A Lazy Scheduling Scheme for Hypercube Computers*

Prasant Mohapatra
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011

Chansu Yu and Chita R. Das
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802

* This research was supported in part by the National Science Foundation under grant number MIP-9104485. A preliminary version of this paper was presented at the International Conference on Parallel Processing, August, 1993.

Proposed running head:

Lazy Scheduling for Hypercube Computers

Mailing Address for sending proofs:

Prasant Mohapatra
Department of Electrical and Computer Engineering
201 Coover Hall
Iowa State University
Ames, IA 50011

Tel. (515) 294-3959

Abstract

Processor allocation and job scheduling are two complementary techniques for improving the performance of multiprocessors. It has been observed that all the hypercube allocation policies with the FCFS scheduling provide only incremental performance improvement. A greater impact on the performance can be obtained by efficient job scheduling. This paper presents an effort in that direction by introducing a new scheduling algorithm called *lazy scheduling* for hypercubes. The motivation of this scheme is to eliminate the limitations of the FCFS scheduling. This is done by maintaining separate queues for different job sizes and delaying the allocation of a job if any other job(s) of the same dimension is(are) running in the system. Processor allocation is done using the buddy strategy. The scheduling and allocation complexity is $O(n)$ for an n-cube. Simulation studies show that the performance is dramatically enhanced by using the lazy scheduling scheme as compared to the FCFS scheduling. Comparison with a recently proposed scheme called *scan* indicates that the lazy scheme performs better than the scan scheduling under a wide range of workloads.

1. Introduction

Processor management in multiprocessors is a crucial issue for improving the system performance. This has become an active area of research for the hypercube computer, which has emerged as one of the most popular architectures [3,7,14]. Hypercube topology is suitable for a wide range of applications and can support multiple users. Judicious selection of processors is essential in a multiuser environment for better utilization of system resources. There are two basic approaches to improve processor management in a multiprocessor system. These are called *job scheduling* and *processor allocation*.

Scheduling decides the job sequence for allocation. Processor allocation is concerned with the partitioning and assignment of the required number of processors for incoming jobs. Structural regularity of the hypercube makes it suitable for partitioning into independent subcubes. Each user or job is assigned an appropriate subcube by the operating system. It is known that optimal allocation in a dynamic environment is an NP-complete problem [6]. Several heuristic algorithms reported in literature are buddy [10], modified buddy [2], gray code [4], free list [9], MSS [6], tree collapsing [5], and PC-graph [16]. These schemes differ from each other in terms of the subcube recognition ability and/or time complexity. Better subcube recognition ability reduces external fragmentation and in turn improves the system utilization. However, complexity of an allocation algorithm increases with the recognition ability. Buddy scheme has low time complexity but it does not have the complete subcube recognition ability. On the other hand, schemes like MSS and PC-graph have the complete subcube recognition ability at the cost of high time complexity.

Comparison of all the hypercube allocation policies shows that the performance improvement due to better subcube recognition ability is not significant [9,12]. The difference between a policy having perfect subcube recognition ability and the simple buddy scheme is minimal. This is mainly because of the first-come-first-serve (FCFS) discipline used for job scheduling. In a dynamic environment, FCFS scheduling may not efficiently utilize the system. There are two drawbacks with the FCFS scheme. First, it is more likely that an incoming job with a request for a large cube has to wait until some of the existing jobs finish execution and relinquish the nodes to form a large cube. All arriving jobs are queued

during this waiting period. There may be several jobs waiting in the queue with smaller cube requests but they cannot be allocated even though the system can accommodate such jobs. This *blocking* property of the FCFS scheme reduces the system utilization. Since the blocking job needs a part of the system (unless the request is for an n-cube), it is not necessary to stop allocating other jobs to the rest of the system. Second, with the FCFS scheduling scheme, a job is usually attempted for allocation as soon as it arrives. This *greedy* property creates more fragmentation and makes the allocation of the succeeding jobs difficult. The subcube recognition ability of an allocation policy is thus overshadowed by the limitations of the FCFS scheduling policy.

It is therefore logical to focus attention on efficient scheduling schemes to improve system performance while keeping the allocation complexity minimal. Little attention has been paid towards scheduling of jobs in a distributed system like hypercube. The first effort in this direction was by Krueger et al [12]. They propose a scheme called *scan*, which segregates the jobs and maintains a separate queue for each possible cube dimension. This eliminates the blocking problem associated with the FCFS scheme. The queues are served similar to the c-scan used in disk scheduling. It is shown that a significant performance improvement can be achieved by employing the scan policy. However, it turns out that the scheme is suitable for a workload environment where the job service time variability is minimal. For a more general workload, the system fragmentation increases and thus the performance gain with scan diminishes. This is attributed to the greedy characteristic of the scheme. While processing a queue, scan attempts to allocate the jobs to the system as and when possible.

