QoS-aware Middleware to Support Interactive and Resource-Adaptive Applications on Myrinet Clusters

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Sandhya Senapathi, B.E.

* * * * *

The Ohio State University

2002

Master's Examination Committee:

Prof. Dhabaleswar K. Panda, Adviser

Prof. P. Sadayappan

Dr. Pete Wyckoff

Approved by

Adviser

Department of Computer and Information Science

ABSTRACT

Simultaneous advances in processor and network technologies have made clusters of workstations attractive vehicles for high-performance computing. However, current generation clusters without any QoS support are not capable of supporting next-generation applications such as visualization applications in a shared manner. Such applications are *interactive* in nature and therefore require guarantees on response time which cannot be satisfied in the presence of contention from other applications using the cluster simultaneously. Existing cluster interconnects do not possess any mechanism for guaranteeing application performance demands. In addition, there is currently no scheme by which we can determine the amount of system resources required to satisfy demands given in terms of application parameters. In this thesis, a QoS-aware middleware layer that is capable of supporting multiple simultaneous user requests with specific user-level constraints is developed. The proposed layer uses application profiled data taken apriori to establish a concrete relationship between the user-level requirements and the system resources required to satisfy the user demands. Therefore the framework helps to execute a requested job with an efficient allocation of system resources while exploiting the *resource-adaptive* property of next-generation applications. The framework is supported at the system-level by a NIC-based rate control scheme that provides proportional bandwidth allocation for applications executing in a shared manner on such clusters. The complete middleware

is demonstrated on a Myrinet cluster for polygon rendering and ray-tracing applications which form the core of many interactive visualization applications. From the performance evaluation, we are able to demonstrate that the use of the QoS framework helps to execute interactive and resource-adaptive applications in a *predictable* manner, while keeping the allocation of system resources efficient so that the cluster can be used in a shared manner by many such applications. We are able to show that the use of the middleware layer helps applications to obtain execution times within 7% of expected execution times. Without the support of the middleware layer, increases of as much as 117% over the expected execution times were observed for applications executing on the shared cluster.

I dedicate this work to my mother Lakshmi and my father Senapathi

ACKNOWLEDGMENTS

I am indebted to my adviser Dr. D. K. Panda for his invaluable guidance, understanding and help during the last two years. I also thank him for supporting me as a research associate for the past year.

I am grateful to Dr. Pete Wyckoff and Prof. P. Sadayappan for agreeing to serve on my Master's examination committee.

I also wish to thank Prof. Han-Wei Shen for his help and guidance, and his students for their assistance with experiments.

A special mention must be made of Darius Buntinas, and Igor Grobman for spending a lot of time and effort in discussing various technical doubts and questions.

I am grateful to the Graduate School for awarding me a University Fellowship for the first year of my graduate study.

Last but definitely not the least, I would like to express my thanks to my dear friend Ranjesh whose unconditional love and support has helped me through many a rough patch.

VITA

June 18, 1979 ..... Born - Chennai, India

May 2000 ..... B.E. Computer Science,
Anna University, Chennai, India

Sep 2000 - Aug 2001 ..... Graduate Fellow,
Ohio State University.

Sep 2001 - Present ..... Graduate Research Associate,
Ohio State University.

PUBLICATIONS

Research Publications

Sandhya Senapathi, Dhabaleswar K. Panda, Don Stredney, Han-Wei Shen, "A QoS Framework for Clusters to Support Applications with Resource Adaptivity and Predictable Performance", *Proc. of Int'l Workshop on Quality of Service 2002* , May 2002.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

High Performance Computing : Prof. D.K. Panda

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Clusters of workstations have emerged as powerful computing tools as a result of recent advances in high-speed networking technology and increasing processor speeds[1]. Such clusters are becoming increasingly popular for providing cost-effective and affordable computing environments for the computational needs of a wide-range of applications[2]. Traditionally applications targeted for clusters have primarily included compute-intensive jobs such as scientific and engineering simulations. However, with the growth of modern networking and the Web technologies, a new generation of applications is being targeted for clusters. These applications include data mining, imaging, virtual reality, multimedia servers, distributed visualization, and tele-medicine[3]. Such a system where a cluster is used in a shared manner by next-generation applications is known as a cluster-based server.

1.1 Cluster-based Servers

A cluster-based server needs to have the ability to provide service to a large number of clients, and at the same time guarantee requested performance for each of the clients. Consider the following example of a visualization application shown in Figure 1.1. For such an application, a client typically needs to access data from a

local file system or from a remote large scale repository, perform computation on this data on a local cluster so that the data can be rendered, and finally transmit the data from all the compute nodes to a front-end node, where the image can be viewed by the client. The actual computation needed for visualization is performed on the nodes of a cluster in parallel. Then this data has to be transmitted to the front-end node of the cluster, where it can be directly viewed by a local client or transmitted over a LAN or a WAN to be viewed by a remote client. This process involves communication both between the nodes of a cluster, and between the cluster and a remote client over the links of a WAN, and therefore QoS guarantees are required for both the WAN links as well as the internal links of the cluster[4]. In an actual scenario of a cluster-based server, many such client requests may be executing simultaneously on a shared cluster, and each of these requests will require some kind of guarantees for their execution.

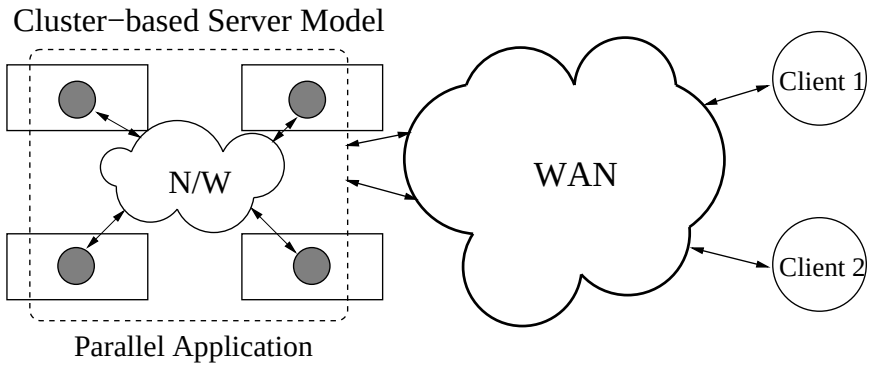


Figure 1.1: Client applications accessing a cluster-based server

The next section takes a closer look at new classes of applications targeted for clusters and their primary differences from traditional applications.

1.2 Next-generation Applications and their Properties

Traditional scientific computing and engineering applications are usually submitted to clusters as batch jobs, where response time is not a critical issue, and different applications can share nodes of a cluster with no significant difference in performance. However, this does not hold true for next-generation applications where *response time* plays a critical part in performance. These applications are different from scientific and engineering applications in that they possess two important properties of *interactivity* and *resource adaptivity* .

1.2.1 Interactivity

Next-generation applications are *interactive* in nature, and therefore the client needs a time limit on the response time within which the execution has to terminate. The execution of such applications is often iterative in nature, where the client gives the request for a certain step in execution, and based on the result that he had obtained, makes the decision about the next step in execution. Due to this property of interactivity, such applications require *predictable* execution time, where the client can be given guarantees that the execution will terminate within a certain time-limit, the value of which is usually specified by the client himself. Existing resource management schemes in shared clusters have no way of giving assurances about execution times to client applications, especially in the presence of other applications simultaneously using the cluster. Therefore such applications require a middleware layer that would be able to make guarantees regarding response time and other application parameters and keep these guarantees even in the presence of contention from other parallel applications.

1.2.2 Resource Adaptivity

Another property that these applications usually possess is that of *resource adaptivity* . Resource adaptivity refers to the ability of such applications to modify their application parameters based on the amount of available resources. For example, consider the same visualization example that was given earlier in this section. The client can choose to view an image at different resolutions and different sizes and at different frame rates, depending on the amount of system resources currently available to him. Resource-adaptivity helps the client to establish a trade-off between these metrics so that the client can choose parameters like image size, image quality and response time that would ensure the best possible execution with the available system resources.

Thus, supporting current and next-generation applications on shared clusters while preventing the inevitable decrease in performance caused by such an approach poses several challenging research problems.

1.3 The Problem

Currently clusters that are used in a shared manner by such applications have no mechanisms of guaranteeing performance demands in the face of network contention from other applications. The challenge therefore, is to design a suitable QoS-aware middleware layer for clusters that can support the execution of interactive and resource-adaptive applications in a shared mode together with other such applications. The requirements for such a middleware layer are enumerated as follows:

1. The framework must be able to make a translation from given application parameters to system resources. For efficient resource allocation, there is a need to

determine exactly what amount of system resources is necessary for satisfying specific application demands.

2. The resource-adaptive property of applications can be used to determine the set of application parameters most suited to the available system resources, when conditions for parameters are not stated by the application.

3. Once a translation has been made, there must be a system-level mechanism that can provide co-ordinated access to system resources such as processing power and network bandwidth. The resource allocation scheme must provide a method by which applications can reserve system resources and guarantee that the resources reserved by an application are for its exclusive use only.

4. The framework must be able to admit as many client requests to a shared cluster as possible while taking into account their QoS constraints, and be able to execute the jobs while delivering the QoS requests of the admitted applications.

1.4 Our Approach

Figure 1.2 shows the structure of the proposed QoS-aware middleware layer. The main components of the middleware are the request handler, the QoS translator, the resource allocator, and the profiler.

The profiler maintains profiled data of an application which characterizes application execution time with respect to different application parameters and system parameters such as number of assigned nodes and network bandwidth reserved between these nodes. The QoS translator makes use of the profiled data to make the best possible match between application-given parameters and system resources and

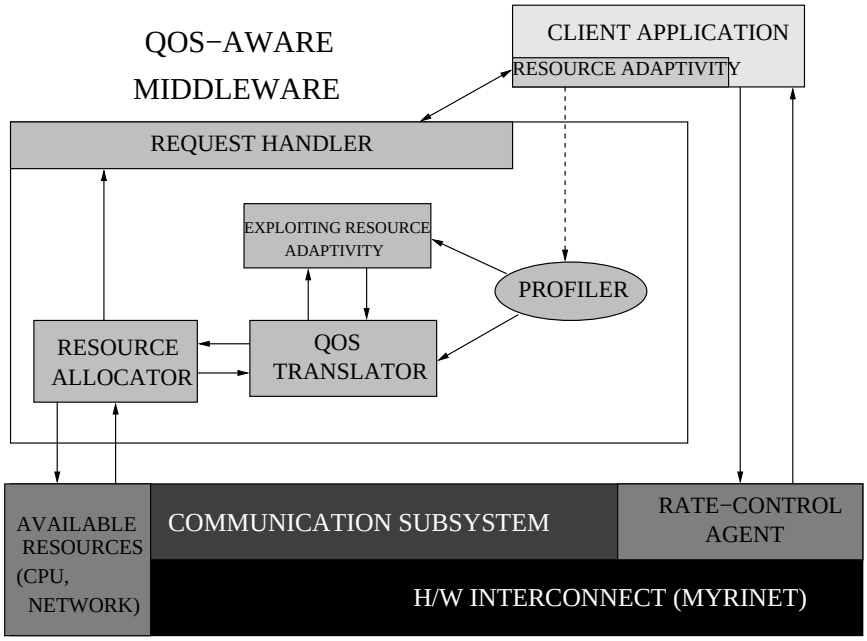


Figure 1.2: High-level overview of the proposed middleware layer

translates the given application parameters into system resource requirements. The decisions regarding actual allocation and de-allocation of the translated resource requirements are made by the resource allocator, based on the amount of resources available in the system.

The system level support for the middleware layer is handled by a NIC-based rate control scheme that was developed as part of an earlier project. This scheme provides proportional bandwidth allocation to application flows and guarantees that applications have exclusive use of the network resources that they have reserved. Therefore by the use of the QoS-aware middleware layer together with the rate control scheme, a unique framework has been defined that promises to support interactive and resource-adaptive applications on a shared cluster without loss of performance.

It has been determined through performance evaluation that the use of the framework has provided application execution times that are not more that 7% higher than the expected response times, while in the absence of the middleware layer, we obtain execution times that are as much as 117% higher than expected execution times.

The rest of the thesis is organized in the following manner. Chapter 2 provides a brief summary of the rate control scheme that provides support for the middleware layer at the system level. Chapter 3 shows a detailed look at the structure and function of the various components of the middleware layer. Chapter 4 gives experimental results that validate both the rate control scheme and the middleware layer and demonstrate the advantages of using such a mechanism. Chapter 5 concludes the thesis and also gives a look into directions of possible future research.

CHAPTER 2

SYSTEM-LEVEL SUPPORT FOR THE MIDDLEWARE LAYER

The support for the QoS-aware middleware layer at the system level is provided by a NIC-based rate control mechanism that provides proportional bandwidth allocation to requesting applications. The rate-control mechanism was implemented[12, 15] by another member of the research group on Myrinet/GM[5, 16]. For completeness, we provide a brief overview of this concept, its implementation in the GM layer, and its extension to MPI[9] in Sections 2.1 and 2.2. Next we provide the enhancements that were added to the rate-control mechanism in the GM layer.

2.1 NIC-based Rate Control

A message stream between applications is denoted by the term *communication flow*. Every communication flow has a well-defined source and a well-defined sink. The end-points of the flow are logical and there may be several communication end-points on the same physical network node. The flows are then multiplexed over the physical link. An application is required to make QoS reservations for each of the communication flows originating from it. Communication flows are regulated by controlling the rate at which data is transferred into the network interface and sent into

the network. Such a rate-control scheme can be implemented at the host, but a NIC-based scheme is preferred since it allows a finer granularity of control, because the NIC deals with frames whereas the host will deal with messages. Since the QoS mechanism is implemented by the firmware on the NIC, which is loaded by the operating system, it can be trusted, and no other policing mechanism is required. Thus, this solution is particularly attractive in that it requires no additional hardware components, or changes in commodity components.

An example of the rate-control mechanism is shown in Figure 2.1. Every NIC connecting to the network has the QoS features uploaded on it. Applications that are executing at the hosts can reserve certain amounts of network resources in terms of network bandwidth, and be guaranteed that the reserved bandwidth is available only to them. This scenario is similar to the case in which every executing application has its own independent virtual network that is unaffected by interference due to the communication flows of other processes on the cluster.

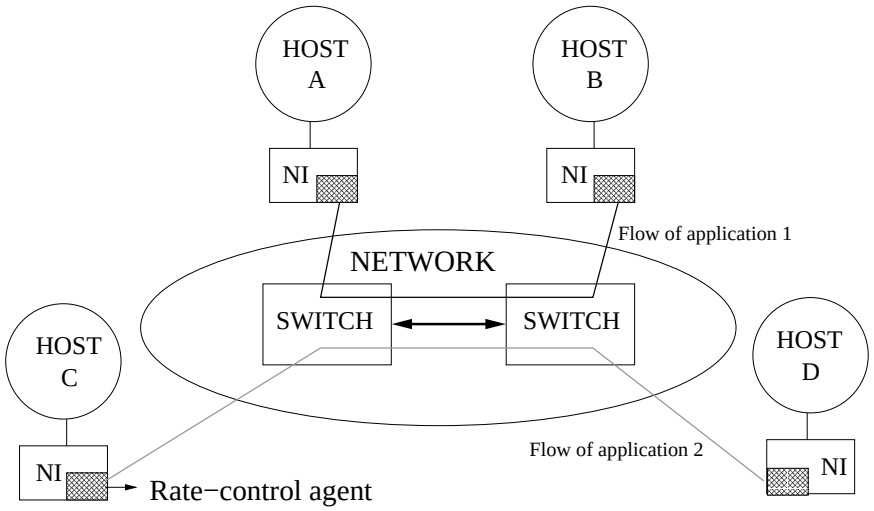


Figure 2.1: Applications sharing the network are guaranteed a certain pre-specified share of the network resources

As shown in Figure 2.1, though the flows of application 1 running on Hosts A and B, and application 2 running on Hosts C and D share links between the switches, there will not be any interference between them if a certain amount of bandwidth is reserved by each communication flow (not exceeding the total capacity of the link), and the rate of injection of each flow into the network is controlled. The bandwidth value given by the application is mapped to a parameter known as the *Inter-Dispatch Time* (IDT) and passed to the rate-control agent at the NIC. The IDT value associated with every flow can be defined as the minimum interval that can elapse between the consecutive injections of the packets from that flow into the network interface.

At any given time, let f_1 to f_n be n communication flows sourced at a particular node. Let their corresponding IDT values be IDT_1 to IDT_n. The actual rate control algorithm uses another set of parameters called *Next Dispatch Time* (NDT) which specifies the absolute time before which a packet from a given flow should not be dispatched. The NDT value of a flow is initialized to the current time when a message send is posted for that flow. Let the NDT values of the communication flows be NDT_1 to NDT_n.

Let t be the time and j be the communication flow.

The rate control algorithm works like this:

NDT_j = min (NDT_1 to NDT_n)

If NDT_j <= t

Dispatch packet from flow j

Update NDT_j to $NDT_j + IDT_j$

Thus for every flow we have to maintain IDT and NDT information at the NIC and these values will be used in deciding the next packet to be transferred into the

network. Assume that we can send out packets at most once every T time units. This T then corresponds to the peak achievable bandwidth B_{max}. Table 3.1 shows a working example of the rate control algorithm in the presence of two flows from sources A and B, each requesting IDT values of 2 and 3 respectively. At time t = 0, the NDT values of both A and B are equal to the current time and both can be sent. We assume, without loss of generality that A is selected as the next flow to be serviced. Once a packet from $flow_A$ is dispatched, its NDT value is incremented by $IDT_A = 2$. At time t = 1, $flow_B$ has the smallest NDT value, and the value of NDT_B is also less than the current value of t. Thus at time t = 1, $flow_B$ is serviced and NDT_B is updated to $NDT_B + IDT_B$. Again at time t = 2, A is ready to be serviced again. The algorithm proceeds in similar steps. It should be noted that at time t = 5, the NDT values of both A and B are greater than the current time, and no packet is dispatched.

t	0	1	2	3	4	5
NDT(A)	0	2	2	4	4	6
NDT(B)	0	0	3	3	6	6
Flow serviced	A	B	A	B	A	-

Table 2.1: Working Example of the Rate-Control Algorithm

2.1.1 Implementation Details

The rate control mechanism[12, 15] was implemented in the GM messaging layer[16] over Myrinet networks[5]. The Myrinet NIC is controlled by firmware called the Myrinet Control Program (MCP). The MCP consists of four state machines: SDMA, SEND, RDMA and RECV, that take care of message sends and receives. Message sends and receives therefore do not involve the operating system, but take place by

direct interaction between the client and the MCP through DMA of data. Additional QoS information for a flow was added to the MCP so that the NIC maintains the IDT values for various flows. IDTs are specified in terms of clock ticks of the *Real Time Clock* (RTC) available on the LANai processor on the Myrinet NIC. The modified SDMA state machine maintains information about the validity of the flow and its NDT and IDT values. If a send has been posted for a flow, then the flow is declared to be valid. Whenever a send token is inserted as the first sendable token for a flow, the flow is validated and its NDT value is updated to the larger of RTC and the existing NDT value. Packet scheduling in the NIC is now based on NDT values. To search for the next packet to be sent, the SDMA state machine finds the flow with the minimum NDT among all valid flows. If the RTC is larger than the NDT value for this flow then the flow is serviced using its first sendable token and its NDT is incremented by its IDT value. This procedure is followed every time a packet has to be sent.

