmed by looking at Figure 5.5. The average cost gain for just Stream 1 doesn't change with slope, because of Stream 1's dominant cost over other streams. Thus, the lower cost is due to the gains for Streams 2 and 3.

In the next round of experiments, to test the impact of having no reordering, we replace the DQC algorithm with DQC-NR and use the same parameters as in the second round. The results are depicted in Figure 5.6 and 5.5. We notice that the advantage from feedback is negligible compared to DQC. The reason for this poor performance is that the metric used by the scheduler takes the length of the queue into account. Since the utilization is high, the queue length is substantial. The queue length metric dominates feedback values. Thus, the feedback has less impact in the DQC-NR case.

In the fourth round of experiments, we try the SMBS algorithm with and without adjustment. There is no feedback for the SMBS algorithm, since the optimization algorithm is centralized. For the SMBS algorithm without adjustment, the behavior is similar to DQC-NR. The increased gain over FIFO is solely due to decreases in cost for Streams 2 and 3. The cost for Stream 1 is not impacted (see Figure 5.5). In fact, the optimization algorithms give all the CPU to Streams 2 and 3, because the cost of average delay is always at 100000. SMBS with cost adjustment produces a reduction in cost for Stream 1 (See Figure 5.5). However, the overall cost for all streams actually increases with increased slope, making SMBS with adjustment perform worse than FIFO (see Figure 5.6). The

