A Study of the Performance Tradeoffs of a Tape Archive

Jason Xie (jasonxie@cs.wisc.edu)
Naveen Prakash (naveen@cs.wisc.edu)
Vishal Kathuria (vishal@cs.wisc.edu)

Computer Sciences Department
University of Wisconsin-Madison

Introduction

Simulation plays a critical role in understanding tradeoffs in any system. In this project we study the tradeoffs involved in a tape archive of files. The archive consists of tape drives, disk cache with disk drives, and a robot that mounts and dismounts the tapes. Files are stored in tapes and a user can submit requests to retrieve them. The archive moves the requested files from tape to disk and notifies the user. There are two main ways of evaluating the tradeoffs and different policies involved in a tape drive. We can either analyze an actual trace or generate them randomly following some distribution. We followed the later approach in the absence of a trace and the flexibility of generating any kind of job mixes. In this paper we begin with a motivation for our project, followed by a theoretical analysis of the expected results. We then present the model for our experiment with results and conclusion.

Motivation

The goals of this project are to study the time a user order spends in the system, before being notified about the retrieval of the requested files and the system utilization given the workloads. We measure waiting time and service time of workload streams, and how different policies and the amount of resources at hand affect them. The different policies of selecting an order to serve from a list of waiting orders are studied. First In First Out (FIFO), Processor Sharing (PS) and Shortest Job First (SJF) are some of the queuing discipline we intend to study. As the distribution of sizes of files vary depending on the work environment, we will study the affect of different job mix and placement on the results. In a typical university environment, number of small files are much more than the number of large files. Performance also depends on the amount of resources at hand. Varying the number of tape drives and disk cache will affect the turn around time and utilization in different ways.

Theoretical Analysis

The job mix affects the system in that it may vary the co-efficiency variation of the service. Let's suppose the user requests arrive at the system in accordance with the exponential distribution. Each request includes a number of files. Files are of different sizes and stored in the tape archive. Upon receiving requests, the tape drives would fetch the requested files from the tape archive. Thus, the placements and the sizes of the files would affect the systems' service (e.g., the transferring time and the seeking time of files). Namely, the system provides its service based on a certain service distributions.

In a M/M/1 system, where the system's service also follows the exponential distribution, Each service is independent of previous services. Another words, the system is memoryless. The user may request files of various sizes, the requested files may located at the different places in the tape archive, Namely the services' co-efficiency variation may differs greatly from one request to another. However, the average length in the WaitQueue would is relatively constant because of the memorylessness feature of the service distribution. The average length of the WaitQueue would remain constant as we vary the queuing disciplines.

In contrast, in a M/G/1 system, where the system's service follows a general distribution and no process sharing in the WaitQueue, the number of the request awaiting the system services would fluctuate greatly as the job mixes vary. Typically, the expected number of requests would increase as the services' co-efficiency variation grows. In our experiments, we implemented a tape archive system, whose service does not follow exponential distribution. We expect to observe the growth in the queue size as we vary the job mixes.

The amount of resources available in the system would affect the performance from the user perspective. In a M/G/K system, where the system has K resource (e.g., K tape drives in our experiments) and the request streams arrive at a certain rate and follow the exponentail distributions, The amount of resource in the system represents the ability of the system to serve the requests. It elevates the level of concurrency in the system as the amount of resource increases. The more resource the system has, the more quickly the system can serve the requests. Consequently, the user requests have lower wait time. The average length of the wait queue decreases. However, if the combined service rates are greater than the requests' arrival rate, the system is under-utilized.

Queuing disciplines impact on the system performance via its choice of which request to service the next. In our implementation, SJF estimates the time system takes to service each request depending on the file size, and then uses the arrival time of the request to estimate the time it would depart, assuming zero waiting time. It then chooses the earliest departing request. Thus it tries to prioritize the requests that arrived early and have requested short files to be retrieved from the archive. It has a low wait time and is fair to jobs that arrived early.

FIFO simply retrieves the request at the head of the WaitQueue based on the request's arrival time at the system. Thus it is possible for later arrived small requests to be queued in the WaitQueue behind a certain earlier arrived large request. Consequently, the small requests' normalized wait time with respect to their service time is smaller than that of large requests. It is not a fair system.

The PS tries to be fair as it picks the user orders depending on the user id. It serves every user in rotation. So if there are n different orders with different user id, then one order of each of these n users will be served before any other orders. In this system, a user can consume disproportionate amount of resources by requesting very large files. But he will not be able to hog the system by requesting many files in the single request, as the split before being put in the waiting queue.

Model

We are to measure wait time, service time and wait queue length of different components in the system given a particular job streams and the amount of available system resources (e.g. number of tape drivers). By varying the queuing discipline, we receive different set of values with respective to these parameters. We then study the effects of the queuing disciplines on these parameters.