In this paper, we propose a new strategy called *Lazy Scheduling* for assigning jobs in a hypercube. The main idea is to temporarily delay the allocation of a job if any other job(s) of the same dimension is(are) running in the system. The jobs could wait for existing subcubes rather than acquiring a new subcubes and possibly fragmenting the system. The scheduling is not greedy and is thus named *lazy* . Separate queues are maintained for different job sizes. We thus eliminate the blocking problem associated with the FCFS scheme. Waiting time in the queue is controlled by a threshold value in order to avoid discrimination against any job size. Ideally, for a non empty queue there is at least one

subcube in the system executing a job of the corresponding size. The proposed scheme tries to improve throughput by providing more servers to the queues that have more number of jobs.

A simulation study is conducted to compare mainly three types of processor scheduling schemes for hypercubes. These are FCFS, scan, and lazy. Job allocation is done using the buddy scheme for all the three disciplines. Another simple scheme called static partitioning is also simulated to demonstrate its effectiveness for uniform job size distribution. The performance parameters analyzed here are average queueing delay, system utilization, throughput, system power (throughput/delay), and job completion time. First, it is shown that the system performance can be boosted by using an efficient scheduling scheme other than FCFS. Next, we show that the lazy scheme provides at least equal performance as that of scan for uniform residence time distribution. For all other workloads, including the most probable residence time distributions like hyperexponential, the lazy scheme out performs scan in all performance measures by 20% to 50%. Thus, the proposed scheduling scheme is suitable and adaptive for a variety of workloads. Finally, by using an allocation scheme with complete subcube recognition ability, we demonstrate the minimal affect of allocation schemes. We therefore argue in favor of using a good scheduling mechanism rather than an expensive allocation policy for hypercubes.

The rest of the paper is organized as follows. In Section 2, subcube recognition ability and time complexity of various hypercube allocation policies are summarized. The scan scheduling and its limitations are also discussed. The lazy allocation scheme is described in Section 3. The simulation environment is described in Section 4. Section 5 is devoted to the performance evaluation and comparisons of various policies. Conclusions are drawn in Section 6.

2. Allocation and Scheduling Policies

2.1. Allocation Policies

Buddy

The buddy strategy is implemented on the nCUBE system and is based on the buddy scheme for storage allocation [10]. For an n-cube, 2^n allocation bits are used to keep track of the availability of nodes. Let k be the required subcube size for job I_k. The idea is to find the least integer m such that all the bits in the region $[m2^k, (m+1)2^k - 1]$ indicate the availability of nodes. If such a region is found, then the nodes corresponding to that region are allocated to job I_k and the 2^k bits are set to 1. When a subcube is deallocated, all the bits in that region are set to 0 to represent the availability of the nodes. The buddy strategy is conceptually simple and statically optimal [4]. This scheme does not have complete subcube recognition ability in a dynamic environment. Allocation and deallocation time complexities are $O(2^n)$ and $O(2^k)$, respectively. The allocation and deallocation complexity can be reduced to $O(n)$ by using an efficient data structure [11] as is described in Section 3.

Modified Buddy

The modified buddy scheme is similar to the buddy strategy in maintaining 2^n allocation bits. Here, the least integer α is determined, $0 \leq \alpha \leq 2^{n-k+1} - 1$, such that α^{n-k+1} is free and it has a pth partner, $1 \leq p \leq (n-k+1)$, α_p^{n-k+1} which is also free. Detail description of the scheme is given in [2]. This scheme has better subcube recognition ability than buddy. The complexities of allocation and deallocation are $O(n2^n)$ and $O(2^k)$, respectively. An extension of this scheme for complete subcube recognition is done using a buddy tree [1]. A fault-tolerant variant of the buddy strategy was reported by Livingston and Stout by using a cyclic buddy grouping scheme [13].

Gray Code

The gray code strategy proposed in [4] stores the allocation bits using a binary reflected gray code (BRGC). Here the least integer m is determined such that all the (i mod 2^n) bits indicate availability of nodes, where $i \in [m2^{k-1}, (m+2)2^{k-1} - 1]$. Thereafter, the allocation and deallocation are the same as in the buddy scheme. A single gray code (GC) cannot recognize all the available subcubes. For complete recognition, $\binom{n}{\lfloor n/2 \rfloor}$ gray codes are needed. The complexity of the multiple GC allocation is $O(\binom{n}{\lfloor n/2 \rfloor}2^n)$ and that of deallocation is $O(2^k)$.