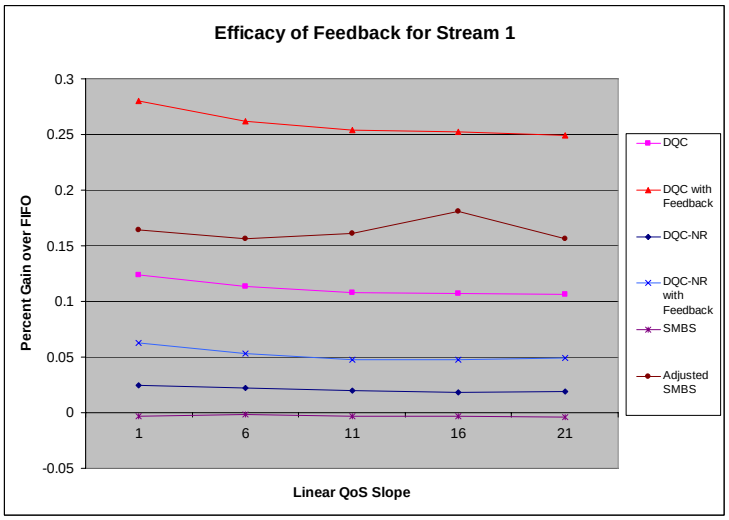


Figure 5.5: Efficacy of Feedback for Stream 1

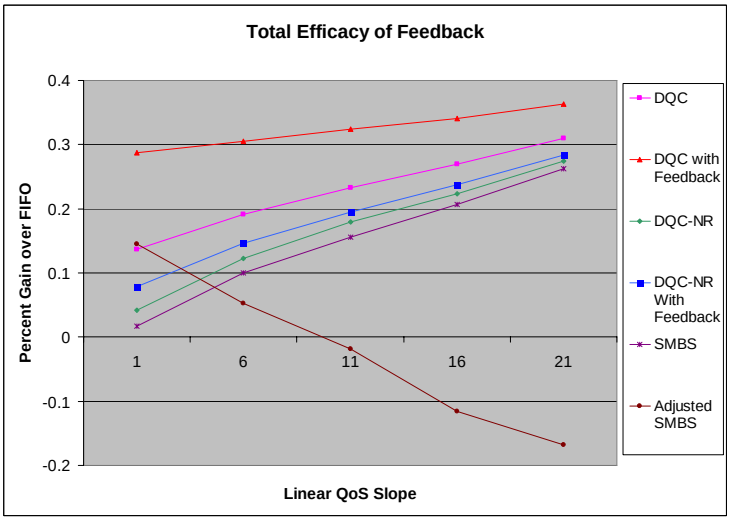


Figure 5.6: Total Efficacy of Feedback

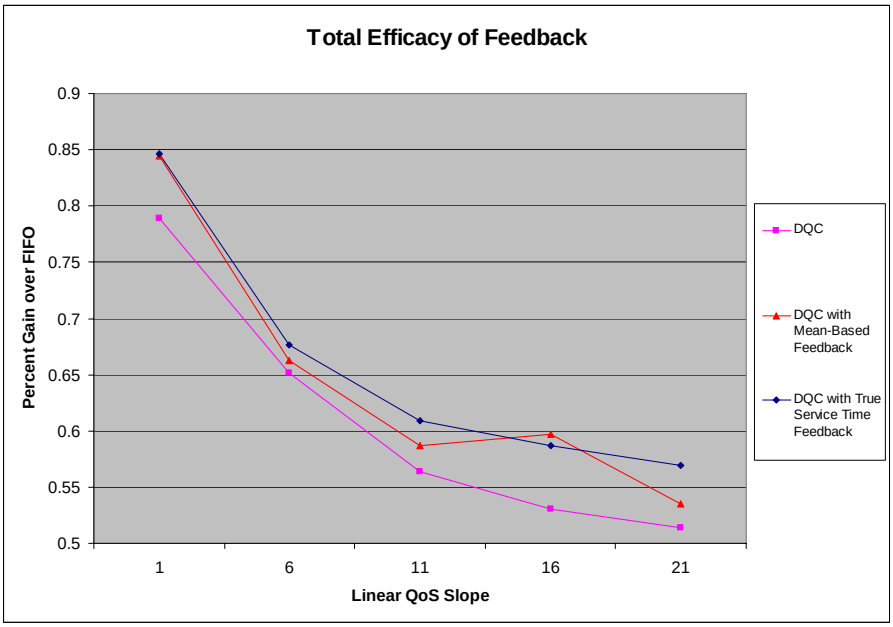


Figure 5.7: Efficacy of Exact Service Time Feedback

optimization algorithm underestimates the impact of the increased share of Stream 1 on Streams 2 and 3. Thus, as the slope of linear QoS increases, the negative impact of this underestimation increases.

In the last round of experiments, we try to change the feedback. From our choice of feedback, it may look as though if the exact future service time was known, a greater reduction in cost could be achieved. We try to verify this assumption, by replacing the mean service time with the exact future service time for each corresponding message. The rest of the parameters are kept the same as in the first round of experiments (Table 5.2). The results are depicted in Figure 5.7. The new types of feedback do not change the efficacy of the algorithm. The reason for such disappointing results is that the mean value generally overestimates the true service time. However, the service time is not the only source of the future delay; the queuing delay is another source. Thus, neither mean service time nor exact service time represent the true future delay experienced on Server 2.

In summary, even the simplest feedback information introduced in this chapter improves DQC scheduling algorithm performance. Moreover, the absence of reordering makes the efficiency of feedback very limited. A different kind of feedback has to be developed for this case. The issues of average cost of delays *versus* cost of average delay presented in Section 4.7 are more acute in a multiple server environment. In order to make SMBS suitable for a distributed environment, a better way of adjusting predictions from the Markov model should be developed to close the gap between average cost of delay and cost of average delay.

Chapter 6

Applications

There are several areas where stream processing is relevant. These areas include financial sector applications, military applications, enterprise integration software, and general purpose ubiquitous messaging environments.

Finance is one of the areas where stream processing could be used. Events on the stock market such as changes to stock prices and the amounts of a given stock sold and bought could be streamed into the systems. An analyst could then use this data by applying different models and correlating different streams in an attempt to predict and capitalize on market behavior.

In large financial institutions, the computing resources are often organized in clusters that are connected by dedicated fiber-optic channels. An example of such a topology is depicted in Figure 6.1. In the picture, each node represents a cluster. This topology is similar to the network layout employed by IBM, where the core ring is comprised of clusters belonging to specific geographic regions such as "U.S. Northeast". The nodes that are attached to regional clusters are local office networks such as "Hawthorne, NY". Such a topology is well suited for distributed, analytical stream processing including financial analysis. Moreover, different agents may have different delay requirements, which are guided by their ability to pay for up-to-date information. Thus, the issue of QoS-based distributed scheduling is relevant in such systems.

Sensor networks compare another fertile area of research that requires stream processing. The *battlefield of the future* serves as a good example of a sensor network that requires distributed stream processing. Modern warfare relies heavily on computing power. Soldiers carry wearable computers. There are UAV drones hovering above the battlefield. Tanks and other units contain computing power and have multiple sensors that stream information during the battle. In such an environment, there is a need to process all information rapidly to help soldiers identify immediate threats and opportunities. Most of the processing involves computationally intensive tasks such as image recognition, threat assessment, *etc.* Moreover, sending information to a central command-and-control is impossible due to the delay and power cost of such communication. Instead, the computing power on the battlefield can form a distributed network to process incoming data streams. Such a