With these objectives in mind, we build a discrete event driven simulator. It contains a generator, wait queue, tape archive, disk cache and CountDownClock. The components interact in a well-defined manner to uniquely determine the values of the descriptive variables at any give model time. The CountDownClock provides the time services to the entire system. It manages the advance of the model time, and activates events at designated model time.

Generator

The generator generates workload at exponential arrival rate of (100-job)/(time unit). The workload represents the service requests from a community of users. Each user has a user id ranging from 1 to 10. The generator is capable of producing workloads of an arbitrary large number of orders (UserOrder). Each UserOrder has a unique order Id. every user order contains requests for 10 to 20 different files. The file numbers represents the files the users want to retrieve from the system. We generate 1000 files for archiving purpose. With every file number we have an associated tape and a position on it where the file is located. We also store the length of the file with file number. These numbers help in calculating the time it will take to retrieve the file from the archive.

We simulate the exponential distribution of the arrival time of jobs with a uniform pseudo random number generator. We first project a random number, and then map it to another random number via the inverse function of the exponential probability distribution function. The resulting sequence of numbers is the arrival time of the user requests to the system.

When a user order is generated, a corresponding arrival event is scheduled. The arrival event is added to a list of events, which are sorted by the time at which they will be fired. A CountDownClock simulates the firing of all the events in the system. We could have had different CountDownClocks for different kinds of events and the first event to occur will happen at the minimum of these clocks. Instead we simulated different CountDownClocks by having a single list of events with multiple type of events scheduled at any arbitrary model time. In this model, events can happen simultaneously. Since we are calculating different statistics without the constraints in a real system in real time, we could do it at ease, without compromising our results. The arrival event is activated at the time when the UserOrder arrives at the system.

CountDownClock

CountDownClock provides the time services to the components in the system. It maps the wall clock time to the discrete event model time, and it triggers the events thereby advancing the state of the system with time. We can simulate concurrent events by our CountDownClock, by assigning the same firing time for the events. There are three different types of event in our system. We have already discussed the arrival event. The other two are tape event and disk event. Tape event means that a tape is done with processing a request. Similarly disk event means that a user has read the files from the disk. All these events are derived from a base class called event. Each one of them has a firing time at which they are triggered and different state variables are updated. When an arrival event is triggered, it either schedules a tape event if there is one free tape drive to service this request, or puts this request into a list of orders waiting for a tape drive to become free. When a tape event is triggered, it schedules a disk event for this order and a tape event for a new order if the waiting queue for tape drive is non-empty. The disk event will schedule another disk event for a new order if there is one waiting for a disk.

The system is a non-preemptive. It does not facilitate mechanisms such as rollback. Namely, the components in the system obtain all the necessary state variables before accurately determine the next scheduled events (its event types and fire times). The components register the events onto the CountDownClock. When the CountDownClock advances time, it activates the events at their designated time. The active events in turn produce one or more events at some future times. For events scheduled at the same model time, the CountDownClock executes them in any order without advancing the modeled time.

By executing the events, the CountDownClock advances the state of the system. The events also enable interactions among different components of the system. In essence it is the driving force of the simulation.

WaitQueue

Each user requests multiple files to be retrieved in a single order. When requesting the tape drive for retrieval, the order is split into multiple requests depending on the number of files requested. When a tape order comes, it goes to either a free tape drive or in the absence of any free drive it is added to the waiting queue. A similar waiting queue is maintained for the disk orders. When a tape drive becomes free it will pick one of the orders depending on what queuing discipline it follows. The queuing discipline followed by the tape drives is independent of that followed by the disk drives. The different queuing disciplines with which we experimented include First In First Out (FIFO), Processor Sharing (PS) and Shortest Job First (SJF). We measure the impact of these disciplines on the performance of the system and the wait time experienced by the user. There are always two orthogonal issues involved with such performance analysis. On the one hand we try to optimize the performance of the system, while on the other hand we try to be fair to users. As a User Order is split into multiple tape orders (and a disk order for every tape order), we need to group them by the order id after they have been served for per-user statistics.

The FIFO, as the name suggests, serves the jobs in the sequence they arrived. If the jobs arrive at the same time (like multiple tape orders for the same user order), then we randomly pick any one of them to serve next. The queue is actually implemented as a vector in Java, and we just remove the element at the 0th position for serving next. FIFO is unfair to short jobs, as longer jobs at the head of the queue will hog the system for a long time. Also the average turn-around-time is not optimal. Relatively speaking, shorter jobs have to wait more than the longer ones. But this discipline is very easy to implement in the system and does not require much CPU cycles.

The PS tries to be fair as it picks the user orders depending on the user id. It serves every user in rotation. So if there are n different orders with different user id, then one order of each of these n users will be served before any other orders. In this system, a user can consume disproportionate amount of resources by requesting very large files. But he will not be able to hog the system by requesting many files in the single request, as the split before being put in the waiting queue.