2.1.2 Enhancements to the Rate-Control Agent

Every time the SDMA state machine has to send a packet, it has to first search through the array of flows, and find the one with the smallest NDT value. Therefore the search for the smallest NDT lies on the critical path of sends, and would introduce delay before every send, consequently reducing the maximum rate at which packets can be sent out. In order to avoid this, the search for the next sendable packet can be done before the actual send takes place so that when the SDMA state machine comes to the point where it has to select the next sendable flow, the information is already available so that access can be made instantly. The index of the flow that has the least

NDT value and has a packet to be sent is maintained in a *least_NDT* variable. The search is actually done *after* a packet is selected for sending and its DMA is started, so that the search occurs parallel with the DMA. If a new flow is created and a packet is added for the new flow between the time the search is done and the next packet is to be sent, the value of the least_NDT variable is updated to reflect the addition of the new flow, if the newly validated flow has a lower NDT value. This ensures that the *least_NDT* variable always has the correct value. This enhancement increases the maximum rate at which packets can be sent out by removing all unnecessary delays from the critical sending path of the SDMA state machine.

2.2 Programming Model Support

So far we have described the incorporation of the QoS framework at the GM level for providing rate control. But GM is only a low-level communication layer for the network. For ease of programming and portability, support has to be added at the application level. A familiar programming model, the Message Passing Interface, was modified, since it is in widespread use and freely available.

2.2.1 The Message Passing Interface

MPI[9] is a message-passing library that offers a range of point-to-point and collective interprocess communication functions to a set of single threaded processors executing in parallel. All communication is performed within the definition of a communicator. A communicator is a group of processes that are communicating with each other, in which each process has a unique id between 0 and N-1, N being the number of processes in the communicator. MPICH is a freely available, portable implementation of MPI. The mechanism for achieving portability is a specification called the

Abstract Device Interface(ADI). All MPI functions are defined in terms of a set of basic MPI communication primitives, which are implemented in the ADI layer. This layer uses message-passing functions native to the underlying system. MPICH-GM is therefore an implementation for Myrinet clusters that uses GM as the underlying message-passing system[17]. As Figure 2.2 shows, the ADI layer in MPICH-GM uses GM primitives to implement MPI primitive functions in terms of which more complex MPI operations are defined.

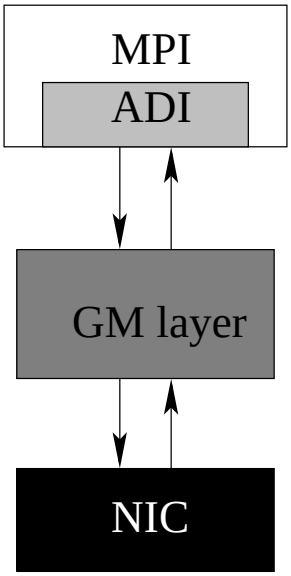


Figure 2.2: The Message Passing Interface

2.2.2 Support for QoS at the MPI Layer

To use the QoS features that GM provides in a standard-compliant fashion, we used the *Attribute mechanism* provided by the MPI layer. MPI provides functions to set and get the values of attributes for a communicator[10]. An attribute is identified by an integer, a *keyval*, and its value may be of any arbitrary type (void * in C).

When used with special *keyval* values that signify that the value is related to QoS parameters, the *set attribute* function call in MPI is mapped to the GM function for requesting bandwidth allocation or deallocation. Process ids are converted into appropriate GM node and port identifiers. Also, since the MPI layer incurs additional overhead, bandwidths achieved at the MPI level are lower than raw GM bandwidths. The *get attribute* call can be used to determine if a request made using the *set attribute* function call is accepted or denied. The bandwidth value requested can also be changed dynamically during the execution of the application by using the same attribute function calls.

Chapter 3 describes how the basic rate-control mechanism is used with a QoS-aware middleware layer to provide guarantees for user-level parameters to client applications.

CHAPTER 3

QOS-AWARE MIDDLEWARE LAYER

Interactive applications such as visualization demonstrate the property of resource adaptivity, in that they can be run with different parameter values based on the amount of resources that are available. For example, a visualization application can be executed with different image sizes. This property of resource adaptivity is useful when executing such applications on a shared cluster as the application can be executed with the parameters that match most closely with available processors and network resources. Applications that possess the property of resource adaptivity can be allocated resources that are optimal for execution with certain parameter values. The optimal resource allocation strategy requires stored profiled data related to known execution runs of the application. The stored data gives the relationship between system and user-level parameters.

3.1 Overview of the Middleware

Figure 3.1 shows the structure of the proposed QoS-aware middleware layer. The main components of the middleware are the request handler, the QoS translator, the resource allocator, and the profiler.

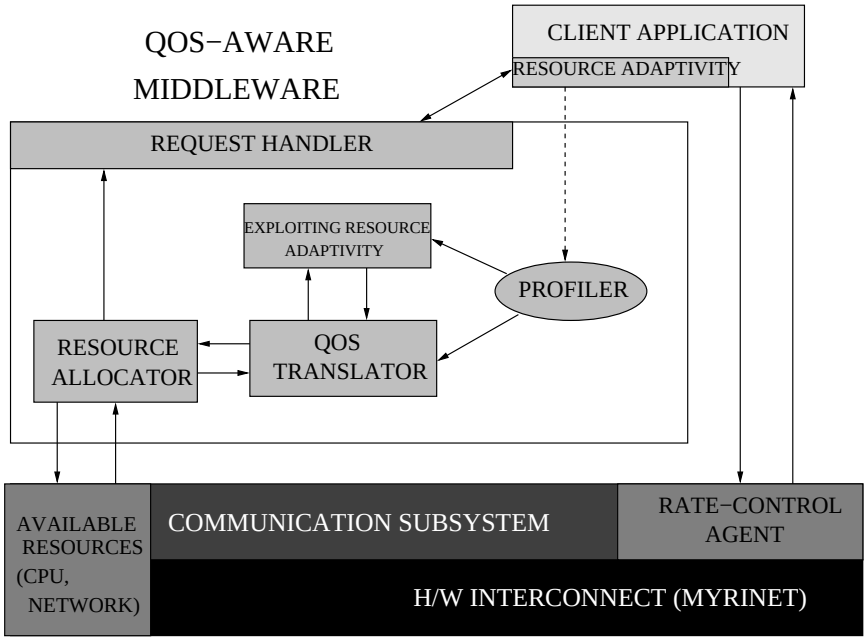


Figure 3.1: High-level overview of the proposed middleware layer

The profiler maintains the profiled data of an application taken apriori. This data characterizes application execution time with respect to different application parameters and system parameters such as number of assigned nodes and network bandwidth reserved between these nodes. Due to the resource adaptivity shown by the application, it can be executed with different application parameters, with varying system requirements, and the profiled data captures the behavior of the application with different application parameters and varying system resource allocations. For the visualization example that we chose before, one can take profiled data for different image sizes and image qualities with different values of system parameters assigned.

The request handler is the component of the middleware that interacts with client applications. This is a thin high-level layer whose functionality is to read incoming application requests and extract application parameter values from them.This information is then passed along to the Quality of Service (QoS) translator.

The main component of the middleware is the QoS translator which takes user-specific parameters and maps them into corresponding system resource values. The demands given by an application are in terms of application-specific parameters, for example, image size, image quality, and response time for visualization applications. For the purpose of translating these values into system resource numbers, the QoS translator uses the profiled data stored about the application. From this data, the QoS translator can determine the amount of system resources required to satisfy application parameters. Though the QoS translator may determine than a certain request can be satisfied by a certain set of systems resources (processors and network bandwidth), it may be the case that the required amount of resources may not be available for use.

Therefore, it is the job of the resource allocator to determine whether the system resource requirements of a request can be satisfied. The resource allocator analyzes requests for allocation from the QoS translator and determines whether they can be granted, based on existing reservations and available resources. The resource allocator also keeps track of current reservations in the system, allocations and de-allocations. This process of determining a set of system resources that can be allocated is iterative and can go back and forth between the QoS translator and the resource allocator. For example the QoS translator may determine that there are several possible allocations of system resources that satisfy the given application parameters. However, it may

be the case that not all these possibilities can be satisfied by the amount of available resources in the system. Thus, it may take several tries between the QoS translator and the resource allocator before a request can be satisfied.

Once the resource allocator determines that a certain set of resources can be provided for the application, it informs the request handler, which in turn passes the result back to the waiting application.

The system level support for the middleware layer is handled by the NIC-based rate control scheme that was explained in Chapter 2.

We explain the different components of our middleware with running examples of two visualization applications: Ray-tracing[14] and Polygon Rendering.

3.2 Profiling Client Applications

The QoS Translator needs previous performance data on an application to determine the set of system resources required to satisfy user-given application parameters. For this reason, we need to store profiled data on an application. This profiled data stores the execution times obtained for the application for different user parameters, and for different system parameters, and therefore can be used to match application parameters against system parameters and vice versa. The profiling information for the polygon rendering application maintains the execution pattern of the application given 2 different image sizes, and a range of bandwidth assignments from the minimum bandwidth required by the application for execution to the maximum possible bandwidth that can be supported by the network on different numbers of nodes. The profiled information for the ray-tracing application can be similar but for an added parameter of the *interleaving factor*, which determines the quality of the final

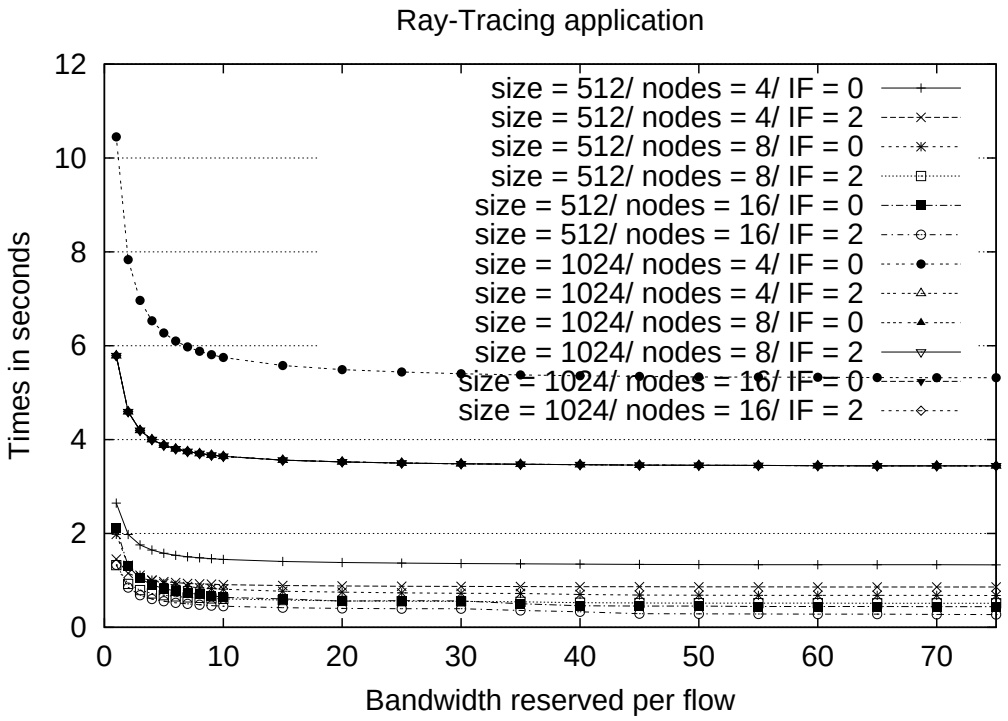


Figure 3.2: Profiling information maintained for the ray-tracing applications with bandwidths denoted as reserved per flow

image. The smaller the value of the interleaving factor, the better the image quality. Such profiling helps us to convert application parameters into system resource requirements. For example, it provides a means of determining the number of processors and the amount of network bandwidth required between these processors for rendering an image size of 512x512 under 2 seconds. In case of applications where no hard conditions are given, the profiled data can also determine the set of application parameters that best fit the available system resources.

Figures 3.2 and 3.3 shows the profiling information obtained on a 4 to 16 node cluster and stored for the ray-tracing and rendering applications, respectively.

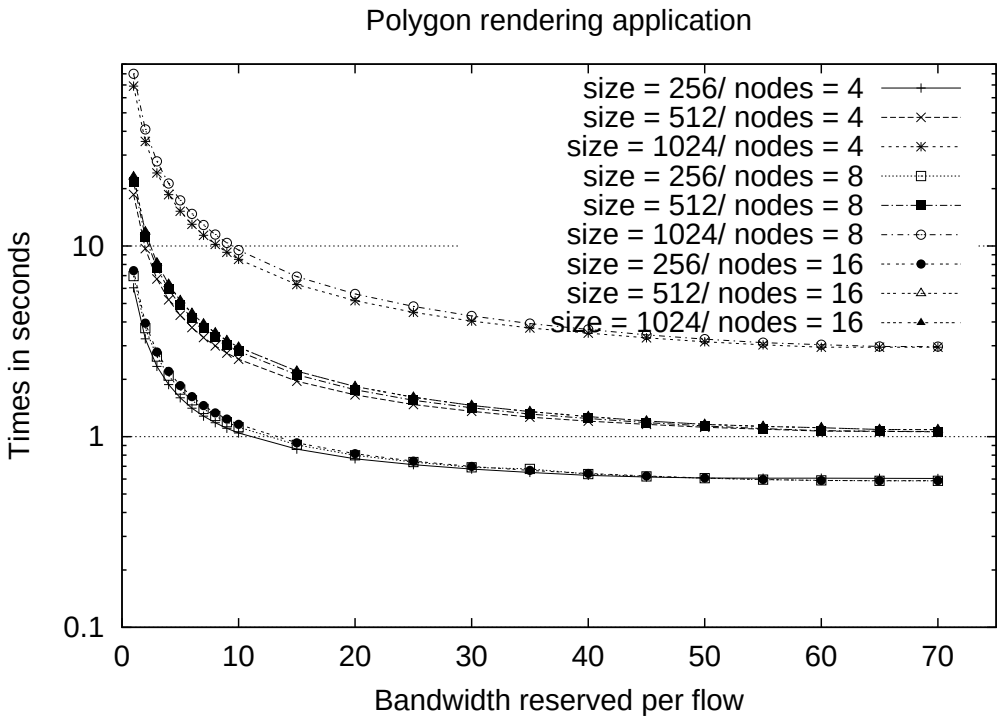


Figure 3.3: Profiling information maintained for the ray-tracing and polygon rendering applications with bandwidths denoted as reserved per flow

3.2.1 Ray-Tracing Applications

In Figure 3.2 there are 12 graphs, for image sizes of 512, and 1024, for interleaving factors of 0 and 2, and take on 4, 8 and 16 nodes. The x-axis measures the bandwidth assigned per flow, and the y-axis measures the time taken for execution, for a set of parameter values. When an application is executed on 4 nodes, and each of the nodes has communication flows to two of the other nodes, as is the case for the example applications that we have chosen, there will be 2 outgoing and 2 incoming flows on the single link from a node, making 4 flows in all. The bandwidth measured on a link is given by the total amount of bandwidth taken by all the communication flows on

that link. Therefore, for such a situation, when the bandwidth assigned per flow is 8 MBps, the actual bandwidth per link for 4 nodes is 8 x 4 = 32 MBps. In the case of 8 nodes, there are 3 outgoing and 3 incoming flows, and the link from a node to a switch is occupied totally by 6 flows.

3.2.2 Polygon Rendering Applications

Figure 3.3 shows the profiling information obtained and maintained for the polygon rendering application.

The graph has 9 curves, for executing the rendering applications with image sizes of 256, 512, and 1024, on 4, 8 and 16 nodes. The x-axis again measures the bandwidth assigned per flow, and the y-axis measures the time taken for execution, for a set of parameter values.

These graphs clearly show the resource-adaptive property of such applications, by which the application can execute with different user parameters and each combination of these parameters require a different amount of system resources to achieve a certain response time. For example, for the ray-tracing applications, to achieve a response time of atmost 2 seconds for an image size of 512x512 with an interleaving factor of 0 requires atleast 2 MBps per flow on 4 nodes, and 1 MBps on 8 and 16 nodes. To achieve the same response time for an image size of 1024x1024 with the same interleaving factor, the application requires atleast 15 MBps per flow on 16 nodes, and the response time cannot be satisfied for the given application parameters on 4 and 8 nodes. Therefore, if the application gives hard conditions for the response time as 2 seconds or less, and for the interleaving factor as 0, and there is less than 15 MBps available on some of the 16 nodes of the cluster, using the above information,

the scheduler can deduce that the only option is to execute the application with an image size of 512x512 on 4, 8 or 16 nodes. Without the use of this scheduler, an application in the above conditions might execute with the larger image sizes, and suffer as a consequence, even though image size is apparently not a hard condition for the application. In a similar way, the profiler can maintain information for other applications that can be run on the cluster.

3.3 Exploiting Resource Adaptivity using the Quality of Service(QoS) Translator

The QoS Translator is involved in the conversion of application specific parameters into system resource requirements. The working of the QoS Translator is illustrated in the following section by iterating through the steps taken for a sample client request.

Consider the following example of a user request for the ray-tracing application. Let us assume that a client request imposes a performance bound on the execution time for the ray-tracing application and requires a frame rate of 1 frame per second. Therefore, the time to render a frame has to be less than 1 second. The assumption that an application can specify such a bound is valid because such visualization applications usually require a certain number of frames to be processed per second.

Figure 3.4 has the same data as the profiled data for ray-tracing applications shown earlier, but shows only the pertinent graph curves, namely the lines that lie below the application given limit of one second.

The shaded area shows the range specified by the application. In this case, since the application has not specified any lower time limit, it is implicitly assumed to be 0.