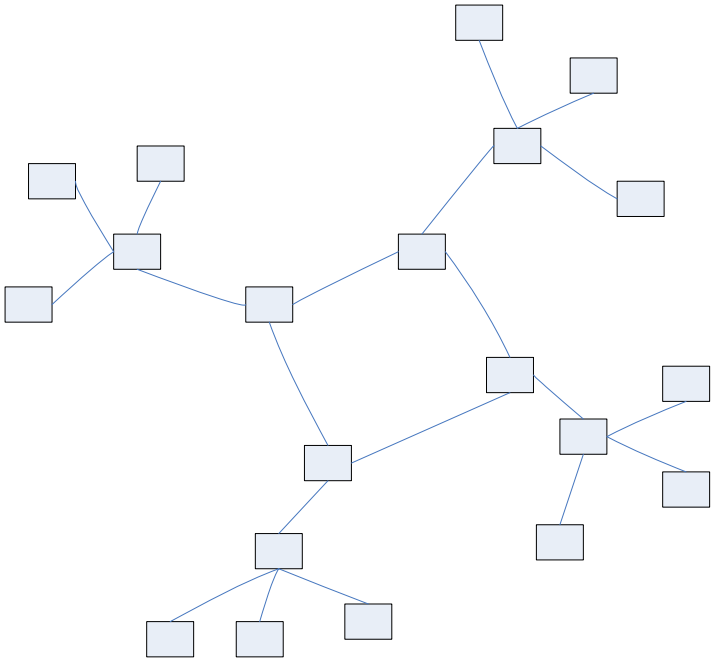


Figure 6.1: Example of cluster topology for an enterprise network

network would not be fully connected. Most likely, wireless links would be established to maximize power consumption. Moreover, the number of streams and their priorities could constantly change as threats come and go. All these features fit well with the description of stream processing applications in Chapter 1.

Enterprise application integration is another area where stream processing may be useful. A large percentage of the software engineering market is occupied with connecting various enterprise application (*e.g.* SAP human resource software with accounting software). The integration could be achieved by creating adapters that export application information as streams of messages. These messages represent changes in application state. The system transforms and disseminates these changes to integrated applications. Transformations may include some business logic and conversions that do not belong in the application at either end. Such an integration system could be a stream processing system for which QoS-based scheduling is required.

The next generation of programming languages and associated runtimes may incorporate streaming into programming languages themselves. Currently, Service-Oriented Architecture (SOA) is being presented as the latest word in system design. In SOA, a system is built from a set of services with standard descriptors, which communicate with each other via message dispatch (SOAP, REST, *etc.*). A programmer can build a system by wiring services together. This approach can be taken to a new level by making classes and components of Java-like languages perform as services such that every function call becomes a message dispatch. Then, the run-time is responsible for taking these components and either running them locally or sending them to be executed on remote runtime engines. In such a runtime, all objects are viewed as services exchanging streams of data. This approach incorporates streaming into the language itself. The runtime for such a language could be distributed as in Hermes [51]. Therefore, scheduling in a distributed stream processing system would be also relevant for such a runtime. The execution priorities for processes would determine the Quality of Service functions.

Chapter 7

Related Work

In general, the area of scheduling is very extensive, especially, in the domains of operating systems and networking [31, 33, 58]. Since our work is uniquely focused on distinct aspects of distributed data-stream processing systems, we discuss only the work that is directly related to scheduling in a stream processing applications. A good description of the requirements for such systems is provided by Bourbonnais *et. al.* [8].

There are several stream processing systems being developed in academia and industry. One such system is SMILE (Smart-Middleware Light-End) [29], from IBM Research. Like many other stream processing systems, SMILE defines a stream processing variant of SQL for querying information over streaming data. The distinctive aspect of SMILE is its novel correctness guarantee model. SMILE introduces an *eventual correctness* guarantee that is less stringent than ACID and is very well suited for stream processing applications. Under eventual correctness, all operators are defined to be deterministic and monotonic. This allows SMILE to have simple fault-tolerance algorithms. Eventual correctness gives great flexibility in optimization such as mapping and scheduling. Also, as part of the SMILE system, a novel mapping algorithm that uses queuing theory analysis has been developed. This analysis was one of the sourses of inspiration for this work.

Another stream processing system, called STREAM [5], is being developed by a team at Stanford. STREAM proposes a SQL-like language, called CQL, for querying streams. Unlike SMILE, STREAM emphasizes approximations of the correct result to speed up query processing and minimize memory use. The scheduling algorithm proposed as part of STREAM [3] focuses on memory management. It does take into account latency bounds on query execution, but it is not QoS driven as Borealis [11] and our algorithm are. In addition, this algorithm is not designed for a distributed environment, where local choices may negatively effect latency on downstream server machines.