The SJF discipline is to retrieve the shortest request in the WaitQueue and service it. It estimates the time it will take to service each request depending on the file size, and then uses the arrival time of the request to estimate the time it would depart, assuming zero waiting time. It then chooses the earliest departing request. Thus it tries to prioritize the requests that arrived early and have requested short files to be retrieved from the archive. It has a low turn around time and is fair to jobs that arrived early.

Tape Drive

We vary the number of tape drives to measure the affect of resources on utilization and the total time taken by a request to be served. Number of tapes in the system remains unchanged and is initialized depending on the number of files and their length. At the time of initialization we generate n files with random length between some minimum and maximum. We also change the mix of these files. We generate many small files compared to the number of large files, to simulate university environment and see it affects on the performance from both the user's and system's perspective. These files are assigned an incrementally increasing number as a file id and are assumed to be laid on the disk in the same order. Since the files are requested randomly, this does not affect the generality of the system. When we are done with filling a tape, we assume a new tape for the subsequent file numbers, until this too gets filled. We repeat this process till all the generated files have been consumed. We maintain the information about the file length, which tape and at what position on the tape this file is located, with the file number in a table. We use this information to calculate the service time for each request.

When there is a request in the waiting queue, each of the tape drive is busy. When a tape drive is done with servicing a request it picks next order to serve from the waiting queue depending on a given queuing discipline. It check if the new file requested is on the tape it currently has, otherwise it dismounts it after rewinding to the middle of the tape, and mounts the tape that contains this file. The head of this tape drive is assumed to be at the middle of this newly mounted tape. Thus we add the cost of rewinding the tape to its middle position if a tape has to be dismounted for the new tape, to the cost of servicing this new request. If the new file requested is on the tape used to serve the old request, then we do not add any dismounting or mounting time, but only the time it takes to position the head at the starting of the file from its previous position. The transfer time is added to the servicing time of all the requests and this is proportional to the file length.

Disk Cache

The Disk Cache serves as a buffer between the tape drive and the user. There are a fixed number of disks in the system and at any point, multiple tape drives can be in the process of flushing their buffers onto disk and there can be multiple users each reading multiple files from the disk cache.

After reading the file from the tape, the tape driver initiates a file transfer from the tape drive buffer to the disk. The disk driver checks if there is a disk available that has sufficient free capacity to accommodate the new file. If there is one such disk then it is allocated to this request and it is marked busy. If there are no disks available then the request from the tape is enqueued into a DiskOrderQueue. When a tape drive is finished flushing its buffer onto the disk, the corresponding user (which requested this file) is notified. In our model, that user immediately starts reading the file. The disk is marked busy at this point. When the user is finished reading the file, the DiskDriver adds that disk into the pool of free disks and deletes the file from the disk. Then it looks up the queue for jobs that can be satisfied by this disk. If there is one such job then this disk starts servicing it, otherwise the disk remains idle till the disk driver receives a request which could be serviced by it.

Experiments

The metrics of a system are represent by the system utilization and the wait time of each user request. The tapes and the disks have very different throughput. In order to balance the service rate between the tapes and the disk, we maintain a 10:1 ratio between the number of tapes and the numbers of disks that is, for every disk, there are 10 tapes. The ratio is derived from the data transfer rates of a Seagate tape and disk product. To answer the questions we posed and to verify the theoretical intuition we beard, we conducted the following experiments

First, we vary the number of tape drives and disk caches so to observe the changes in expected waiting time and system utilization for different queuing disciplines. Given the generated workload, we identify the resources necessary for the system to operate in the steady state (around 75% of utilization). We then increase the resources to observe which of the queuing disciplines is successful in decreasing the waiting time the most.

Second, we introduce different mixes of requests by varying the size of the requested files. The tape archive contains information on the file size for a given file. The sizes of files in the cache are chosen in the range from 10M to 10K. As a result, coefficient of variation of service times in the system is quite high. We study the average queue length under these conditions.