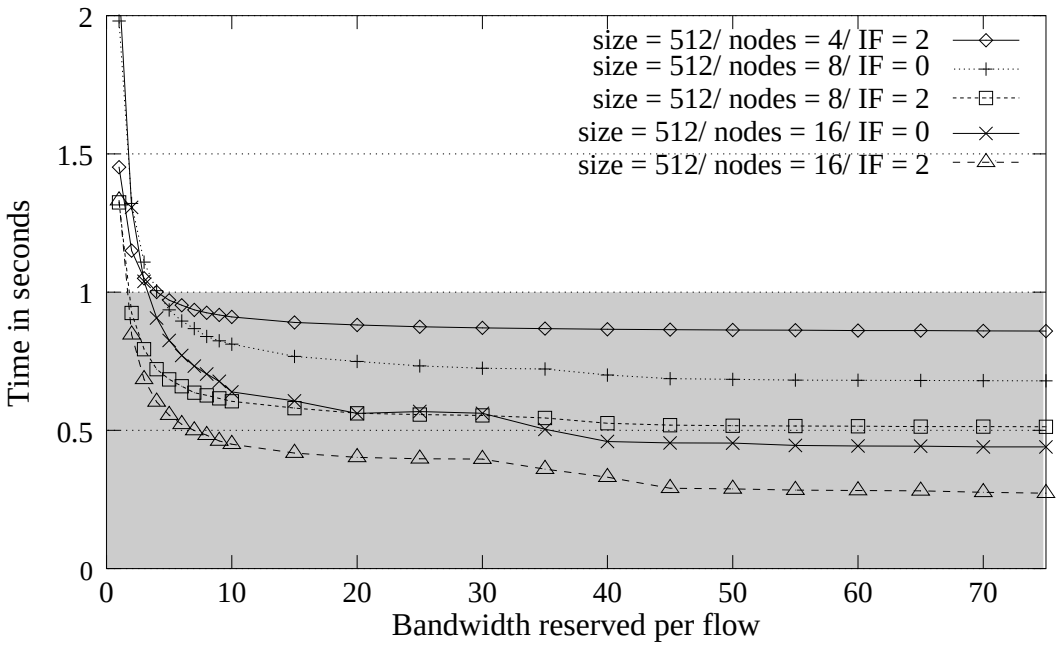


Figure 3.4: Graph-lines that match the application condition of execution time of 1 second or less

The scheduler can assign 4, 8 or 16 nodes to the application, with different bandwidth allocations. Because the time constraint specified by the application is less than 1 second, and since all graphs with points below 1 second have image size of 512x512, the image size that can be allocated to the application has to be 512. At this point, the translator checks to see whether the application has given any requirement for the image size or not. If the application has not specified the image size, or if the image size that it has given is equal to 512, then the translator goes ahead with trying to narrow down the set of points from which possible system allocations can be done. If the application has specified an image size and it is bigger than 512, then the translator deduces that this request cannot be satisfied, and returns the result

to the scheduler, which can then inform the application that its request cannot be satisfied.

For the current example, let us assume that the application has not specified any particular image size, and the selected set of graph curves remains valid. Also, let us assume that the client application desires an interleaving factor of 0.

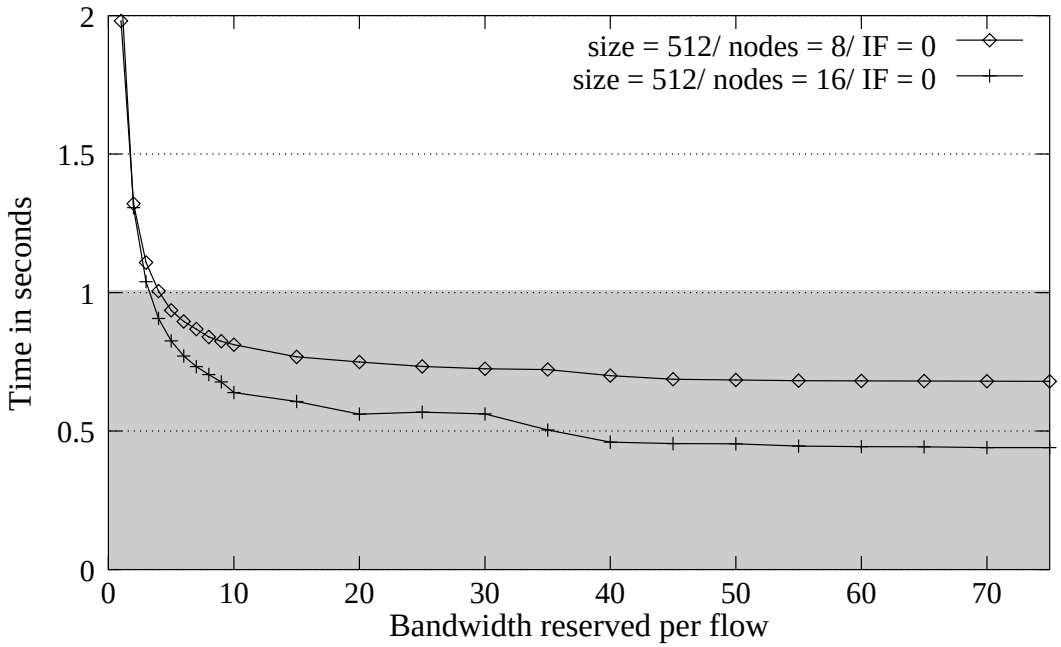


Figure 3.5: Graph-lines that match also the second condition of interleaving factor of 0

Figure 3.5 shows the new set of points that satisfy the given criteria so far, namely that the rendering time for a frame must be less that 1 second, and the interleaving factor must be 0.

Another interesting point can be deduced from the graphs. In order to provide the application with the best possible performance, the first and naive solution may

be to assign the maximum bandwidth possible to the application for its given constraints. On a closer look at the graphs, it can be seen that after a certain point reservation of more bandwidth does not really improve the execution time of the application. Therefore for every graph, we can define a *knee-point* value that gives the bandwidth sufficient to guarantee good application performance for the given parameters. Formally, this knee-point value can be defined as the bandwidth value for which the overall execution time is greater than the execution time for the maximum bandwidth allocation by a percentage value under 5 percent. The profiler stores this knee-point value for all the graphs, and in appropriate situations, will send the bandwidth corresponding to the knee-point to the resource allocator for consideration.

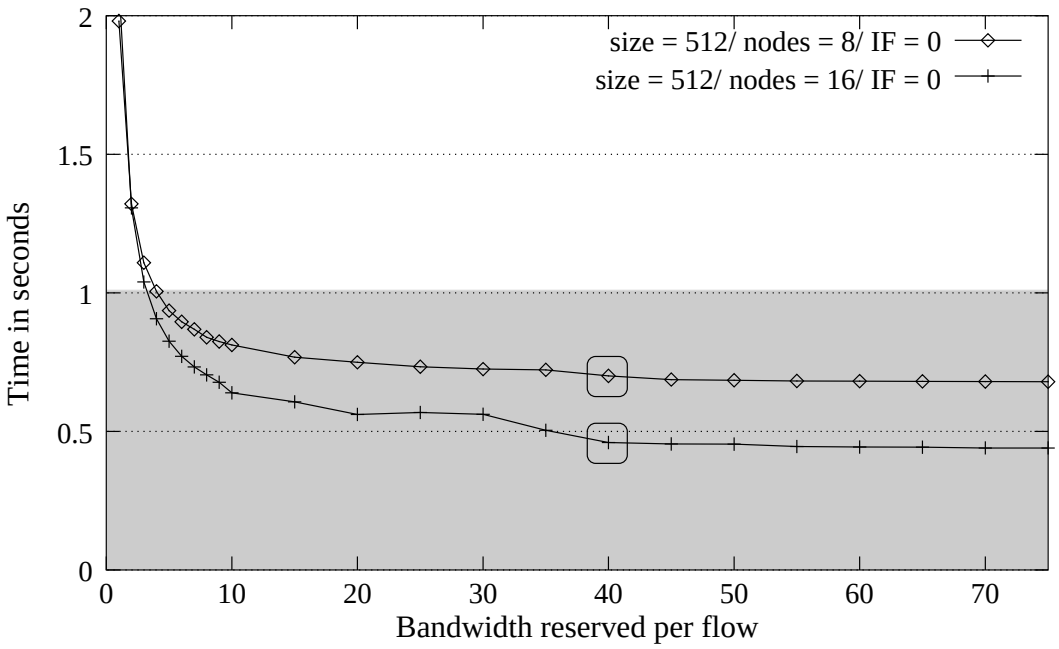


Figure 3.6: Knee-points of graph-lines that match all given application conditions

For the above example, Figure 3.6 shows the two graphs with the knee-point of each graph marked.

For each of the curves that satisfy all the given conditions of the application, the translator also keeps track of the knee-point.

The final set of points that satisfy the given conditions is the set of points on each graph line from the smallest bandwidth value that is necessary to satisfy the given time limit until the knee-point values. The smallest bandwidth values for the 2 graphs are 4 and 5 MBps respectively, and the knee-point value for both the graphs is 40 MBps.

Once the translator has obtained the smallest set of points that can be used to satisfy the application request, it tries to find out if the corresponding system resources can be allocated by conferring with the resource allocator.

For ensuring that under light load conditions, applications are able to use all the available bandwidth, the translator sends bandwidth requests to the allocator in increasing order, that is, the bandwidth corresponding to the least time value is sent first, and then for the next higher time value and so on. This process of trying to find a suitable allocation for an application is iterative in nature, and works as follows. The translator sends each possible alternative to the resource allocator which determines whether the system resource requirements of that alternative can be satisfied. If the allocator finds an alternative that can be satisfied, it sends the result back to the request handler, with the information necessary for allocation, such as the number of nodes, the nodes involved in the allocation, and the amount of bandwidth reserved between these nodes. The request handler then sends this information to

the application which can then send the request to the rate-control agent at the NIC to inform it of the new allocation made.

The translation works similarly for the polygon rendering applications, the only difference being the nature of the profiled data. For the polygon rendering applications, the only application parameter is the image size. Therefore the QoS translator narrows down the set of graphs to be considered using the time limit and image size, if given by the application, and asks the resource allocator for possible allocations using this set of points.

The next section describes the working of the resource allocator.

3.4 Resource Allocator

So far we have examined the way by which the application constraints are used to select the type and number of resources that are allocated to it, in the event that the application demand can be satisfied. However, the selection of resources to be allocated also depends on the number of processors available and the network resources available on the links of the cluster. The resource allocator has to use both the application-given criteria and the table of available resources to select the minimum set of resources that can satisfy the application's demands. The resource allocator maintains information about current reservations in the system, and the bandwidth available on the links between every pair of nodes. It also maintains a list of indices that denote the links with the highest available bandwidth so that allocations are spread evenly across all nodes in the system. As allocations and deallocations are made, this list is also updated. The translator sends the alternatives to the resource allocator in order of best case to worst. Therefore the framework always

tries to allocate resources for a user request so that its performance is maximized. Currently our scheme incorporates a greedy mechanism of allocating resources, where the resource allocator allocates to the demanding application all the resources that are currently available. But this greedy mechanism is augmented by careful use of the knee-point value that was described in the previous section. Instead of allocating all the bandwidth on a link to a requesting application, the resource allocator need only allocate the bandwidth corresponding to the knee-point, so that the application is able to obtain close to peak performance. Thus resources are being utilized efficiently, so that we can accommodate more user requests in the system.

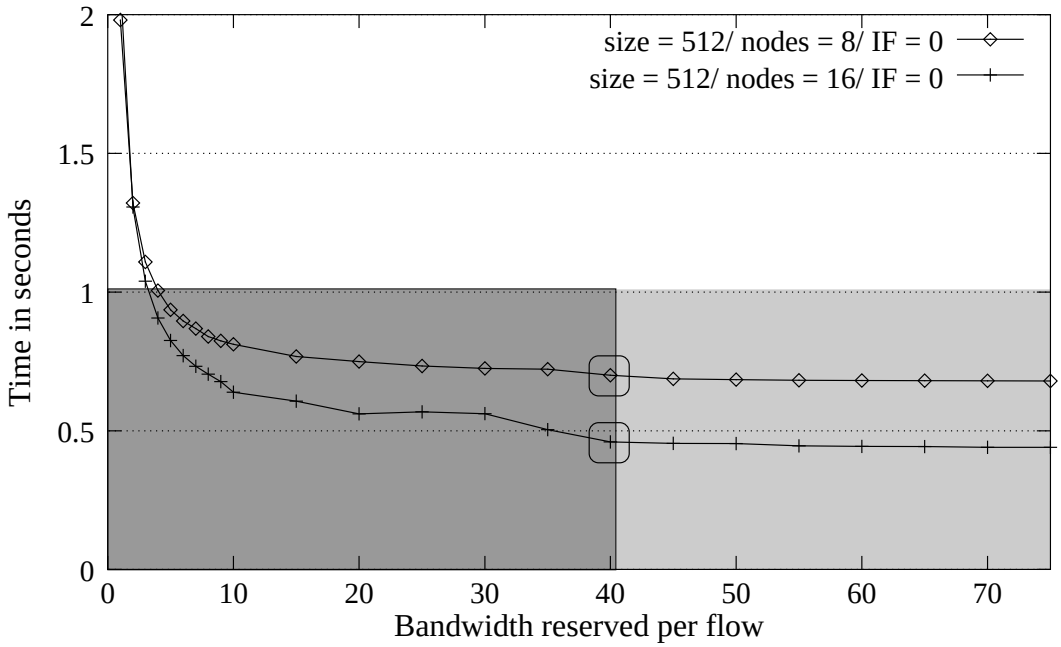


Figure 3.7: Knee-points of 40 MBps give maximum bandwidth allocations for the graph-lines

Figure 3.7 shows the incorporation of the knee-point values into the equation, so that the maximum bandwidth that can be assigned to the requesting application is limited to improve resource utilization in the system. For both the graph lines shown, the knee-point is at 40 MBps.

The working of the resource allocator will now be illustrated by expanding upon the example that we were considering in the previous section. We had marked out the knee-point values denoting the bandwidth that is sufficient to satisfy the performance demands of the application. When the resource allocator is processing this request there can be three scenarios:

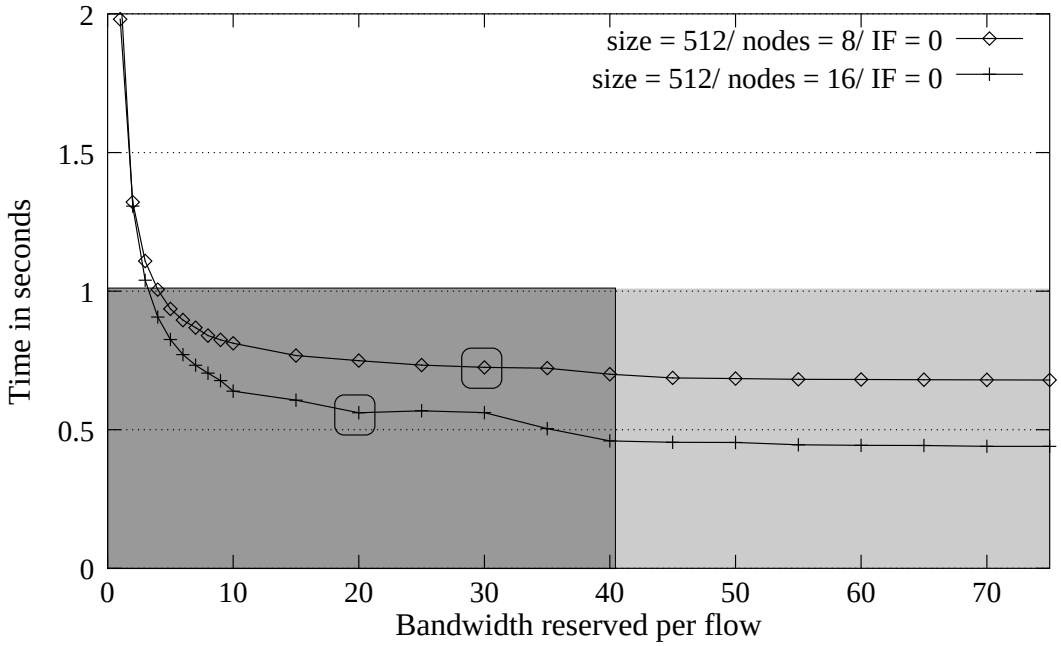


Figure 3.8: Limitations on available network bandwidth posed by existing reservations in the system

1. Even the smallest bandwidth of the shaded region cannot be satisfied. The cluster is currently being used to its fullest capacity and the new application cannot be admitted at this time. The resource allocator notifies the request handler about its decision, and the request handler in turn conveys the result to the application that it cannot be admitted. The application can either try again immediately with a lower performance bound, or can try again with the same request after some time.

2. The maximum bandwidth for atleast one of the set of points in the shaded region can be allocated under the current load in the system. In order to maximize the performance of the application, the resource allocator tries to allocate the bandwidth that would give the lowest time first, then the next lowest time and so on. In actuality, the search is done by a set of parallel threads, one for each graph line. Each thread tries to find the highest bandwidth allocation possible for that graph line, by comparing the bandwidth required for a point with the bandwidth available. Once all the threads have completed their searching, the master thread compares the alternatives selected by each of the thread, and selects the alternative that would provide the application with the least response time. In the above example, there will be two searching threads. Let us assume without loss of generality, that there is no other application currently executing. Therefore, it is possible to allocate the knee-point bandwidth values for both the graph lines, and the two threads return the knee-point values. On comparing the expected response times for both the points, it is evident that the allocation of 40 MBps on 16 nodes would give the best possible time for an image size of 512x512 and an interleaving factor of 0. Therefore the resource

allocator chooses to allocate 40 MBps per flow on 16 nodes for the requesting application.

3. Due to existing reservations, it is not possible to allocate the knee-point value of any of the graph lines. If this is the case, each search thread returns the largest bandwidth value which can be allocated taking into account existing allocations and resources available. Figure 3.8 shows the scenario for the example that we had considered. The maximum bandwidth available is only 30 MBps per flow for 8 nodes, and 20 MBps per flow for 16 nodes. Therefore each of the search threads returns the points marked in the graph. The resource allocator selects the point that provides the best performance, and allocates 20 MBps per flow, on 16 nodes, as shown in Figure 3.9.

In any case, once a decision has been made regarding resource allocation, it is the duty of the resource allocator to change its data structures to reflect this new allocation. The resource allocator also keeps track of the links that have the highest bandwidth so that allocations can be made evenly across all nodes, and not concentrated on a set of nodes. It modifies this data also to reflect the new allocation made. Finally it also notifies the main request handler of its final decision.

The next section gives the formal algorithm describing the working of the QoS translator and the resource allocator, and the interaction between them.