Borealis is the first stream processing system that uses QoS functions to drive system performance. It is a second generation stream processing system, built as an extension to Aurora and Medusa [1] systems. Borealis allows users to define streaming computations in a graphical fashion. In addition, it allows users to specify several different QoS requirements. There are a number of

novel algorithms proposed as part of Aurora and Borealis that include scheduling and mapping. Abadi *et al.* [1] present the design of the future Borealis system. It outlines the distributed scheduling algorithm that is being developed for the next version of the system. In spirit, our approach is similar to one described by Abadi *et al.* since our algorithm tries to use auxiliary information that travels in messages to predict the impact of scheduling policy on global QoS objectives. However, our framework proposes to use control to make local scheduling more adaptable and to alleviate the need for several tiers of optimization algorithms proposed by Abadi *et al.*. Our framework removes the need to subdivide the network in order to implement a centralized scheduling policy. While removing these constraints, our algorithm does not utilize any load shedding or message reordering schemes outlined by Abadi *et al.*. In the future, it would interesting to compare different scheduling strategies and incorporate other types of QoS functions into our framework.

The scheduling strategy in Abadi *et al.* is based on Aurora scheduling outlined by Carney *et al.* [11]. Carney *et al.* propose several schemes to schedule sets of query execution graphs with associated quality of service constraints. All schemes are essentially greedy. Several metrics are proposed to optimize for different things including cost of execution per operator, latency and memory requirements. In cases where QoS is defined, the operators are evaluated based on the expected impact on utility. Thus, the scheduling is operator-based and not flow-based. In order to cut down on the overhead, scheduling decisions are made for a set of tuples rather than on a per-tuple basis. However, the approach of Carney *et al.* does not take into account the effect of queuing in the distributed stream processing system. As was shown in our work, queuing could have a great impact on system performance and must be taken into account by the scheduling algorithm to achieve good performance.

NiagraCQ [13] is a streaming system being developed at University of Wisconsin, Madison. Unlike previously described projects, NiagraCQ utilizes an XML-based query language. NiagraCQ proposes several heuristics for combining selection and join predicates, which could be useful for mapping algorithms. However, no specific scheduling algorithm is proposed.

TelegraphCQ [12] is a Continuous Query processing system from UC Berkeley. The core goal is to make the system highly adaptable in the face of a changing environment. The key components of the system include the Fjord API [34] that allows push and pull-based communication between components of the query plan, and Eddys [26] that encapsulate the logic of the streaming operator and permit on-line self-optimization. The implementation of TelegraphCQ is not distributed and does not use Quality of Service functions, thus the issues of scheduling to control execution cost are not relevant.

There are several other stream processing systems, including Tapestry [54], Gigascope [20], Hancock [19], Tangram [40], Tribeca [52], HourGlass [42, 47] and IrisNet [24].Tapestry is one of the first stream processing systems described in the literature. It was developed at Xerox PARC in the

early 1990s. The system proposes an extension to the standard database that would allow users to issue continuous queries over append-only tables. Tapestry defines the notions of incrementality in query results and monotonicity, which are later used by the SMILE system. Tapestry also defines an associated extension to SQL that lets users define these continuous queries. Since Tapestry is envisioned as a database extension, the issue of distributed implementation and the associated problem of scheduling is not examined [54].

Gigascope and Tribeca are examples of stream-oriented systems specifically designed for network monitoring. Gigascope uses an SQL-like language. However, since many computations over networking data are hard to express in SQL-like languages, Gigascope provides a facility for easy addition of custom functions. Performance and extensibility are the main achievements of Gigascope. The architecture of Gigascope is distributed, but it uses a shared memory model. Thus the scheduling, as presented in this work, is not applicable to the Gigascope system [20]. Similar to Gigascope, Tribeca strives to achieve performance and extensibility. The language to specify transformations is procedural rather than SQL-like. However, the emphasis is put on the ability of analysts to add their own statistical models into the system. For performance reasons, Tribeca doesn't allow regular joins unless all joined streams have window operators applied to them. The Tribeca architecture is not distributed. The design contains several optimization strategies for better system performance including ad-hoc query optimization. These strategies are centered on eliminating unnecessary I/O operations and providing more robust memory management [52].