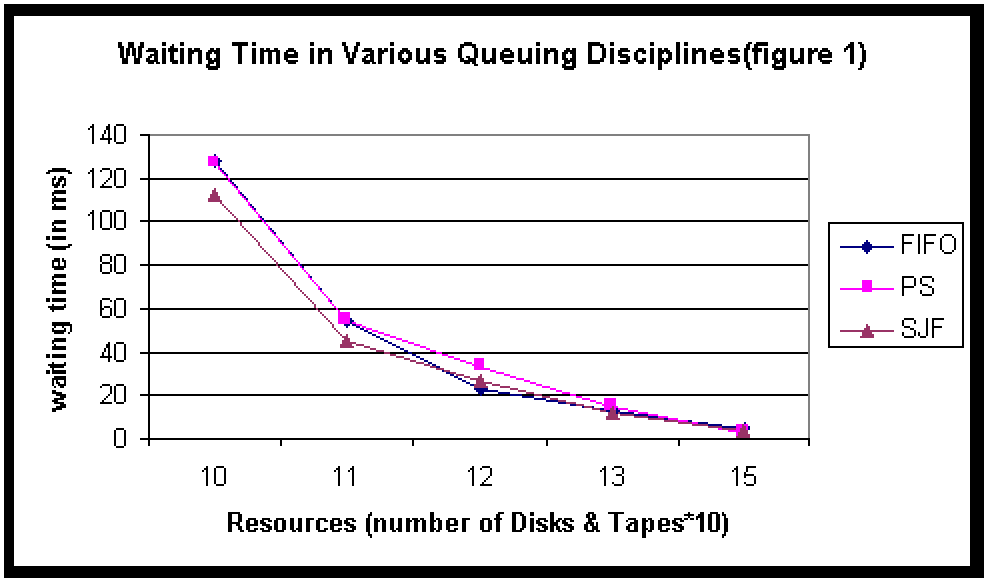
Third, the impacts of queuing disciplines are studied. For a given workload, the WaitQueue employs FIFO, SJF and PS respectively. We measure the differences of the wait time and service time for the user requests.

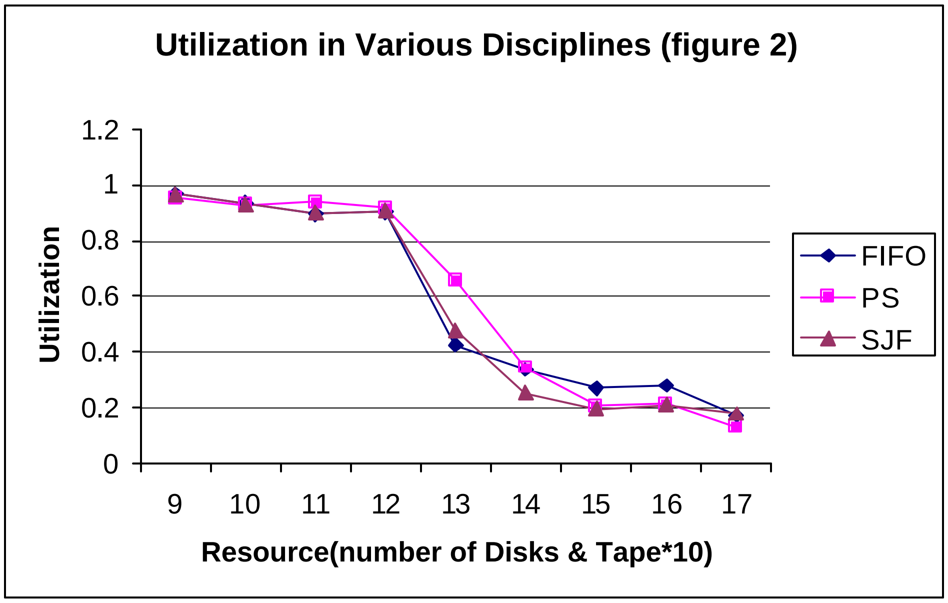
Results

In the figure1, we increase the amount of resource in the system, and submit the same workload to the system. We observe a decrease of wait time for all disciplines. Also, as the wait time decrease, the system utilization also decreases (figure 2).

Among the three queue disciplines, SJF consistently performs better than other disciplines in terms of waiting time for requests. This confirms our earlier conjectures about SJF. The performance of FIFO comes close to SJF as resources are increased. This is reasonable because the system utilization goes down as the resource increases. There are always resource awaiting the newly arrived requests. Therefore, the earlier arrived large requests will not block the late arrived smaller requests.

As far as the utilization, it remains high when there is limited amount of resources in the system. However, it decreases drastically when more resources are added to the system. As showed in the figure 2, the utilization drops sharply while adding the 12th through 15th disks and corresponding tapes. Subsequently, the slope again flattens out as we keep adding resources.





Overall, the queuing discipline matters less if the amount of resource is ample. This is due to the reductions in contentions. Otherwise, SJF yields the lowest wait time for the user requests.

Conclusions and Future Work

We simulated a small number of aspects of a real tape archive and it demonstrated how, even in a simplified simulation, the performance is a complex function of resources and their characteristics, scheduling discipline, workload distribution etc. There is nothing like a single policy that performs the best for all workloads and number of resources. But a detailed simulation of the actual system can help one test some “intelligent” scheduling policies that take into account the nature of the system.

There were many other features that could have been stimulated and taken into consideration by the scheduling policy. A policy could look at all the files that are there in the cache and pick up those requests first and service them. The policy could group together all the requests to a tape and service them through a single read of the tape if the mount and unmount time is extremely high.