3.5 Formal Algorithm

We now provide a formal algorithm that describes the combined working of the QoS translator and the resource allocator. Before going to into the details of the

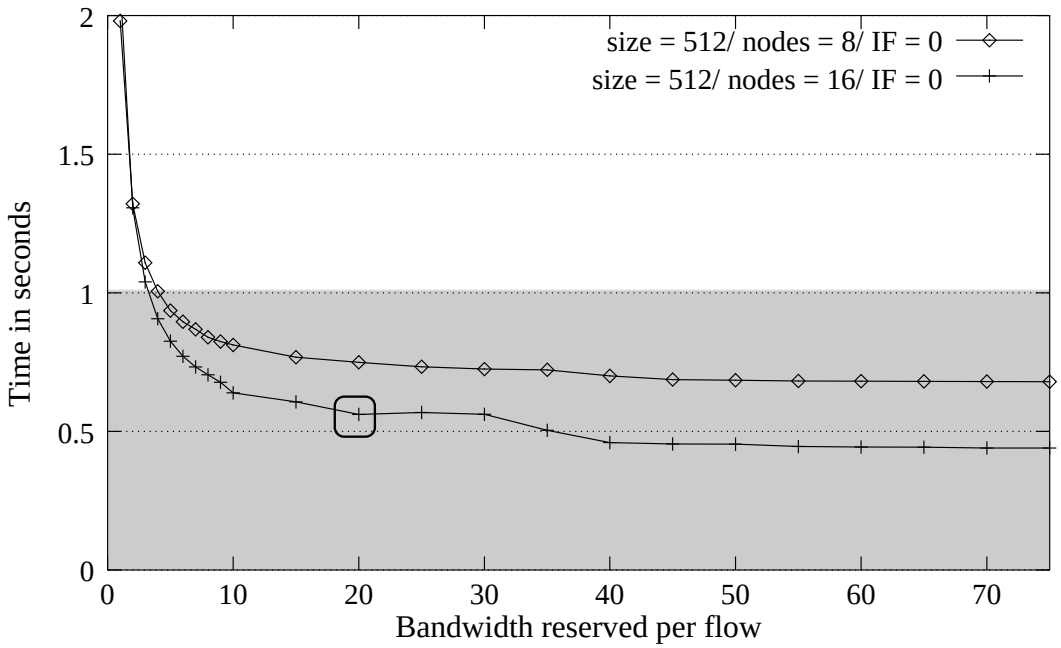


Figure 3.9: Final point returned by the resource allocator giving the application the least possible response time

algorithm let us first formalize the descriptions of the profiled data and application parameters that we had described.

3.5.1 Formal Definition of Parameters

The input to the QoS-aware middleware is a set of application parameters. The application usually specifies some kind of time bound, which can be denoted by { t_1, t_2 }, where t_1 refers to the lower time bound, and t_2 refers to the upper time bound. In addition to this, the application can also specify a set of parameters specific to a particular application. Let these parameters be denoted by { x_1, x_2 ... x_n }. For example, for the ray-tracing application, n is equal to 2, as the application can specify the image size and interleaving factor that it requires. For the polygon rendering

application, n is equal to 1, and the application can specify only the image size that it needs.

The profiled data consists of a set of points, arranged into graph-curves for clarity. Let the graph-curves be denoted by G_1, G_2 ... G_k. Each graph-curve contains a set of attribute values unique to that graph-curve, which can be used to identify it. These attributes are as follows:

G_i.attr = { x_1, x_2 ... x_n, N, K}

{ x_1, x_2 ... x_n } are the application parameters described,

N = number of nodes, and K = Knee-point bandwidth of this graph-curve.

Each graph-curve consists of a set of points:

G_i.points = { P_1, P_2 ... P_l }

P_i = { BW, t } tuple, BW = Bandwidth, and t = execution time.

Input to the QoS-aware middleware: X ϵ Power Set{ t_1, t_2, x_1, x_2 ... x_n }

Output from the middleware: SUCCESS or FAILURE depending on whether the request can be satisfied.

Input to the QoS translator: X ϵ Power Set{ t_1, t_2, x_1, x_2 ... x_n }

Output from the QoS translator: Y_i = { BW, N, t }, BW = bandwidth, N = number of nodes, and t = execution time.

The point Y_i has final system resource allocation information to be sent to the Resource Allocator.

Input to the Resource Allocator: Y_i = { BW, N, t }, BW = bandwidth, N = number of nodes, and t = execution time.

3.5.2 Algorithms for the Middleware Layer

Schedule describes the overall working of the middleware layer.

Schedule(X)

```
1.      Y = QoS_Translate(X)
2.      if Not_equal(Y.BW, -1)
3.              Resource_Allocate(Y)
4.              Return SUCCESS
5.      Return FAILURE
```

QoS_Translate describes the overall working of the translator.

QoS_Translate(X)

```
1.      for i = 1 to k
2.              if match(G_i.attr, X)
3.                      Start_Thread(Search_Thread, X, G_i, i)
4.              else
5.                      Y_i.BW = -1
6.              end if
7.      end for
8.      Thread_Wait()
9.      y = Get_Smallest_Y()
10.     Return y
```

Search_Thread(X, G_i, index_of_thread)

1. for j = 1 to l
2. if match_time($G_i.P_j$.t, X.t_1, X.t_2)
3. if allocate_possible($G_i.P_j$.BW, G_i.N)
4. $Y_{index\_of\_thread}$.BW = $G_i.P_j$.BW
5. $Y_{index\_of\_thread}$.N = G_i.N
6. $Y_{index\_of\_thread}$.t = $G_i.P_j$.t
7. exit
8. end if
9. end if
10. end for
11. $Y_{index\_of\_thread}$.BW = -1

3.5.3 Explanation of the Pseudo-Code

Schedule describes the working of the middleware layer. The following is a step-by-step explanation of the pseudo-code.

1. Give the application parameters to *QoS_Translate* and obtain the output

2. If the Bandwidth component is not equal to -1, it means that the application request can be satisfied by available system parameters

3. Then send the Y value to the Resource Allocator for allocation

4. Return SUCCESS to the requesting application

5. Else Return FAILURE

QoS_Translate is the overall function that searches through the profiled data and finds matches to application given parameters which can also be satisfied by available system resources. The following is a step-by-step explanation of the pseudo-code.

1. For each graph-line in the profiled data

2. If there is a match between the properties of the graph-line and the given input parameters

3. Assign a parallel search thread to search through the points of the graph-line

4. Else if no match exists

5. Indicate that the corresponding Y_i value is invalid by setting its Bandwidth component to -1

6. End of If statement

7. End of For statement

8. Wait until all the parallel search threads started have terminated

9. Search through the Y_i values to get the values with the smallest response time

10. Return the Y_i value with the smallest response time

Search_Thread describes the working of the parallel search threads responsible for finding the set of valid points for each graph. Each thread confers with the resource allocator to determine the point that has the maximum bandwidth allocation on a graph-curve, and consequently the least response time, and at the same time can be satisfied by available system resources. If there is a point on the graph whose system requirements can be satisfied, the thread returns the information pertinent to that point. Else, a null record is returned to show that there are no points on that

graph-curve that can be satisfied by existing system resources. The following is a step-by-step explanation of the pseudo-code.

1. For each point on the graph-line G_i from the knee-point to the point with smallest bandwidth assigned

2. If the execution time corresponding to the current point $G_i.P_j$ lies between the bounds given by the application

3. If the Resource Allocator can allocate the bandwidth corresponding to $G_i.P_j$ on the number of nodes corresponding to G_i.N with the current available resources

4. Set the Bandwidth component of the Y_i value corresponding to the passed index to the bandwidth corresponding to $G_i.P_j$

5. Set the Number of Nodes component of the Y_i value corresponding to the passed index to the number of nodes corresponding to G_i

6. Set the Execution Time component of the Y_i value corresponding to the passed index to the execution time corresponding to $G_i.P_j$

7. Exit the *Search_Thread* function as the best point for that graph-line has been determined

8. End of inner If statement

9. End of outer If statement

10. End of For statement

11. Indicate that the Y_i value corresponding to the passed index is invalid by setting its Bandwidth component to -1

Each entry in the Y list corresponds to the point of largest bandwidth on every graph-curve that satisfies both the application-given constraints and the system

resource limitations. There may be entries that are empty, either because the corresponding graph-curve does not match given application parameters or because none of the bandwidth values for that graph-curve can be satisfied by available system resources.

The Y_i returned by *QoS_Translate* is sent to the resource allocator so that this reservation can be entered into the system. The resource allocator subtracts Y_i.BW from the links leading to the nodes involved in the reservation. It then returns information necessary for reservation such as bandwidth value, number and names of nodes and ports to the request handler which in turn passes this information to the application so that the application can make the reservation call to the rate-control agents at the NIC. If no such Y_i exists, that is, the request cannot be satisfied, then the resource allocator simply returns the control back to the request handler informing it of its final decision and the request handler conveys this information back to the waiting application.

Chapter 4 describes the performance evaluation of the proposed middleware layer.

CHAPTER 4

PERFORMANCE EVALUATION

In order to present the advantages of our framework in a clear manner, the performance results are divided into two main sections. We first present the basic results that validate the effectiveness of the system-level support for the QoS-aware middleware layer, namely the rate-control mechanism, in guaranteeing predictable execution time of the application even in the presence of network contention. Then we present detailed, application-level evaluation of the QoS-aware middleware layer that shows the benefits gained by using such a framework.

4.1 Experimental Setup for the Rate-Control Mechanism

The following three kinds of experiments were carried out.

1. Applications were run with varying bandwidth reservations to show the impact of rate control on their execution time.

2. Applications were run in the presence of background bandwidth-hungry flows to show the effectiveness of the QoS framework in reserving network resources and delivering predicted execution time.

3. Multiple applications were executed simultaneously on the same set of nodes, on different sets of processors, but sharing the same underlying network. Since each application reserves a certain percentage of network bandwidth, the network gets essentially partitioned into disjoint virtual networks, where each virtual network is available for the exclusive use of the application that has reserved it. This test also highlights the predictable execution time for an application in the presence of background applications.

4.1.1 Experimental Testbed

The implementation was evaluated on a cluster of workstations with eight 700 Mhz Quad Pentium III processors, running Red Hat Linux kernel version 2.4.7-10 smp. These machines were connected by an 8-port Myrinet switch and LANai 7.2 NICs with 66 MHz processors. The communication layer running on the Myrinet cards was GM 1.5.1, and the MPI version was MPICH 1.2.1.7. To determine the maximum bandwidth that can be supported by the network, MPI bandwidth and latency tests were performed on the testbed. It was determined that the maximum bi-directional bandwidth that can be supported per link is 210 MBps.

4.1.2 Overview of Applications

The tests were performed using the NAS Benchmark Suite[18], and two visualization applications. The NAS Benchmarks used were: Integer Sort (IS), Block tridiagonal solver (BT), LU solver (LU), Conjugate Gradient (CG), Multigrid (MG), and Pentadiagonal solver (SP). All the NAS benchmarks with the exception of MG were executed with size class A and some of the benchmarks were executed with number of iterations and size differing from the class A default. The visualization applications

tested were an iso-surface extraction application and a ray-tracing application[14] that can be executed with different sizes of input data.

4.2 Experimental Results for the Rate-Control Mechanism

4.2.1 Impact of Rate Control on Application Execution Time

The applications were run with different bandwidth reservations. Uniform allocations were made to every communication flows in the application. The tests were performed for four and eight nodes. The results are shown in Figure 4.1 for the NAS benchmarks and the visualization applications for four nodes, and in Figure 4.2 for eight nodes. Since reservations were made for all possible communication links, each application on N nodes will have N-1 outgoing flows, and N-1 incoming flows, making the total number of flows on the link from a node to the switch as 2N-2. By this calculation, on four nodes, the maximum reservation that can be made per flow is $210/6 = 35$ MBps, and for eight nodes, it is $210/14 = 15$ MBps. The results show that as the allocated bandwidth for each flow is increased, the execution time decreases. Since these are not simple bandwidth or latency tests, but complex applications with various computation to communication ratios and different communication patterns, the rate at which the execution time decreases varies according to the application. These tests also show the resource-adaptive nature of the application, in that the application can be run with any assigned QoS value. It also shows the predictive property that the QoS mechanism provides i.e., a reservation of a certain amount of network resources for an application provides a guarantee that the application will complete within a certain amount of time. By detailed study of the graph curves, one can also determine the exact minimum bandwidth required for each application for

a certain data size. For example, 15 MBps is sufficient for the iso-surface extraction application on 4 nodes. This indicates that an application need not reserve all available network bandwidth and the remaining available bandwidth can be shared with other applications.

4.2.2 Guaranteeing QoS in the Presence of Background Flows

The same applications were tested on four nodes again, but now in the presence of 2 and 3 background flows that consists of a sender running continuously and pumping out data to a receiver. The experimental setup is shown in Figure 4.3. The first test consists of 2 background flows, whereas the second test consists of 3 background flows. Since these are parallel applications that interact frequently, it is sufficient that we run the background flows on one of the nodes on which the application is executing so as to put the maximum load on the NIC at that node. In the communication system without QoS features, these flows are treated equal to the communication flows of the application and will therefore cause more interference. Whereas in the network layer with QoS features, these background flows, not having made any reservations will be treated as best-effort traffic, and will take second priority to the premium traffic.

The results are shown in Figures 4.4(a) and 4.4(b) for four nodes. Figure 4.4(a) shows the results obtained for the NAS benchmarks in the presence of three background flows. Figure 4.4(b) shows the results obtained for the ray-tracing application in the presence of 2 and 3 background flows. From the graphs, it is clear that the QoS framework guarantees a predictable response time for the application even in the presence of heavy-duty flows running on the same nodes. Since the nodes in the