Another example of a domain-specific stream processing system is Hancock. Hancock is a C-like language and associated run-time that can be used to define signatures over customer transaction streams. Signatures are products of data mining used for fraud detection, customer service and marketing. Unlike general streaming projects or network traffic analysis projects, Hancock puts forth a domain-specific, extensible language for signature development. The runtime for the language is non-distributed. Thus, the issues of queuing and scheduling are not discussed in the context of Hancock [19].

Tangram is one of the earliest projects that proposed the evolution of regular databases into stream processing systems. Instead of persistent queries, a concept of transducers is introduced. Transducers are similar to iterators and represent computations being executed against the streaming data. The language for expressing the computations is PROLOG, not SQL. However, since Tangram is envisioned as a database extension, the issues of scheduling in a distributed environment are not relevant to Tangram's design [40]. Tangram also served as the inspiration for work done by Tucker and Maier [57] that defines the notion of *stream punction*, special meta-information messages that are injected into streams to improve the performance of streaming operators.

IrisNet is a project that takes a different approach from classical stream-oriented systems [24]. IrisNet views a stream-processing system as a widely distributed network of services. Each service

manages information produced by a set of sensors and lets users register queries on this data. IrisNet realizes that such services would have a lot of common components such as data storage and query registration. Thus, IrisNet proposes a framework for development of such streaming services. Although, IrisNet doesn't discuss scheduling of requests, IrisNet's system design makes a good target for the scheduling approaches outlined in this work.

HourGlass [47] is another service-oriented framework for collection and distribution of sensor data. Just like IrisNet, HourGlass tries to identify key features of sensor-data dissemination networks and create a specialized service-oriented framework to address these unique features. As part of the HourGlass project, an algorithm extension for network-oriented placement of operators is proposed. This algorithm is closely related to the mapping problem. However, similar to other mapping and scheduling solutions presented in the literature, queuing delays are not taken into account [42].

Finally, GATES is a grid-based middleware for processing distributed data streams. The design of GATES focuses on extension of the existing Open Grid Services Architecture to allow efficient distributed stream processing. To the best of our knowledge, GATES is the only other system that utilizes queuing theory to tune algorithm performance. However, unlike the work presented here, GATES uses statistics about queues to determine sampling rate on a server. GATES doesn't have a concept of quality of service function or a notion of feedback to determine impact of a local computation on overall quality of the result [14]. Some work in the grid environment has focused on control-based algorithms that try to meet certain overall quality of service. Li *et al.* [33] introduce framework for designing control-based algorithms. However, this work doesn't focus on streaming applications or queuing. Instead, a distributed framework for control of different resources such as bandwidth is developed. The goal of this framework is to achieve system stability while meeting certain global quality of service requirements. In this context, the quality of service is defined not as a function, but as a set of properties such as fairness, sampling rate, deadline fulfillment, and error management. In the future, we need to study how the control theory used by Li *et al.* could be used in a distributed streaming environment.

A streaming system could also be viewed as an overlay network produced by the mapping function. This view is similar to distributed hash tables (DHT, [36, 43, 44, 49, 60]). The Peer-to-Peer Information Exchange and Retrieval (PIER) system [27] proposes to use DHT for creating a widely-distributed relational database. The system is designed to handle regular database queries over relational data rather than continuous queries over the data streams. The main problem PIER is trying to solve is how to execute queries involving relational join over the data that is stored in a DHT. Thus, the issue of scheduling as defined in this work is never addressed by PIER. In the future, it would be interesting to see if a DHT approach could be extended to continuous queries and how such an extension would affect scheduling techniques.

Infopipes at Georgia Tech [30] is another project whose goal is to define and implement a data-

stream processing system. Infopipes provides a new abstraction that simplifies development of distributed data-stream applications. A related project from Georgia Tech presents a distributed, utility-driven, resource allocation algorithm [32]. This algorithm takes into account business utility of operators in order to aggregate them and deploy them on a distributed network of servers. We envision that smart utility-driven scheduling is still performed by the run-time after allocation, since many times the re-allocation of operators may be costly.

Event correlation engines are a special type of stream processing systems that are very popular in enterprise computing [2, 18]. Active Middleware technology (Amit) is one of the leading correlation engines [2]. Amit defines a rich and extensive language and associated run-time for event detection and correlation. In the future, for a correlation system like Amit, the distributed implementation of the run-time could face the same issues of control of queuing delays as outlined in this work.