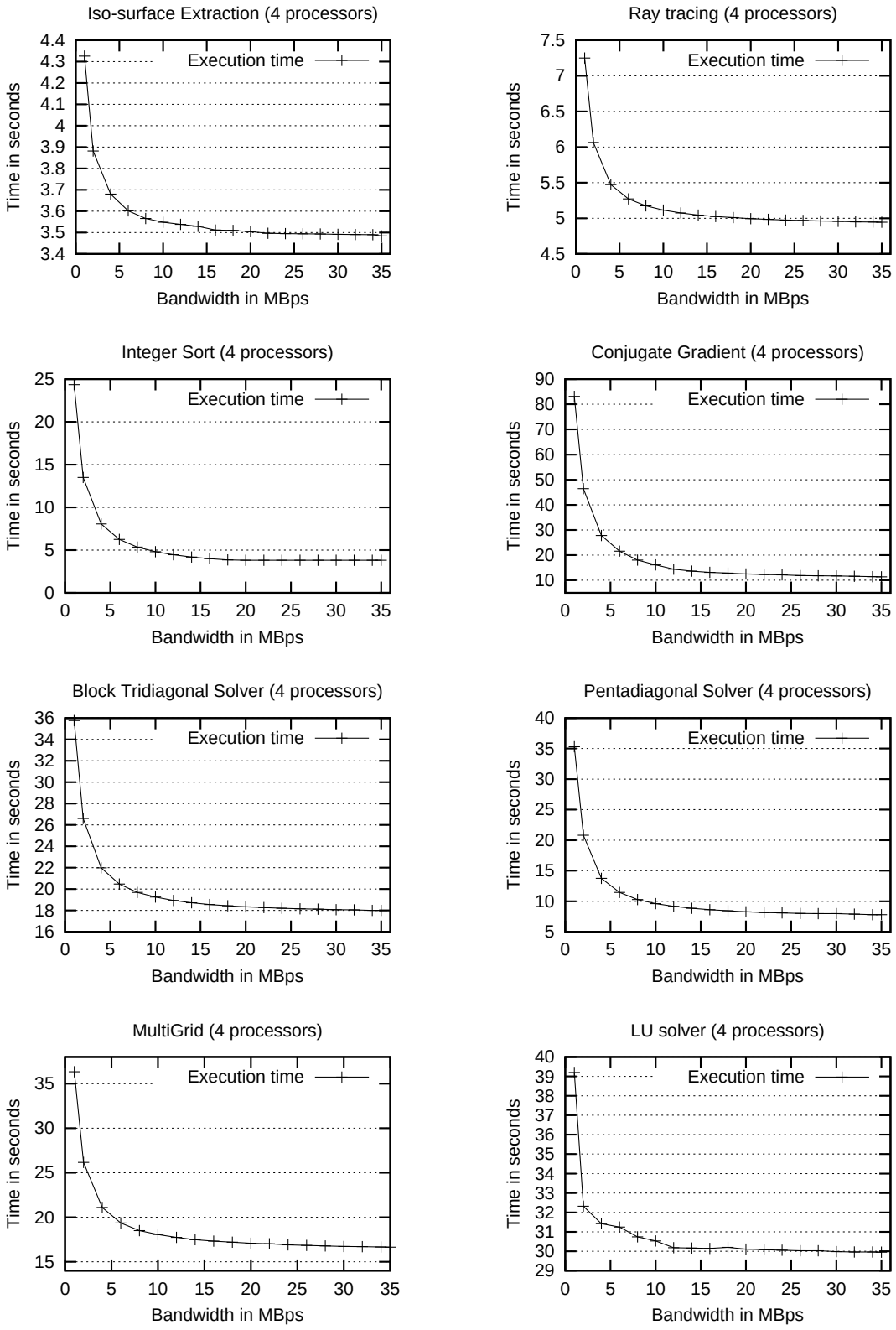


Figure 4.1: Impact of rate-control on execution time of applications on 4 nodes

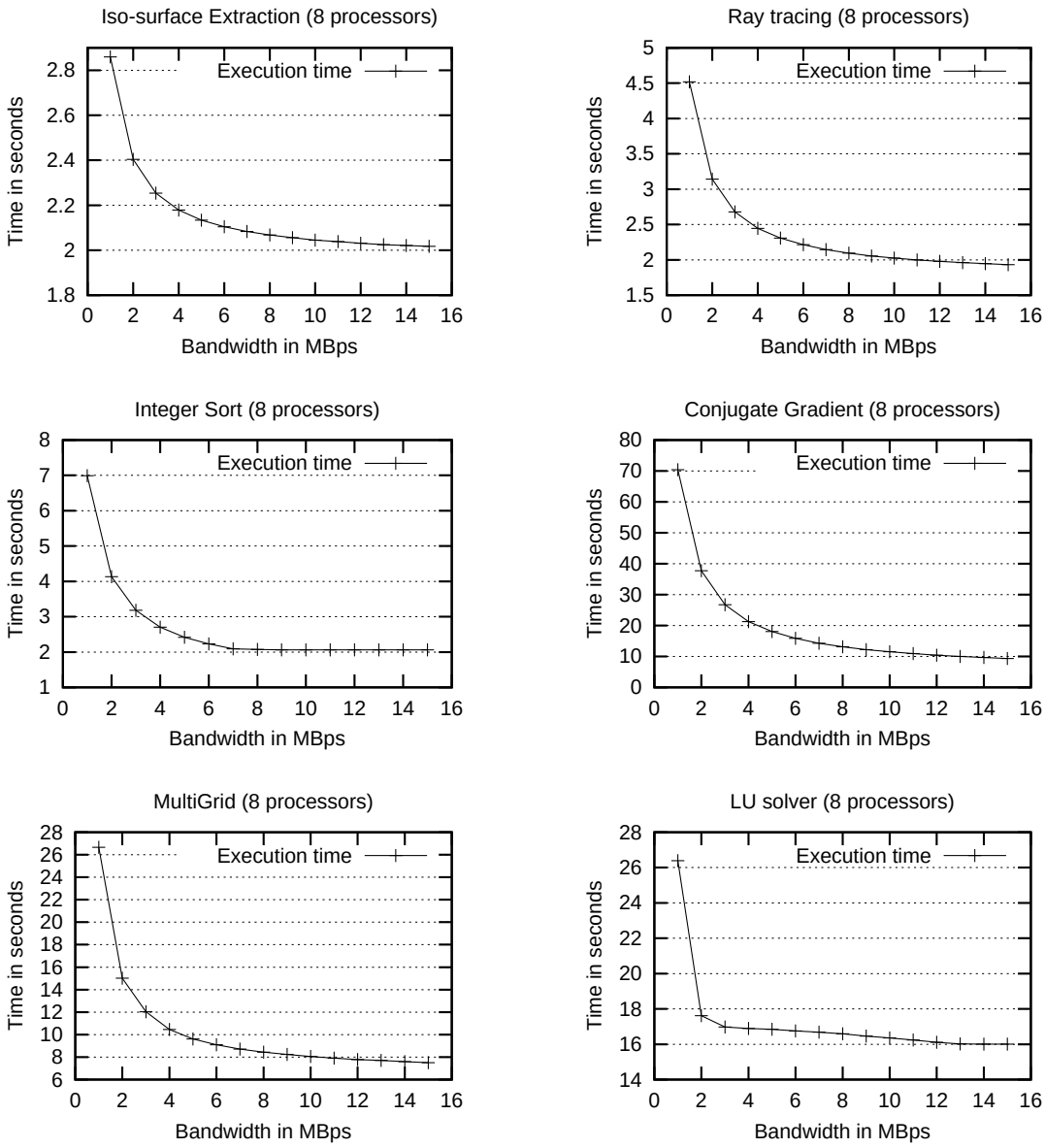


Figure 4.2: Impact of rate-control on execution time of applications on 8 nodes

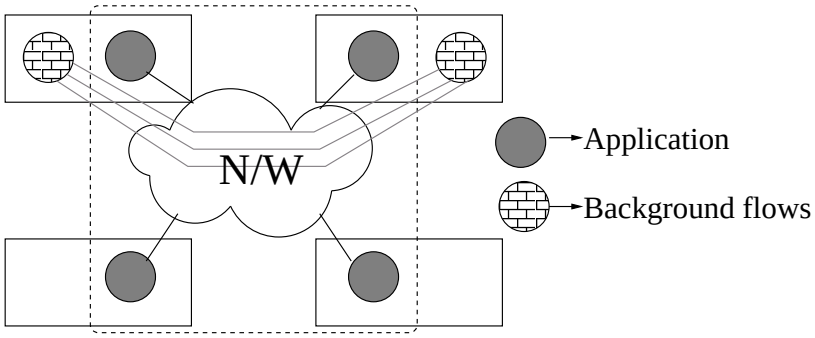
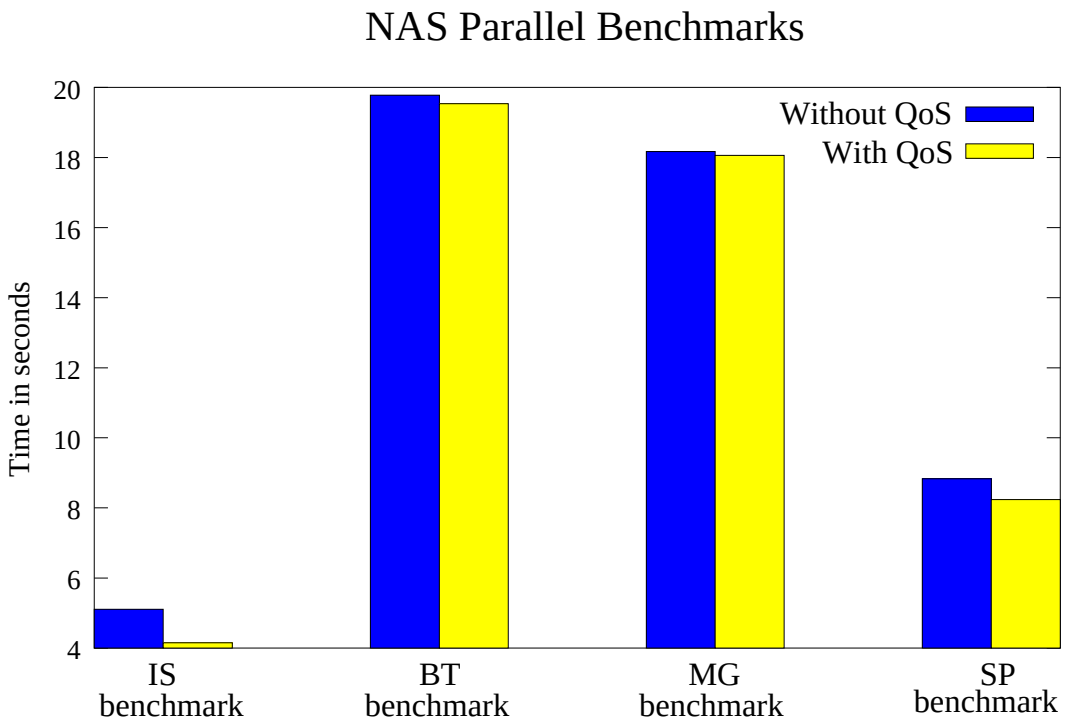


Figure 4.3: Performance evaluation framework in the presence of background flows

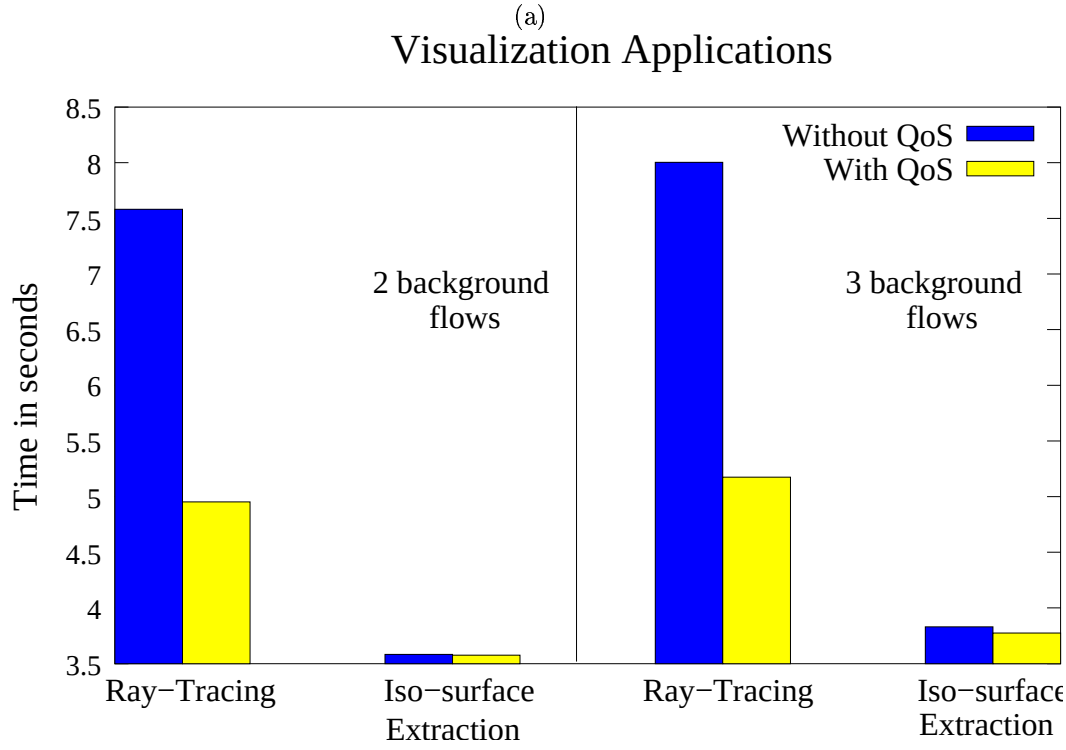
testbed are quad SMPs, we can reasonably suppose that the flows are assigned to different processors from those on which the application is running, and the interference due to the flows is solely at the network level.

4.2.3 Executing Multiple Jobs Simultaneously with Individual QoS Requirements

In this evaluation, we ran 2 applications simultaneously with different QoS values. Since each application has allocated bandwidth, it has exclusive use of the portion of network resources assigned to it, and behaves as if it is running on an exclusive virtual network with resources equal to the value of bandwidth allocated to that application. This mechanism thus paves the way for efficient and reliable use of clusters wherein network resources can be partitioned for client applications with no interference between users. The performance evaluation framework for this experiment is shown in Figure 4.5. The results of these tests are shown in Figures 4.6(a) and 4.6(b) for 2 parallel applications on four and eight nodes, and in Figures 4.7(a) and 4.7(b) for 3 parallel applications on four and eight nodes.



(a)



(b)

Figure 4.4: Performance comparison between QoS-based and non-QoS-based frameworks

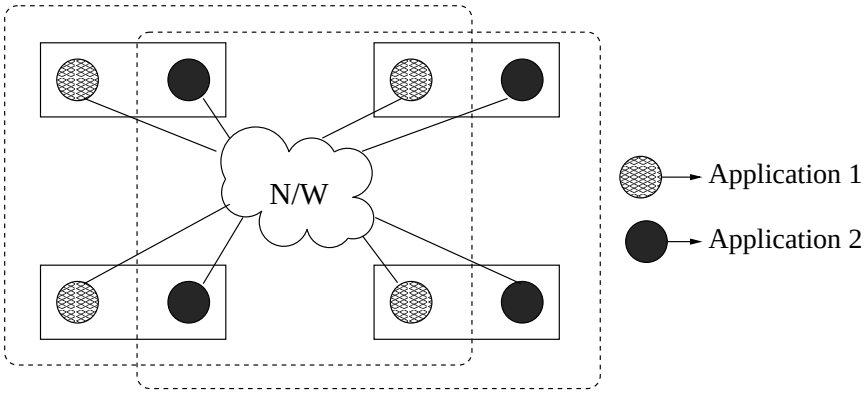
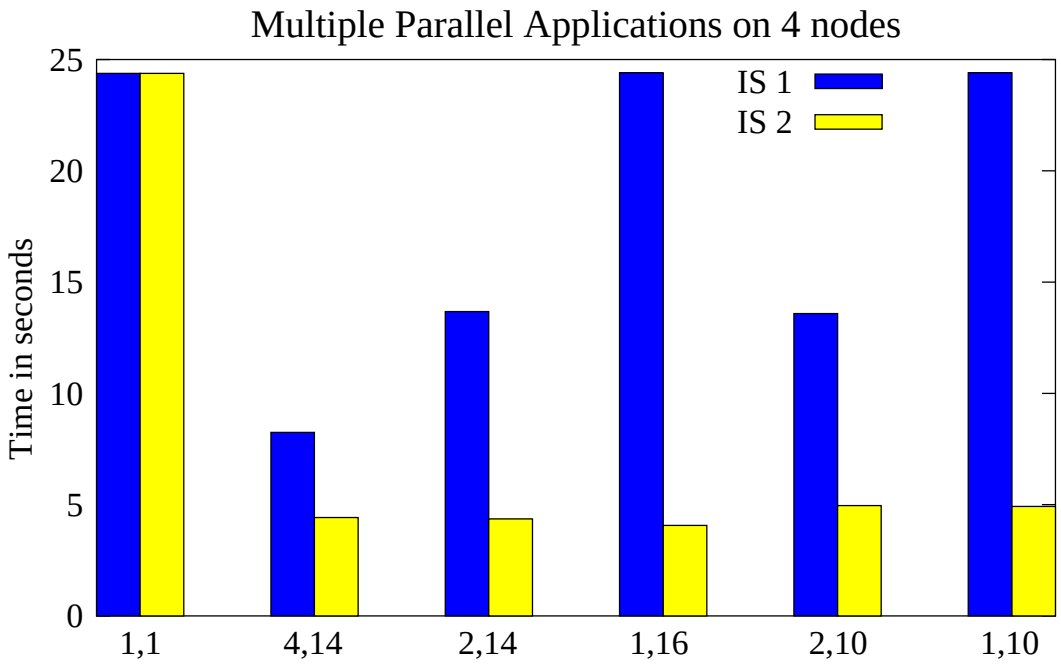


Figure 4.5: Performance evaluation framework for executing multiple applications simultaneously with individual QoS requirements

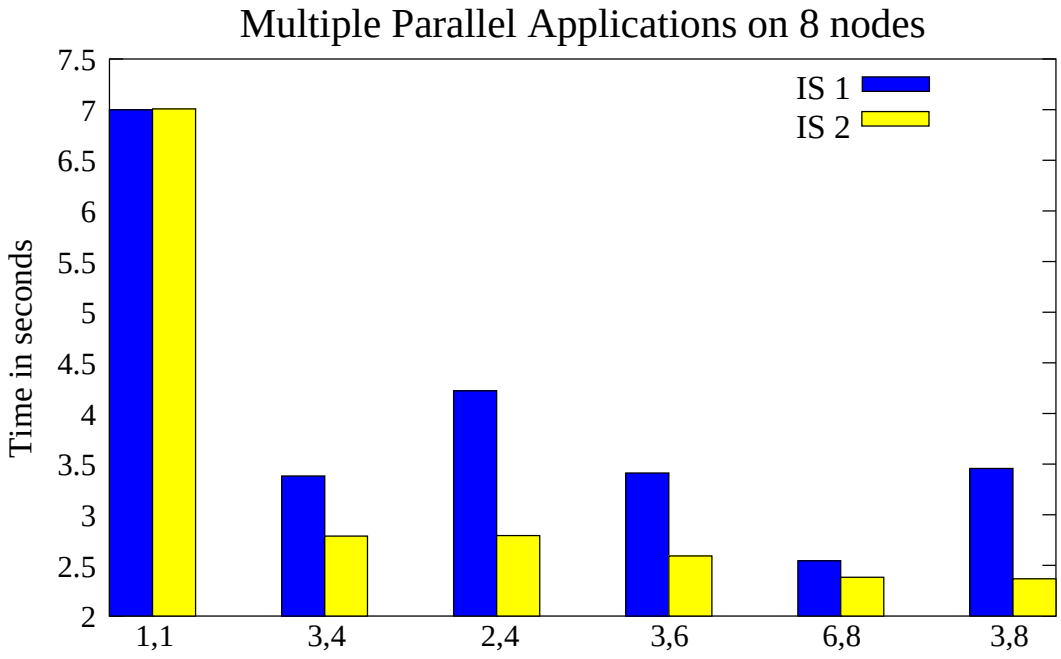
The x-axis plots the value of assigned bandwidth to the application pairs and triples in MBps. The y-axis plots the total execution time of the applications in seconds. It can be seen that even in the presence of a background application, the execution times of the application do not change, and are the same as in the experiment where only a single application is run. For example, from Figure 4.6(a), the execution times for the bandwidth pair (5,8) is (0.1404, 0.09175). From the graph for Integer Sort (IS) in Figure 4.1, we can see that the execution time is 0.1404 when the bandwidth reserved is 5 MBps, and it is 0.0925 when the bandwidth reserved is 8 MBps. Thus the presence of another application does not affect the performance of an application as long as it has reserved a fraction of the network bandwidth.

4.3 Experimental Setup for the QoS-aware Middleware Layer

In order to provide user requests that would be most similar to a real-world example, requests were sent to the middleware at random intervals. Also, in the real world, when a user is rendering an image, it is possible that he would try to look at

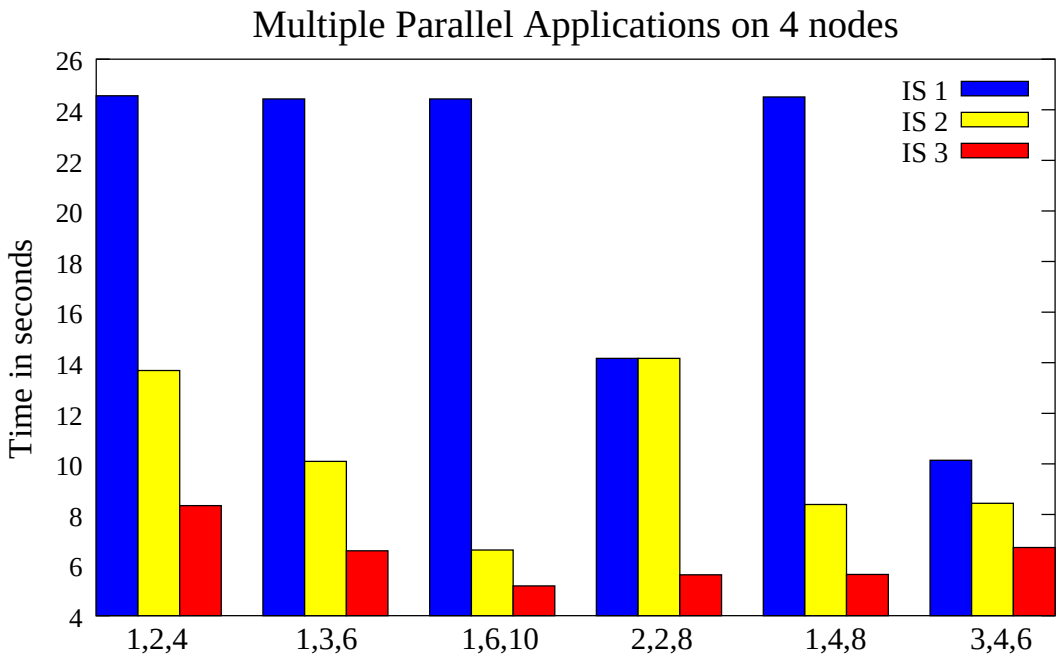


(a)

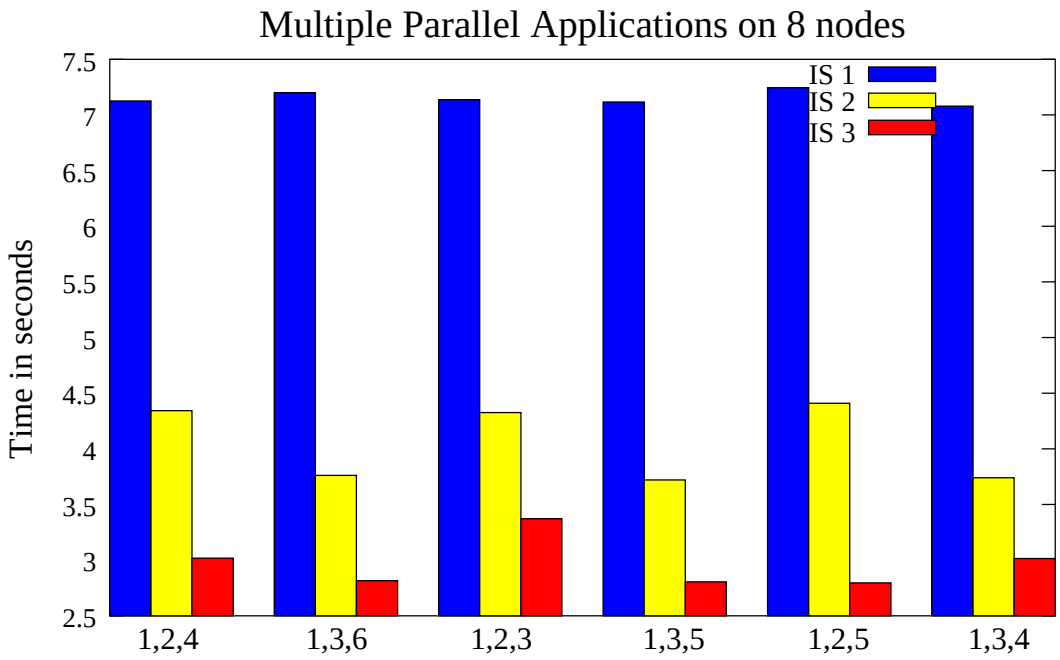


(b)

Figure 4.6: Performance evaluation of 2 parallel applications executing on 4 and 8 nodes with individual QoS requirements



(a)



(b)

Figure 4.7: Performance evaluation of 3 parallel applications executing on 4 and 8 nodes with individual QoS requirements

the image from different directions, magnify a part of the image, or rotate the image. Each of the above actions will results in a further rendering action, as the image to be viewed will be completely different from the image being viewed. Therefore, in order to simulate this scenario, we modeled every user request as a collection of rendering requests, each of which can be for a different image size and quality.

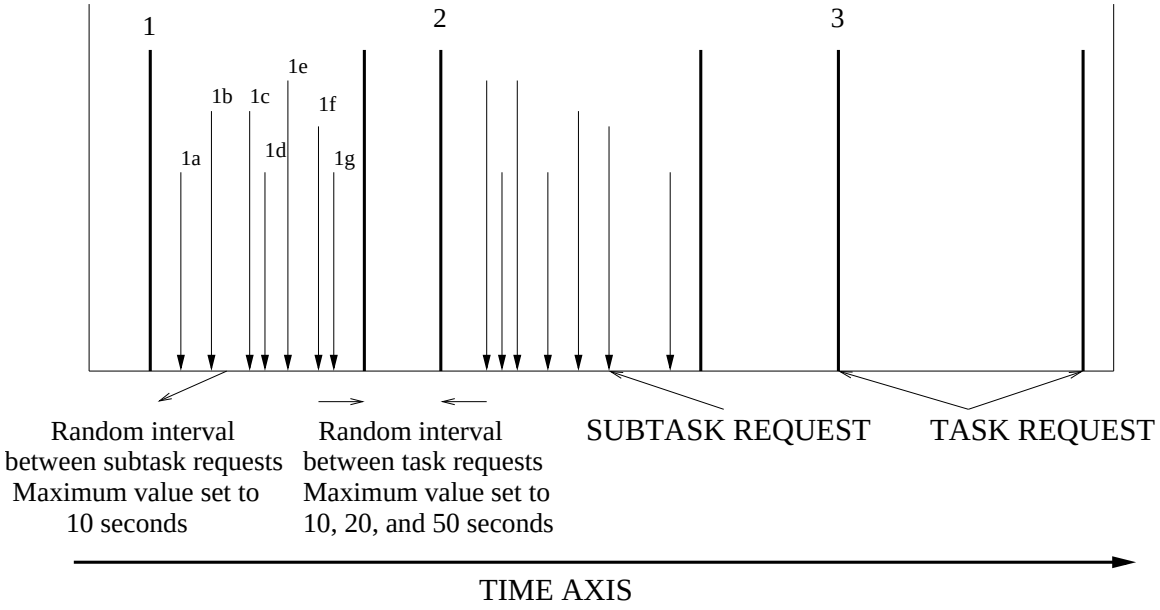


Figure 4.8: Experimental format showing the definition of task and subtask requests

Therefore, the experimental setup consists of a set of tasks arriving at random intervals, where each task is actually a further collection of subtasks with different application level parameters and different random time intervals between them. A subtask is a request for a single rendering application with specific application parameters. The experimental format is shown in Figure 4.8. For such an experimental setup, we have changed the following parameters:

1. The maximum time interval between successive high-level requests. This time interval was assigned values of 10, 20 and 50 seconds. The actual time interval between requests was varied randomly between 0 and the maximum number.

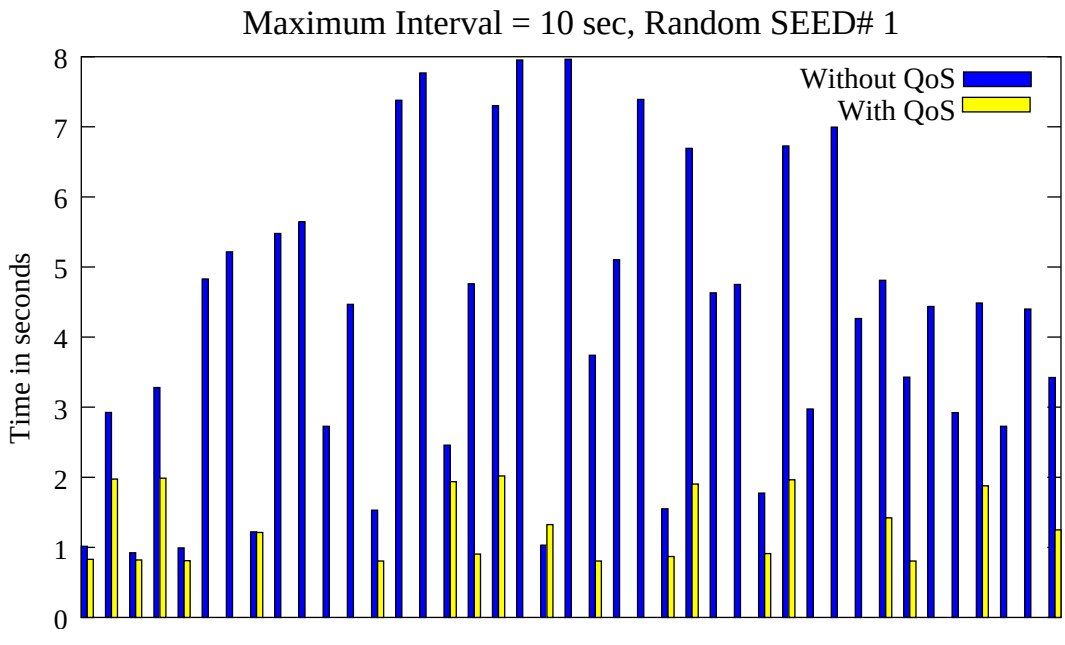
2. The actual random intervals between input requests that were part of the same high-level request was varied by using different random seed values.

3. The ratio of arrival image sizes and interleaving factors. For the ray-tracing application, first the input requests had interleaving factors of 0 and 2 in a 50:50 ratio, while the image sizes of 512 and 1024 were varied in ratios of 50:50, 30:70, and 70:30. Then the image sizes of 512 and 1024 were kept at a steady 50:50 ratio, while the ratios of the interleaving factors were varied as before.

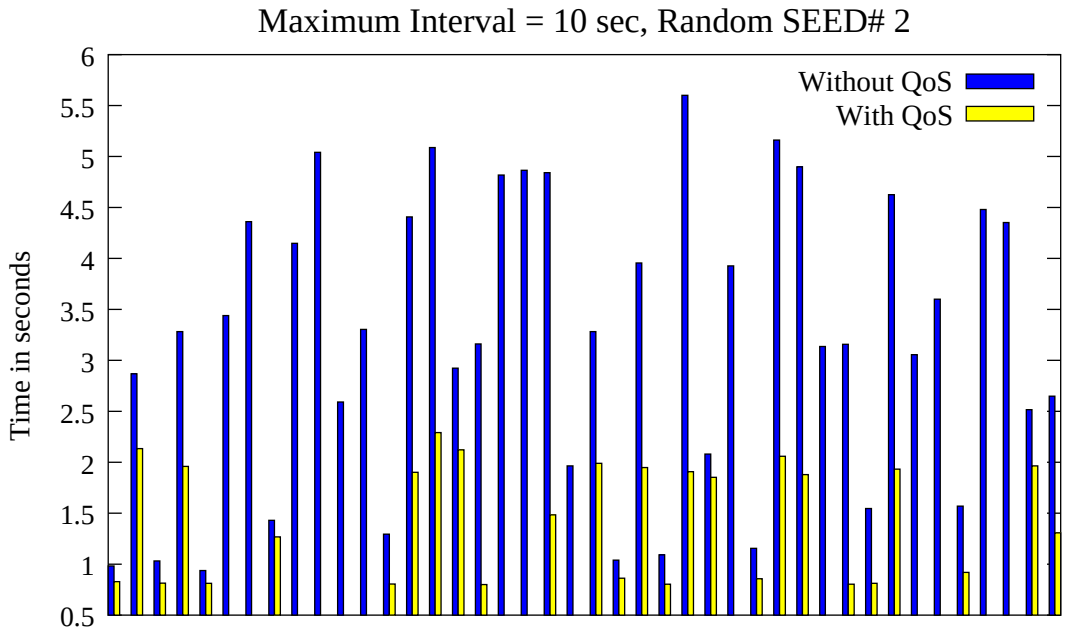
4. For the polygon rendering application, only the image sizes were varied in different ratios.

4.3.1 Experimental Testbed

The experimental testbed consisted of a cluster of workstations with sixteen 1GHz Dual Pentium III processors, running Red Hat Linux kernel version 2.4.7-10 smp. These machines were connected by 8-port Myrinet switches and LANai 9.2 NICs with 132 MHz processors. The communication layer running on the Myrinet cards was GM 1.5.1, and the MPI version was MPICH 1.2.1.7.

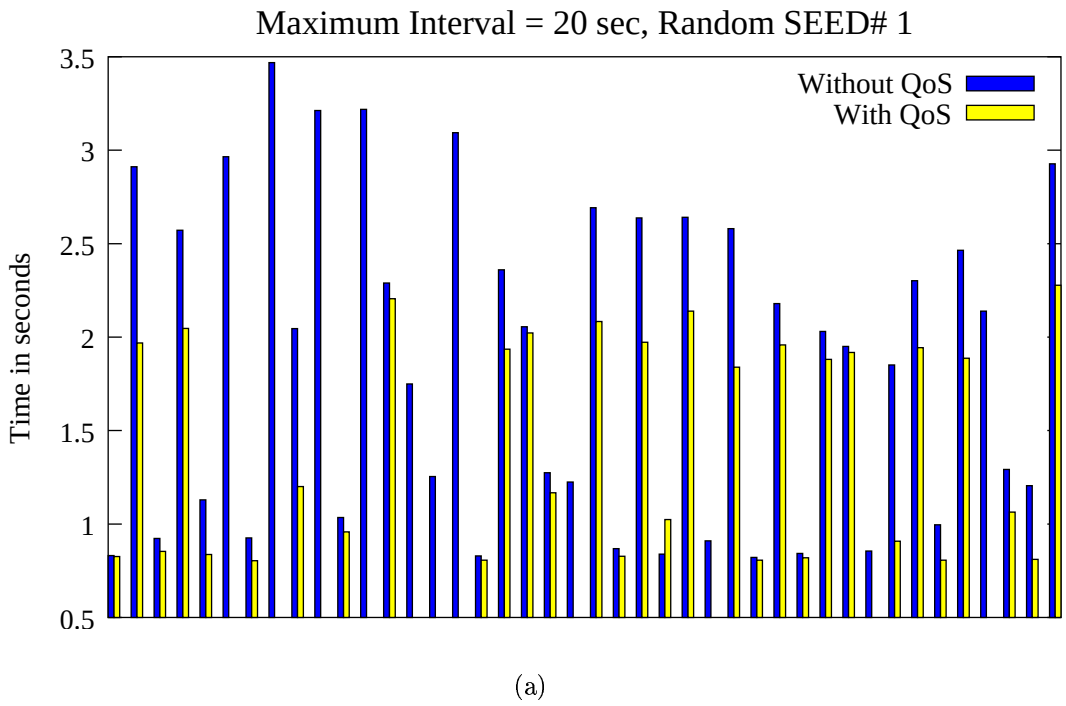


(a)

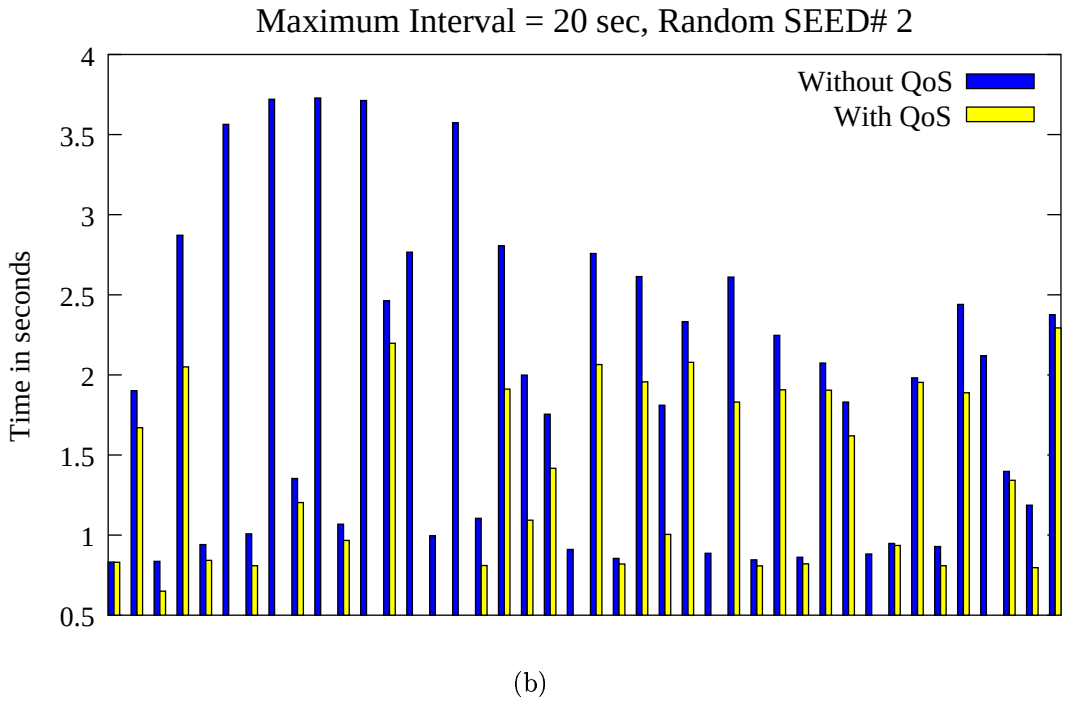


(b)

Figure 4.9: Experimental results for polygon rendering applications with maximum inter-arrival time of 10 seconds



(a)



(b)

Figure 4.10: Experimental results for polygon rendering applications with maximum inter-arrival time of 20 seconds

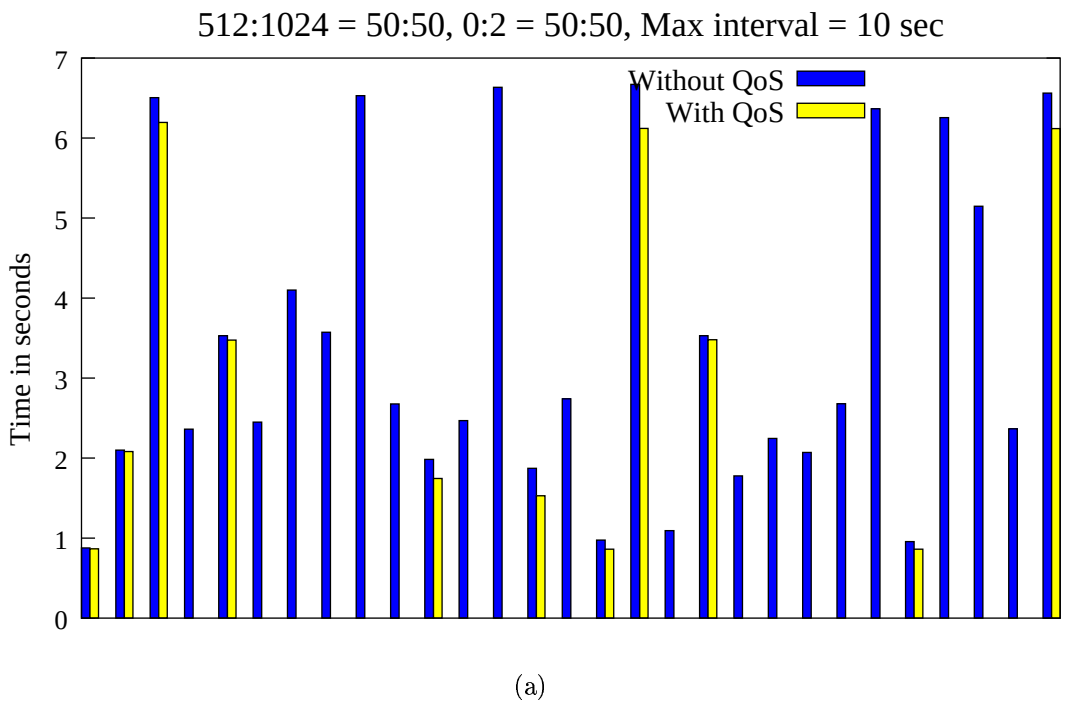
4.4 Experimental Results for the QoS-aware Middleware Layer

4.4.1 Experimental Results for the Polygon Rendering Applications

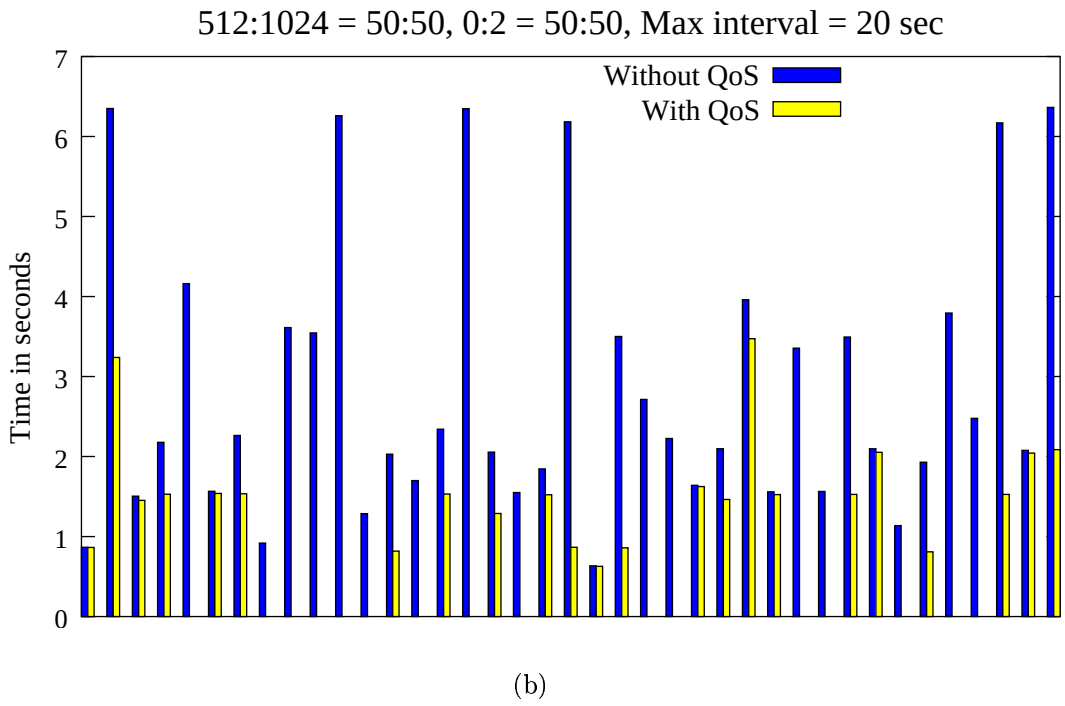
Figures 4.9 and 4.10 show the results obtained for the polygon rendering applications when the maximum inter-arrival time between requests is 10 and 20 seconds. User requests were for image sizes of 256, 512 or 1024, with desired response times. The distribution of image sizes was random across requests, with the number of requests for each image size approximately equal. The translator uses the profiled data for rendering applications to match a user request with corresponding system resource requirements, and the required resources are checked by the resource allocator to see if they are available. The x-axis shows the number of incoming requests. The y-axis shows the execution time for the application in seconds. The light-colored bars on the graph shows the execution times with the QoS middleware running on the system, and the shaded bars shows the execution times when no such middleware is present. It can be seen that there are requests which execute when there is no middleware, but are not allowed to execute in the presence of the QoS framework. These requests cannot be satisfied due to low availability of system resources, and correspondingly, it can be seen that if these requests are allowed to execute, the execution times obtained are much higher than expected, and therefore the user requirements are not satisfied in such situations.