In enterprise computing, stream processing systems manifest themselves in the notion of enterprise service buses. An enterprise service bus (ESB) is a system responsible for integrating different services. The services connect to the ESB and submit data into the ESB. The ESB is responsible for performing data transformation and delivery of the transformed data to interested parties. The data transformation encompasses business rules that are programmed into the bus. Thus, the ESB helps to achieve better integration among different systems and services. In the context of our work, the business logic inside ESB can be viewed as a streaming computation. The services could be seen as users, and the bus itself as a stream processing system. Currently, there are several proprietary bus implementations avaiable on the market [15, 16, 17, 22, 48, 56] and a couple of open-source implementations [53, 46]. Most ESB deployments today are not distributed. However, this area is experiencing rapid growth and large scale ESB deployment is soon to be come reality. In such an environment, the approaches outlined in this work would become relevant to ESB design and implementation.

There has been a lot of work done in trying to use control theory to design new, fair, adaptive TCP protocols to maximize transmission rates for different sources on the Internet [28, 39]. However, all the control theory has been developed on an end-to-end basis with the assumption that each packet has the same size and router buffering is very limited. For stream processing systems, the situation is different. The messages in the system are of varying size. In addition, the intermediate servers perform computation, which has an effect on future service rates. As a corollary, these servers may store much longer queues than regular Internet routers. The utility is measured in terms of delay rather than throughput. In short, our framework is trying to combine analysis techniques developed by the networking community and extend them to the new area of stream processing.

Another area of related research is the area of Markov chain approximation algorithms. There are several methods based on aggregation-disaggregation of Markov chains [7, 50]. Some of these approaches allow iterative approximation of a steady-state in a product form without storing the

Q matrix in memory. However, the structure of our Q matrix is not suitable for such approaches. Instead, our approximation exploits the unique structure of the Q matrix to achieve limited scalability.

An alternative way to find steady-state probabilities of a Markov process is to use approximate mean-value analysis to determine the approximate expected queue sizes under our SMBS scheduling algorithm [7, 37, 41]. The work on approximate mean-value analysis is too extensive to be described here; Bolch *et al.* [7] provide a good introduction to this technique. However, to the best of our knowledge, our Markov process lacks both global and local balance to apply any of the existing algorithms.

Also, several classical optimization problems for scheduling over queuing networks have been developed over the years [7, 45]. However, all these optimizations define cost for using resources rather than cost of delivering information in a timely manner. It would be interesting to see how optimization over several dimensions (*i.e.*, taking into acount both resource and delay) could be combined. The area of stochastic control offers a good set of tools that could be used to tackle this problem. In the future, we will also study the applicability of the algorithm proposed by Sennot *et al.* [45] to our work.

Chapter 8

Future Work

There are several interesting directions for future work. For the static mean-based scheduling algorithm, more work needs to be done to create better approximations of the Markov model. The current approximation only works well for up to five queues. It would be useful if a closed-form solution could be found. Such a closed-form solution would be a novel result, even outside of this work. Our current framework makes it easy for future approximation algorithms to be plugged in. More study of SMBS behavior in a distributed environment is also needed, since the current solution has only been tested under limited conditions.

For the dynamic queue control algorithm, the key direction for future work focuses on the design of better feedback for local scheduling algorithms. The feedback should go beyond expected service times of downstream computations, and should provide a more accurate reflection of network conditions including queueing delays. More importantly, formal control theory tools need to be used to show how robust such feedback would be, such as how quickly the scheduling would respond to spikes in system load. Such an approach has been shown to work well for general networking and we believe it could work equally well for streaming systems [39]. Moreover, we need to address the issues raised in Abadi *et al.* [1], Carney *et al.* [11], Babcock *et al.* [3]. In particular, the impact of message reordering has to be studied more rigorously. Formal bounds on the difference from the optimal solution have to be proved for the cases when message reordering is impossible due to the nature of streaming computation. Also, other quality of service functions presented by Abadi *et al.* should be integrated into our control framework.

Moreover, we need to study the impact of non-trivial flows, *i.e.*, flows that contain splits and joins, on our scheduling algorithm. The joins may be hash joins described by Babcock *et al.* [4] or joins defined over the time scale described in Zimmerman *et al.* [61], Manohar *et al.* [35]. The latter complicates the analysis of the scheduling algorithm considerably. The nature of streaming computation, and in particular the operators with negative selectivity, also has to be studied further in the context of our control problem.

Another interesting and important direction involves integrating the scheduling algorithm with

mapping algorithm being developed by Tian [55]. Clearly, mapping impacts how much scheduling can improve on the system performance at run-time. Similarly, mapping needs to know how flows will perform after they are mapped, which in turns depends on scheduling. One of the potential approaches may involve the mapping algorithm using scheduling as a prediction model to determine prices for particular mapping strategies.