4.4.2 Experimental Results for the Ray-Tracing Applications

Figures 4.11 to 4.14 show the experimental results obtained for ray-tracing applications. The graphs are obtained not only by varying the maximum arrival interval

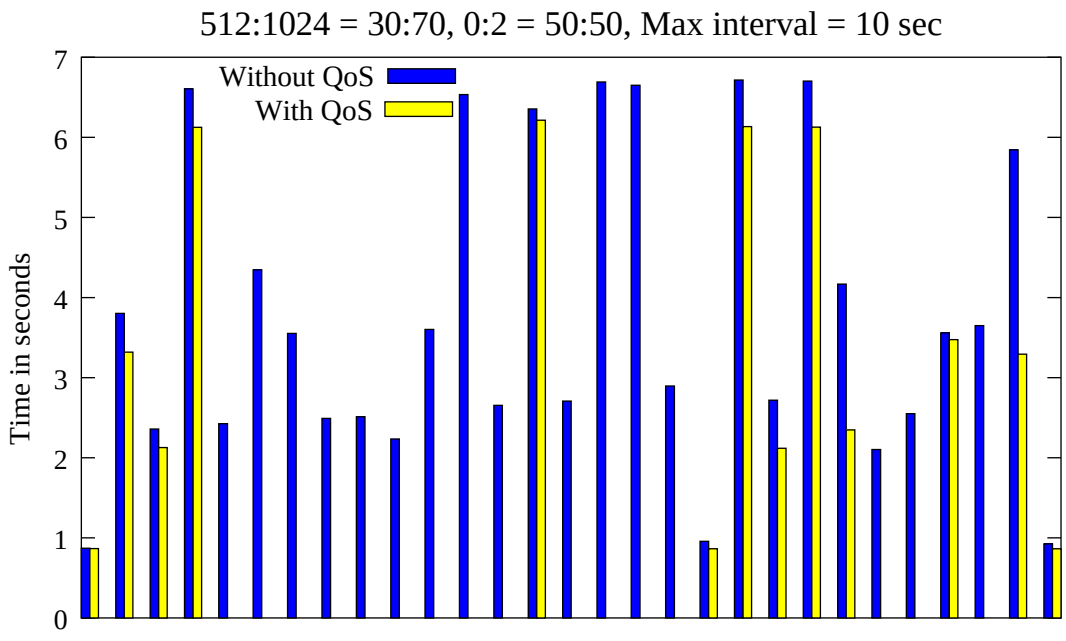


(a)

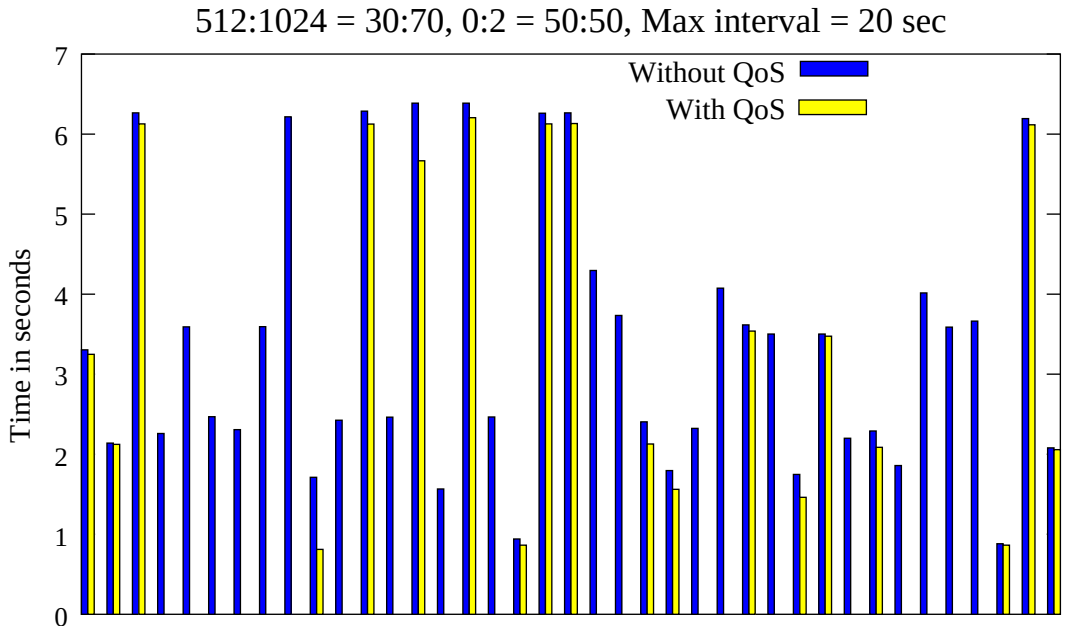


(b)

Figure 4.11: Results for ray-tracing applications with image sizes 512:1024 = 50:50 and interleaving factors 0:2 = 50:50

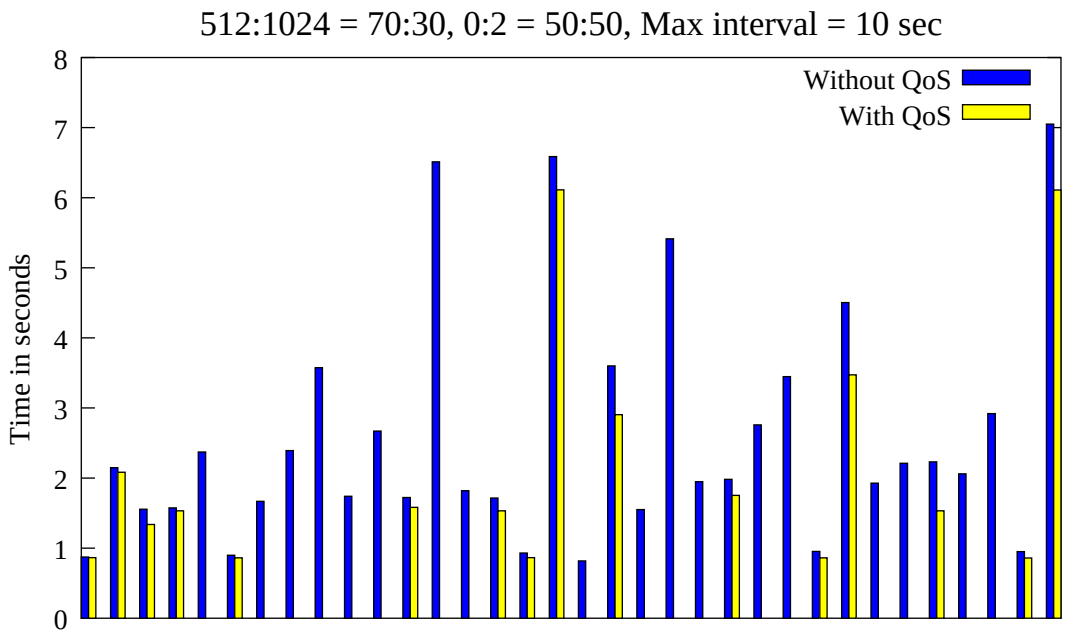


(a)

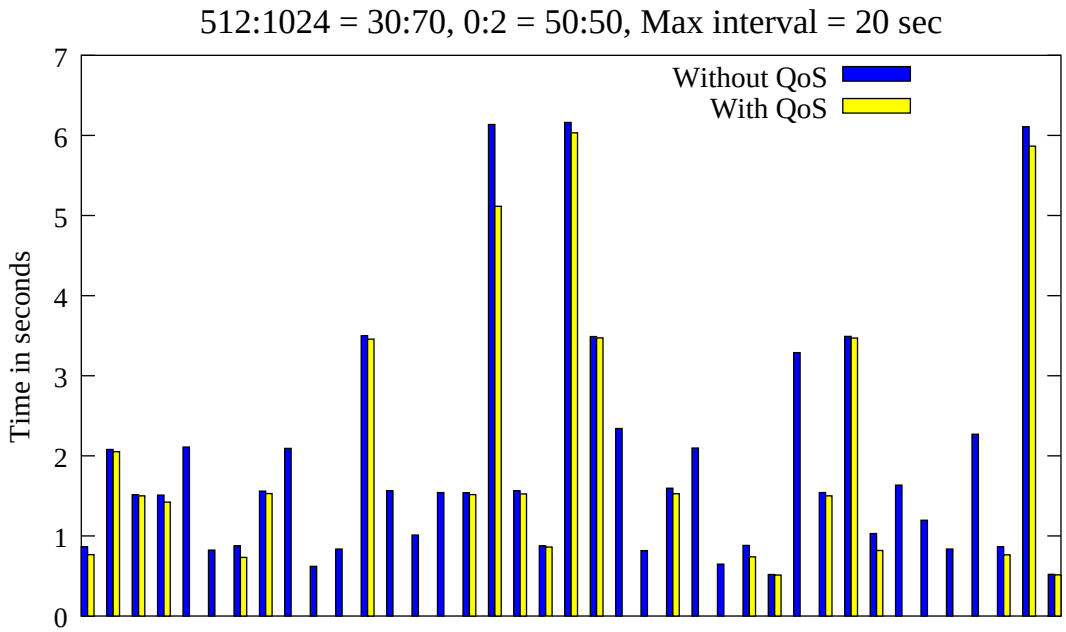


(b)

Figure 4.12: Results for ray-tracing applications with image sizes 512:1024 = 30:70 and interleaving factors 0:2 = 50:50

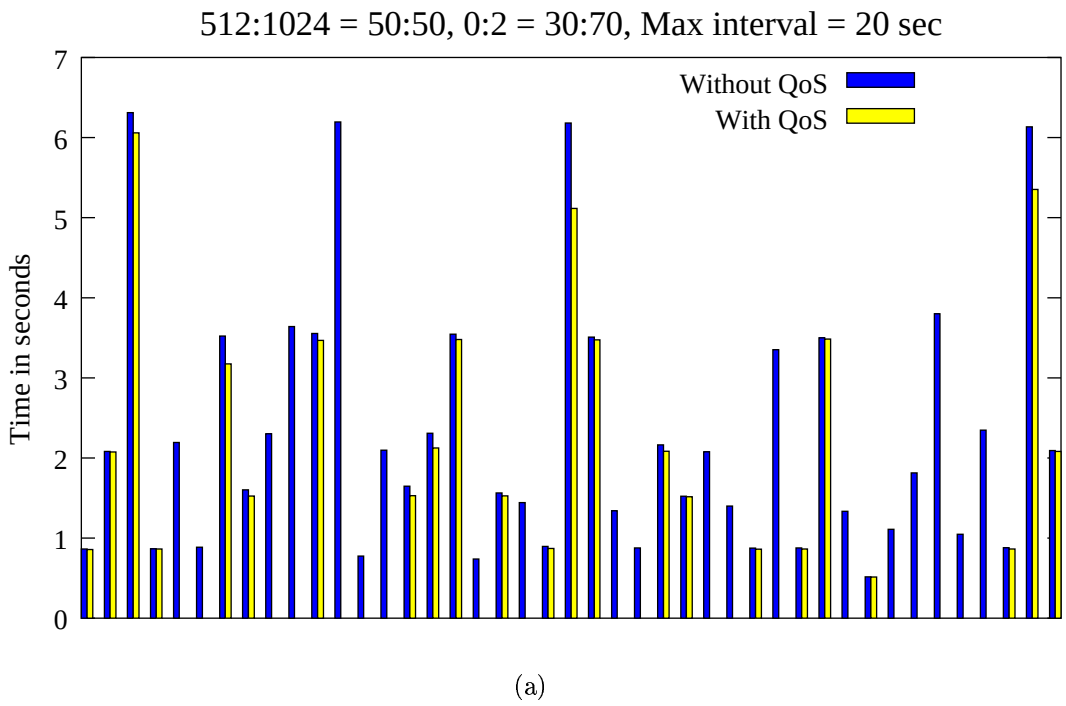


(a)

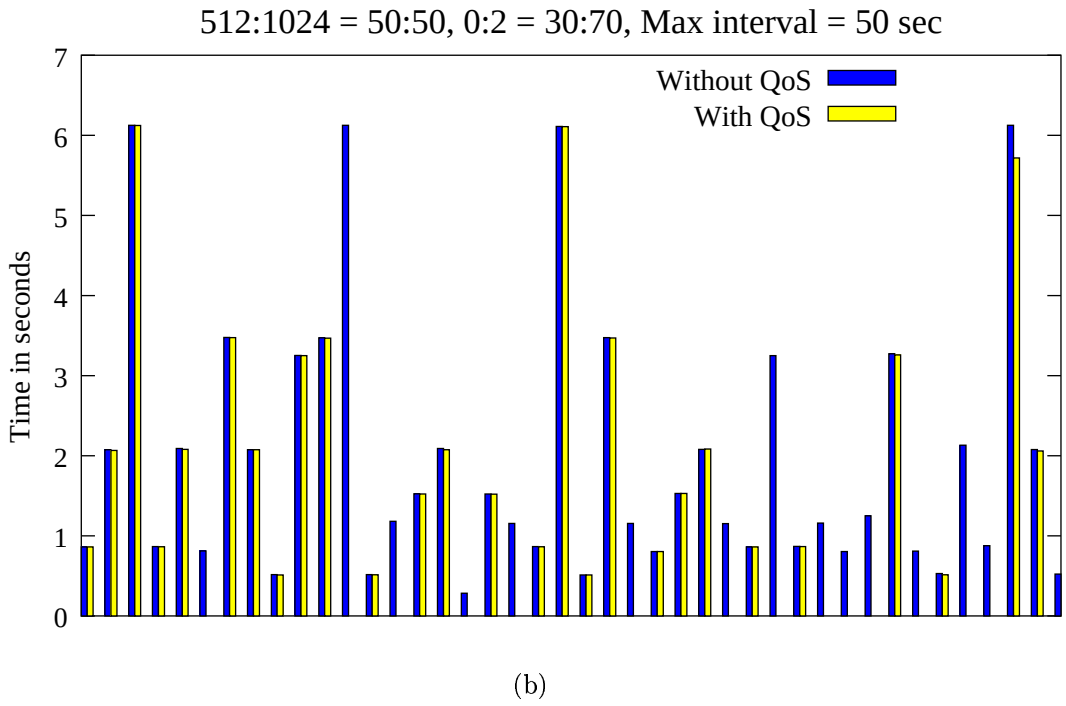


(b)

Figure 4.13: Results for ray-tracing applications with image sizes 512:1024 = 70:30 and interleaving factors 0:2 = 50:50



(a)



(b)

Figure 4.14: Results for ray-tracing applications with image sizes 512:1024 = 50:50 and interleaving factors 0:2 = 30:70

time, but by also changing the ratio of user requests for image sizes of 512 and 1024, and for interleaving factors of 0 and 2. Figure 4.11(a) shows the experimental results for ray-tracing applications when the ratio of image sizes of 512 to 1024 is 50:50, the ratio of interleaving factors of 0 to 2 is 50:50, and the maximum inter-arrival time is 10 seconds. Similarly, the ratios have been assigned as shown for the other graphs. The maximum interval arrival time has also been changed from 10 to 20 to 50 seconds, and results taken. The actual parameters are shown with each graph. As with the rendering application, it can be seen with the QoS-aware middleware layer, some client requests cannot be admitted due to lack of resource availability, but as a result, performance of existing applications improve due to the lack of contention.

4.4.3 Performance Metrics

To compare performance of applications when using the QoS-aware middleware layer with their performance when there is no such layer present, we define and use the following metrics.

1. We compare the execution times of applications obtained in the presence of the middleware layer with the execution times obtained when there is no such layer present. The comparison is done only for those jobs that have been *admitted* by the middleware layer. The percentage differences are computed using the following equation.

   Percentage difference = (T_{noqos} - T_{qos})/T_{qos}

   where T_{qos} = Execution time in the presence of the middleware layer, and T_{noqos} = Execution time in the absence of the middleware layer.

We show that significant performance increases can be obtained using the middleware layer.

2. We also compare the execution times obtained with and without the middleware layer with the *expected* execution times of the application. Again, the comparison is done only for the *admitted* jobs. The percentage differences are computed using the following equation.

   Percentage difference = (T_{actual} - $T_{requested}$)/$T_{requested}$

   where T_{actual} = Actual Execution time, and $T_{requested}$ = Requested response time of the application.

   We are able to show that applications can obtain very close to their expected execution times only with the support of the middleware layer.