Chapter 9

Conclusions

The area of distributed stream processing will continue to expand as systems for network traffic monitoring, online financial data processing and enterprise system integration become widely adopted. In this thesis, we have presented the problem of cost-based scheduling of streams in a distribubted stream processing system. Unlike previous environments where scheduling has been studied, this new setting creates new challenges such as the need to deal with fluctuating stream statistics and rapidly changing system attributes and the need to conduct scheduling based on quality of service functions.

We proposed two different frameworks for analysis and development of the scheduling algorithms. Our frameworks work with any type of quality of service function, unlike several algorithms proposed in the literature that limit the types of QoS function one may use [11]. The first approach is based on generation of a model of a scheduling algorithm. The model is then used as an evaluation function to find the parameters that would make the scheduling algorithm minimize the global cost of stream processing. In this approach, the model is decoupled from the optimization process. Thus, in the future, the search for better models and optimization strategies can be conducted independently. We have outlined a static mean-based scheduling algorithm (SMBS) that fits this framework. In this approach, the model is based on a Markov process and the optimization strategy uses a genetic algorithm. Although the model evaluation and the optimization algorithm are computationally intensive, the advantage over naïve scheduling is substantial.

Our second approach is based on run-time analysis of the cost of scheduling one stream over another. The costs are driven by the quality of service functions. In conjunction with the costs, feedback is used to provide information about global system conditions. This information adjusts the costs such that the global cost objective is minimized. Currently, this information includes expected service times. In the future, more sophisticated feedback that captures a system's queuing delays will be provided and formally analyzed. However, even with simple feedback, our scheduling algorithm not only outperforms naïve scheduling, but also outperforms the same smart scheduling with no feedback.

We hope that the two frameworks introduced in this work will serve as a good foundation for the development and analysis of future algorithms. These algorithms would utilize more tools from control theory to develop more accurate feedback, resulting in distributed stream processing systems with greater robustness in the face of constant change.

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, M. C. Ugur Centintemel, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *CIDR*, 2005.

[2] A. Adi and O. Etzion. Amit - the situation manager. *VLDB J.*, 13(2):177–203, 2004.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *VLDB Journal on Data Stream Processing*, 2004.

[4] B. Babcock, M. Data, R. Motwani, and J. Widom. Models and issues in data stream systems. In *ACM Symp. on Principles of Database Systems*, 2002.

[5] S. Babu and J. Widom. Continuous queries over data streams. In *SIGMOD Record*, September 2001.

[6] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[7] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains.* John Wiley & Sons, Inc., 1998.

[8] S. Bourbonnais, V. M. Gogate, L. M. Haas, R. W. Horman, S. Malaika, I. Naran, and V. Raman. Towards an information infrastructure for the grid. *IBM Systems Journal*, 43(4):665–688, 2004.

[9] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, April 1994. Special issue on "Simulation Software Development".

[10] R. Bulirsch and J. Stoer. *The Conjugate-Gradient Method of Hestenes and Stiefel in Introduction to Numerical Analysis.* New York: Springer-Verlag, 1991.

[11] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 2003 International Conference on Very Large Data Bases*, September 2003.

[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthya, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[13] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[14] L. Chen, K. Reddy, and G. Agrawal. Gates: A grid-based middleware for processing distributed data streams, 2004.

[15] B. Corp. BEA aqualogic service bus. http://www.bea.com, 2006.

[16] I. Corp. Websphere enterprise service bus. http://www.ibm.com, 2006.

[17] O. Corp. Oracle enterprise service bus. http://www.oracle.com, 2006.

[18] Correl8. Corel8. www.correl8.com, 2006.

[19] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for extracting signatures from data streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 9–17, 2000.

[20] C. Cranor, T. Johnson, and O. Spatscheck. Gigascope: a stream database for network applications. In *Proceedings of ACM SIGMOD*, June 2003.

[21] J. H. et. al. JAMA : A java matrix package. http://math.nist.gov/javanumerics/jama, 2006.

[22] Fiorano. Fiorano SOA platform. www.fiorano.com, 2006.

[23] R. Freund and N. Nachtigal. QMR: A quasi-minimal residual method for non-hermitian linear systems. *Numerical Mathematics*, 60, 1991.

[24] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An architecture for a world-wide sensorweb. In *IEEE Pervasive Computing*, pages 22–33, October 2003.

[25] B.-O. Heimsund. Sparse matrix toolkit. www.mi.uib.no/ bjornoh/mtj/smt, 2005.

[26] J. M. Hellerstein and R. Avnur. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[27] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier, September 2003.