3. Finally we also show the admission rates for the middleware layer, that is, the percentage of incoming requests that are admitted. The admission rate is calculated as follows.

   Admission rate = $N_{admitted}/N_{total}$

   where $N_{admitted}$ is the number of jobs admitted by the middleware layer, and N_{total} is the total number of requests submitted to the framework.

   We are able to show that close to 100% admission rates are possible at light loads, and reasonable admission rates at very heavy loads.

The performance improvements gained for the ray-tracing applications are shown in Table 4.1. The performance improvements for the polygon rendering applications are shown in Table 4.2. From the above tables, it can be seen that upto 80% improvement can be obtained for the polygon rendering applications and upto 64%

Max Interval in sec	Ratio of 512:1024 image sizes	Ratio of 0:2 int. factor	Maximum % Improvement	Average % Improvement
10	50:50	50:50	34.45	9.29
20	50:50	50:50	45.27	7.48
50	50:50	50:50	43.23	3.05
10	30:70	50:50	43.68	13.68
20	30:70	50:50	52.57	7.57
50	30:70	50:50	4.43	0.39
10	70:30	50:50	31.30	10.92
20	70:30	50:50	20.51	5.72
50	70:30	50:50	3.57	0.49
10	50:50	30:70	64.86	17.53
20	50:50	30:70	17.23	3.75
50	50:50	30:70	6.66	0.559
10	50:50	70:30	39.76	15.49
20	50:50	70:30	49.94	11.04
50	50:50	70:30	6.86	0.725

Table 4.1: Performance Improvements for the Ray-Tracing Applications

improvement can be obtained for the ray-tracing applications. The performance benefits gained are more for rendering applications because the communication to computation ratio is higher for these applications. It should be noted that whatever the communication to computation ratio, as long as communication plays a role in an application, which is the norm for all parallel applications, our framework will result in definite and significant performance advantages.

The second metric that we defined is the percentage increase between the expected time of execution given by a user request, and the actual execution time attained by the application. Figure 4.15 compares these percentage values for the QoS and non-QoS frameworks for maximum inter-arrival times of 10, 20, and 50 seconds, for the

Max Interval in sec (Seed value)	Maximum % Improvement	Average % Improvement
10 (SEED #1)	80.99	43.06
20 (SEED #1)	35.11	11.80
10 (SEED #2)	74.67	37.25
20 (SEED #2)	44.52	13.48

Table 4.2: Performance Improvements for the Polygon Rendering Applications

ray-tracing applications, and Figure 4.16 does the same for the polygon rendering applications.

From the graphs, it can be seen that the difference in the percentage increase values is most marked for the polygon rendering applications, again due to the fact that there is a high communication to computation ratio in these applications. For the polygon rendering applications, the difference in percentage increase of the actual execution time over the expected response time between the QoS and the non-QoS frameworks is as high as 117% for a maximum inter-arrival time of 10 seconds. As the inter-arrival times get larger, the contention and load in the system decreases, and so does the differences between the expected and actual times. It should also be noted that for the ray-tracing applications, though the communication contributes a very small percentage of the total execution time, there is a marked advantage in using the middleware layer due to the contention from other clients. This shows that whatever the communication to computation ratio of an application, our framework does provide a significant benefit.

Finally we determine the admission ratios of our framework under different load conditions for both the test applications. Figure 4.17 shows the resultant data for the

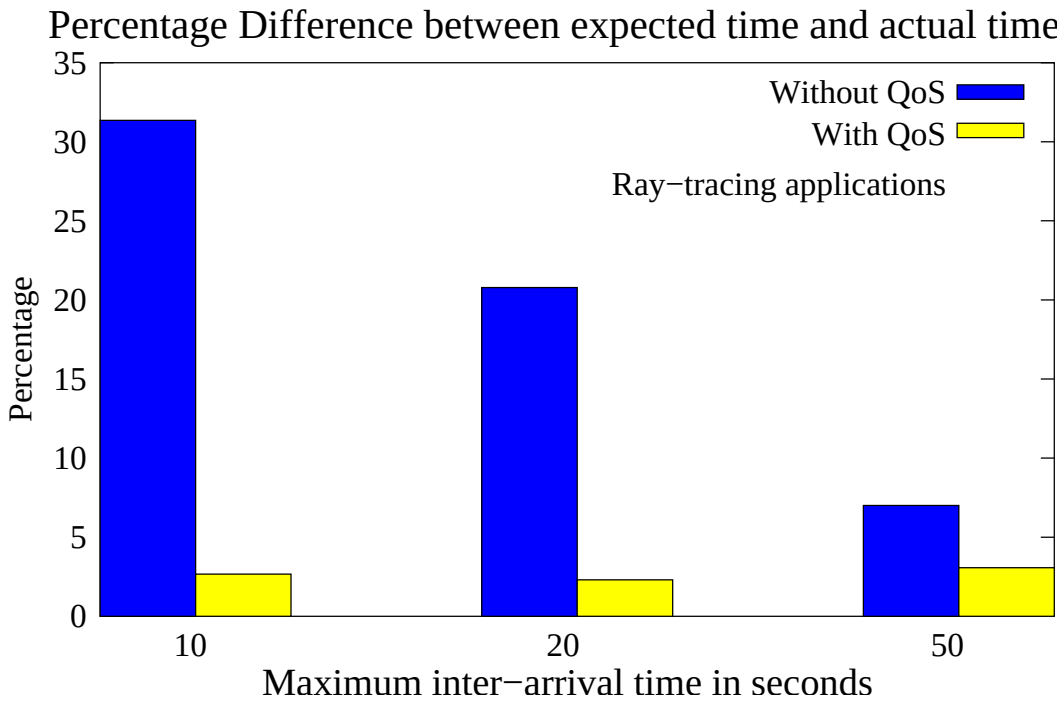


Figure 4.15: Percentage differences between expected and actual time for the ray-tracing applications

ray-tracing and rendering applications. The graph shows that as the inter-arrival time increases, the admission rate also increases due to the lighter load on the system. At very heavy loads, the admission rate falls below 40% for the ray-tracing application, and below 60% for the polygon rendering applications. But this lowered admission rate is validated by the lesser percentage increase values observed in the previous graphs, showing that, by keeping the admission rate small, the system is able to guarantee the applications execution time that is close to the requested response time. As the load on the system becomes lighter, the system is able to guarantee the applications the requested response time, while admitting more jobs, due to less

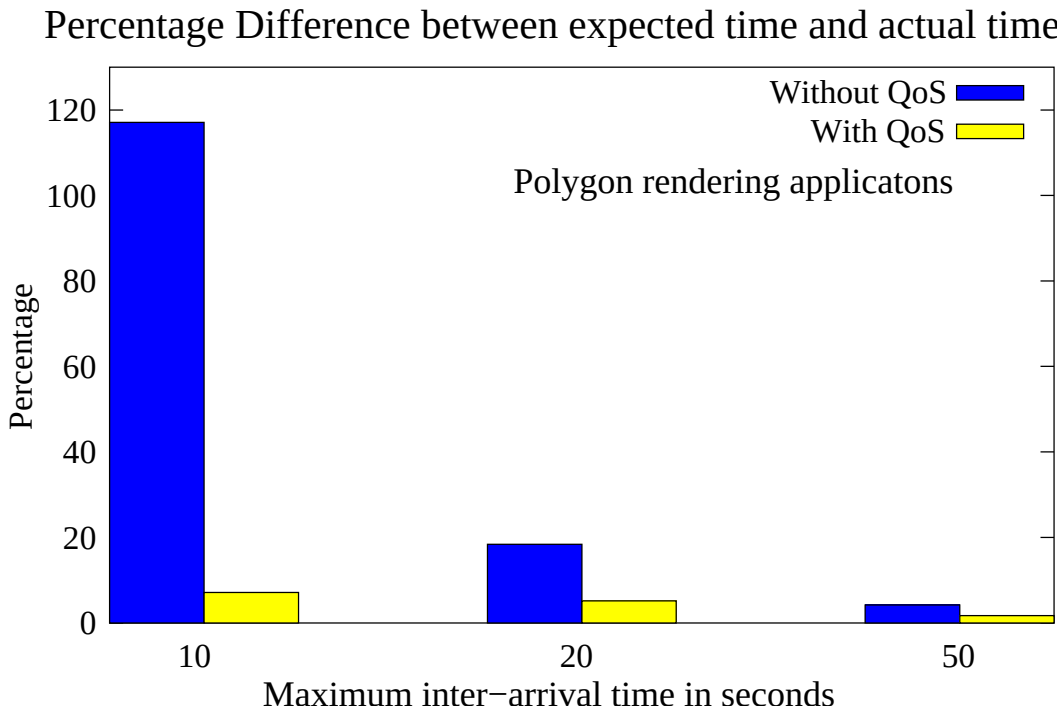


Figure 4.16: Percentage differences between expected and actual time for the polygon rendering applications

contention in the system. At lightest loads, the admission rate reaches almost 100% for the polygon rendering applications.

The results shown here demonstrate that the use of our proposed scheduler mechanism guarantees to deliver close to the requested response time of client applications even in the presence of contention from other applications. In the absence of a scheduler mechanism, it can be seen that the resulting high load and contention in the system can lead to very high differences between the applications' requested execution time and the actual time the applications complete execution, especially for the polygon rendering client applications. Since the above applications have the property of interactivity, this increase in response time at high loads is very undesirable,

and the advantages posed by the QoS-aware middleware in such a scenario become significant.

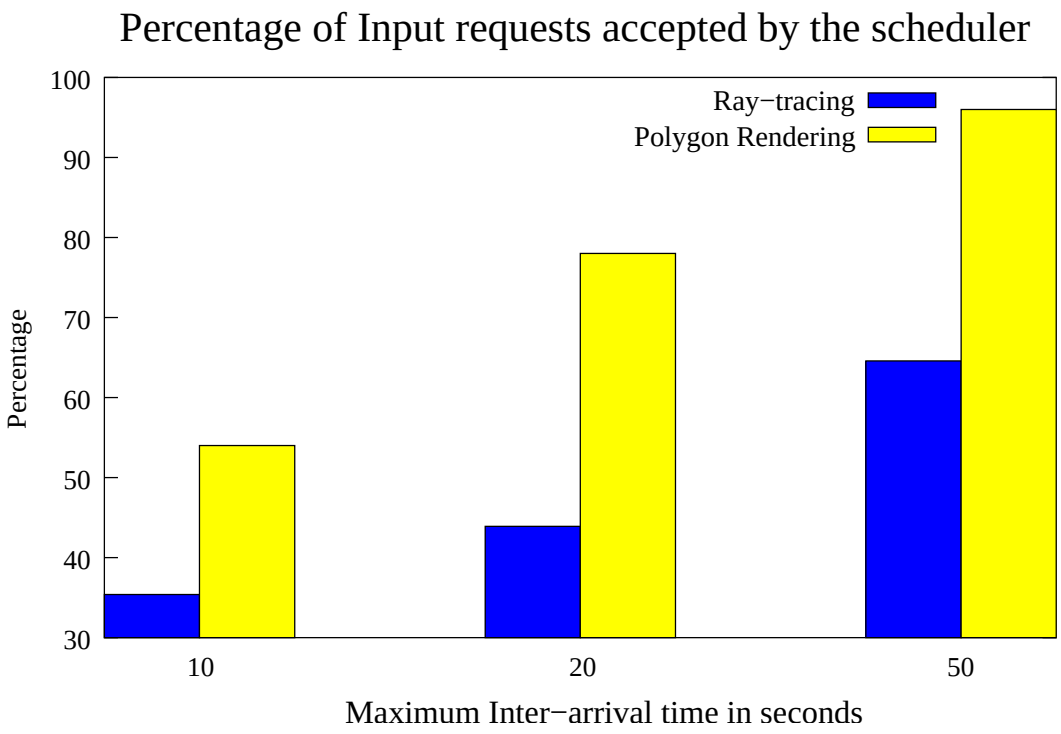


Figure 4.17: Admission rates at different arrival times for the test applications

Chapter 5 concludes this thesis, and also provides some insight into possible future research directions based on this work.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this thesis, we presented a QoS framework for supporting applications with resource adaptivity and predictable execution performance. The QoS-aware middleware layer exploits the resource-adaptive property of applications to determine the exact resource requirements needed to satisfy application demands. In order to provide the best match between application parameters given, and system resources needed to satisfy these demands, we realized that the use of profiled data taken apriori about the application was the best option. The developed middleware layer uses the profiled data of applications to choose the best set of resources to satisfy an application's demands. Resource allocation is done in an efficient manner such that only the minimum set of system resources (processors and network bandwidth) needed for the application to achieve its proposed performance goals are allocated. The framework is supported at the network level by a NIC-based rate control scheme that provides proportional bandwidth allocation to communication flows. The combined use of this rate control scheme and the middleware guarantees to first determine the exact resource requirements of applications and then to satisfy these QoS reservations for network and CPU resources. Therefore, the proposed framework promises to support next generation interactive applications (visualization, data mining, virtual reality,

etc.) on a shared cluster. By using the proposed middleware, we were able to demonstrate that applications can obtain execution times within 7% of expected response times while in the absence of such a framework, they may encounter an increase of 117% over the expected response times.

Currently the framework only uses a proportional bandwidth allocation mechanism for reservation of network resources. We are exploring the integration of CPU and disk scheduling mechanisms with the middleware layer. The scheduling algorithm used by the resource allocator component of the middleware currently uses a greedy scheme in that it allocates a maximum set of resources to satisfy the needs of the applications. The effect of using different types of scheduling algorithms can be explored. It will be interesting to consider application classes with different priorities and re-evaluate the scheduling algorithms in this context. Currently the framework is implemented by a centralized scheme, where all requests are sent to a single central manager which then handles the requests and sends the replies back. Another possible future research direction can be the design of a distributed framework and the advantages and disadvantages of such a scheme can be compared versus the centralized scheme currently in practice. Currently the scheme is implemented and tested on Myrinet networks. Another direction of future research can be the implementation of the scheme on other high-performance networks, such as Gigabit Ethernet[6], and Infiniband[11].

BIBLIOGRAPHY

[1] T. Anderson, D. Culler, and Dave Patterson. *A Case for Networks of Workstations (NOW),* IEEE Micro, pages 54-64, Feb 1995.

[2] D. K. Panda and C. B. Stinkers, editors. *Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC),* Lecture Notes in Computer Science, Volume 1362, Springer-Verlag, 1998.

[3] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure,* Morgan Kaufmann Publishers, 1998.

[4] I. Foster, A. Roy, and V. Sander. *A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation,* Int'l Workshop on Quality of Service (IWQOS '00), June 2000.

[5] N. J. Boden, D. Cohen, et al. *Myrinet: A Gigabit-per-Second Local Area Network,* IEEE Micro, pages 29–35, Feb 1995.

[6] R. Sheifert, *Gigabit Ethernet,* Addison-Wesley, 1998.

[7] *GigaNet Corporations,* http://www.giganet.com.

[8] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach.* The IEEE Computer Society Press, 1997.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard,* March 1994.

[10] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. *MPICH-GQ: Quality of Service for Message Passing Programs,* Int'l Conference on Supercomputing (SC '00), November 2000.

[11] *InfiniBand Trade Association,* http://www.infinibandta.com .

[12] A. Gulati, D.K. Panda, P. Sadayappan, P. Wyckoff. *NIC-Based Rate Control for Proportional Bandwidth Allocation in Myrinet Clusters.* In Int'l Conf On Parallel Processing, September 2001.

[13] S. Senapathi, D.K. Panda, D. Stredney, H. Shen. *A QoS Framework for Clusters to Support Applications with Resource Adaptivity and Predictable Performance.*In Int'l Workshop on Quality of Service, May, 2002.

[14] A.Garcia, H. Shen. *An Interleaved Parallel Volume Renderer with PC-clusters.* Submitted to Eugraphics Workshop on Parallel Graphics and Visualization, 2002.

[15] A. Gulati, *A proportional bandwidth allocation scheme for Myrinet clusters.* M.S. thesis, Dept. of Comp. & Info. Science, The Ohio State University, June 2001.

[16] *The GM Message Passing System.* Documentation available at hhtp://www.myri.com/scs/index.html.

[17] W. Gropp, E. Lusk, N, Doss, N. Skjellum. *A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard.* Tech. Report, Argonne National Laboratory and Mississippi State University.

[18] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow. *The NAS Parallel Benchmarks 2.0.* Tech Report NAS-95-020, December 1995.

[19] J. H. Kim, A. Chien. *Rotating Combined Queuing (RCQ): Bandwidth and latency guarantees in low-cost, high-performance networks,* In Proceedings of the Int'l Symposium on Computer Architecture: 226-236, May 1996.

[20] M. Gerla, B. Kannan, B. Kwan, P. Palnati, S. Walton, E. Leonardi, F. Neri. *Quality of Service support in high-speed, wormhole-routing networks.* In Int'l Conf. On Network Protocols, Oct. 1996.

[21] K. Connelly, A. Chien. *FM-QoS: Real-time communication using self-synchronization schedules.* In Proc. Of Supercomputing conference, San Jose, CA, November 1997.

[22] H. Eberle, E. Oertli. *Switcherland. A QoS communication architecture for workstation clusters.* In Proceedings of ACM ISCA R98, Barcelona, Spain, June 1998.

[23] R. West, R. Krishnamoorthy, W. Norton, K. Schwan, S. Yalamanchili, M. Rosu, S. Chandra. *Quic: A quality of service network interface layer for communication in NOWs.* In Proc. Of the Heterogeneous Computing Workshop, in conjunction with IPPS/SPDP, San Juan, Puerto Rico, April 1999.

[24] I. Foster, C. Kesselmann, C. Lee, B. Lindell, K. Nahrstedt, A. Roy. *A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation.* In 7th IEEE/IFIP International Workshop on Quality of Service, 1999.

[25] I. Foster, A. Roy, V. Sander. L. Winkler. *End-to-end Quality of Service for High-End Applications.* IEEE Journal on Selected Areas in Communications, Special Issue on QoS in the Internet, 1999.

[26] F. Chang, V. Karamcheti, and Z. Kedem. *Exploiting Application Tunability for Efficient, Predictable Resource Management in Parallel and Distributed Systems.* Journal of Parallel and Distributed Computing, Vol. 60, pp. 1420-1445, 2000.

[27] Y. Zhang, A. Sivasubramaniam. *Scheduling Best-Effort and Real-Time Pipelined Applications on Time-Shared Clusters.* In Proceedings of the ACM Symposium on Parallel Algorithms and Architectures(SPAA), pages 209-218, July 2001.

[28] S.Chatterjee, J. Sydir, B. Sabata, T. Lawrence. *Modeling Applications for Adaptive QoS-based Resource Management.* In Proceedings of the 2nd IEEE High Assurance Systems Engineering Workshop, August, 1997.