[28] C. Jin, D. X. Wei, S. H. Low, G. Buhrmaster, J. Bunn, D. H. Choe, R. L. A. Cottrell, J. C. Doyle, W. Feng, O. Martin, H. Newman, F. Paganini, S. Ravot, and S. Singh. FAST TCP: From theory to experiments. In *IEEE Network*, January/February 2005.

[29] Y. Jin and R. Strom. Relational subscription middleware for internet-scale. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003.

[30] R. Koster, A. Black, J. Huang, J. Walpole, and C. Pu. Infopipes for composing distributed information flows. In *Proceedings of the 2001 International Workshop on Multimedia Middleware*, 2001.

[31] R. Krishnamurthy, S. Yalamanchili, K. Schwan, and R. West. Share-streams: A scalable architecture and hardware support for high-speed QoS packet schedulers. In *Symposium on Field Programmable Custom Computing Machines (FCCM) IEEE*, April 2004.

[32] V. Kumar, B. F. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *IEEE International Conference on Autonomic Computing*, 2005.

[33] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations, 1999.

[34] S. R. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *CDE Conference*, February 2002.

[35] R. Manohar and K. M. Chandy. Δ-dataflow networks for event stream processing. In *16th IASTED International Conference on Parallel and Distributed Computing and Systems*, November 2004.

[36] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS02*, 2002.

[37] D. Neuse and K. Chandy. SCAT: A heuristic algorithm for queuing network models of computing systems. *ACM Segmetrics Performance Evaluation Review*, 1981.

[38] J. Norris. *Markov Chains.* Cambridge University Press, 2005.

[39] F. Paganini, Z. Wang, J. C. Doyle, and S. H. Low. Congestion control for high performance, stability and fairness in general networks. In *IEEE/ACM Transactions on Networking*, pages 43–56, February 2005.

[40] D. S. Parker, R. R. Muntz, and H. L. Chau. The tangram stream query processing system. In *Proceedings of the 1989 International Conference on Data Engineering*, pages 556–563, February 1989.

[41] D. C. Petriu and C. Woodside. *Approximate mean value analysis based on Markov chain aggregation by composition.* Linear Algebra and its Applications. Elsevier Science Journal, 2004.

[42] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *To appear: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, April 2006.

[43] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proc. 2001 ACM SIGCOM Conference*, August 2001.

[44] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems, 2001.

[45] L. I. Sennot. *Stochastic Dynamic Programming and the Control of Queueing Systems*. Probability and Statistics. Willey, 1999.

[46] ServiceMix. Servicemix enterprise service bus. http://servicemix.org/, 2006.

[47] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An infrastructure for connecting sensor networks and applications. Technical report, Harvard Technical Report TR-21-04, 2004.

[48] S. Software. Sonic enterprise service bus. http://www.sonicsoftware.com, 2006.

[49] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: Scalable peer-to-peer lookup service for internet applications. In *Proc. 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[50] J. C. Strelen. Approximate product form solutions for markov chains. *Performance Evaluation*, 30(1–2):87–110, 1997.

[51] R. E. Strom, D. F. Bacon, A. Lowry, A. P. Goldberg, D. M. Yellin, and S. Yemini. *Hermes: A Language for Distributed Computing*. Number ISBN 0-13-389537-8 in Series in Innovative Technology. Prentice-Hall, 1991.

[52] M. Sullivan and A. Heybey. Tribeca: a system for managing large databases of network traffic. In *Proceedings of the USENIX annual technical conference*, pages 15–19, June 1998.

[53] SymphonySoft. Mule enterprise service bus. http://mule.codehaus.org/, 2006.

[54] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 321–330, San Diego, California, United States, 1992. ACM.

[55] L. Tian. Resource allocation in streaming environments. Master's thesis, California Institute of Technology, 2006.

[56] TIBCO. TIBCO enterprise service bus. http://www.tibco.com, 2006.

[57] P. Tucker and D. Maier. Applying punctuation schemes to queries over continuous data streams. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pages 33–40, March 2003.

[58] R. West, Y. Zhang, K. Schwan, and C. Poellabauer. Dynamic window-constrained scheduling of real-time streams in media servers. *IEEE Transactions on Computers*, June 2004.

[59] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *The 21st International Conference on Data Engineering*, 2005.

[60] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, UC Berkeley, April 2001.

[61] D. M. Zimmerman and K. M. Chandy. A parallel algorithm for correlating event streams. In *19th IEEE International Parallel and Distributed Programming Symposium (IPDPS 2005)*, April 2005.