Free List

The free list strategy proposed in [9] maintains $(n+1)$ lists of available subcubes, with one list per dimension. A k-cube is allocated to an incoming job by searching the free list of dimension k. If the list is empty, then a higher dimension subcube is decomposed and is allocated. Upon deallocation, the released subcube is merged with all possible adjacent cubes to form the largest possible disjoint subcube(s). The list is updated accordingly. This scheme has the ability to recognize a subcube if it exists in the system. The time complexity of allocation is $O(n)$ and that of deallocation is $O(n2^n)$.

MSS

This strategy is based on the idea of forming a maximal subset of subcubes (MSS), and is described in detail in [6]. The MSS is a set of available disjoint subcubes that has the property of being greater than or equal to other sets of such subcubes. The main idea is to maintain the greatest MSS after every allocation and deallocation of a subcube. Conceptually free list and MSS are the same except that the free list does not maintain MSS after allocation. Determination of the best decomposition is a major overhead of this scheme. MSS allocation and deallocation complexities are of the order of $O(2^{3^n})$ and $O(n2^n)$, respectively.

Tree Collapsing

Tree collapsing strategy, introduced in [5], involves successive collapsing of the binary tree representation of a hypercube structure. The nodes that form a subcube but are at a distant are brought close to each other to facilitate recognition. This scheme generates its search space dynamically and has complete subcube recognition ability. Its allocation and deallocation complexities are $O(\binom{n}{k}2^{n-k})$ and $O(2^k)$, respectively.

PC-graph

The main idea of this approach is to represent available processors in the system by means of a prime cube (PC) graph [16]. Subcubes are allocated efficiently by manipulating the PC-graph using linear graph algorithms. Relationship between free subcubes are maintained in this graphical approach. This scheme also has complete subcube recognition ability. The allocation complexity is $O(\frac{3^{3n}}{n^2})$ and that of deallocation is $O(\frac{3^{2n}}{n^2})$.

2.2. Performance Comparison

It is observed from [9, 12] that the performance variations due to different allocation policies are minimal. None of the allocation policies utilizes more than half of the system capacity irrespective of the input load [12]. Fragmentation of the system overrides the advantages obtained from better subcube recognition ability. More sophisticated allocation policies, although show little performance improvement, introduce overheads and higher time complexity. We have therefore adopted the buddy allocation scheme, which is simple and has low time complexity.

2.3. Scheduling Strategies

Fragmentation is dependent not only on the allocation policy but also on the order in which the jobs are assigned. Ordering of the jobs is determined by the scheduling policy. Performance is degraded in the FCFS scheduling when small cube requests can not be allocated to the available subcubes because of a larger job ahead in the queue. System performance can be improved if the available subcubes can be allocated to the smaller jobs in queue by adopting a different scheduling policy.

A scheduling strategy called *scan* is proposed in [12]. Conceptually, it is similar to the disk c-scan policy. The algorithm maintains $(n + 1)$ FCFS separate queues, one for each possible cube dimension. A new job joins the queue corresponding to its subcube dimension. All jobs of a given dimension (including those that arrive when the queue is in service) are allocated before the scheduler move on to the jobs of next dimension. The authors propose two disciplines: *ScanUp* and *ScanDown*. When queue i is empty, ScanUp allocates jobs from queue $(i + 1)$ modulo $(n + 1)$, while ScanDown continues with $(i - 1)$ modulo $(n + 1)$. The scan scheme could suffer from indefinite postponement. Starvation

could be avoided by disallowing allocation of jobs that arrive after the scheduler starts allocating from the queue. These jobs can be queued until the scheduler scans the queue again. It is shown that significant performance improvement can be achieved with this scheme compared to the most sophisticated allocation policies.

Problems associated with the scan policy are the following. It tries to reduce fragmentation by allocating equal-sized jobs. The scheme performs well when the job residence time has little variation. In this environment, the system at any instant would have jobs of almost the same size and thus fragmentation is reduced. Usually the residence times of all the jobs are not the same. Thus, dynamic allocation and deallocation eventually creates a mixture of jobs from different queues, which lead to fragmentation. Performance of the scan scheme is therefore dependent on the workload. The study in [12] is conducted with exponential job residence time distribution with a mean of one time unit. Normally, job residence time distribution resembles close to hyperexponential distribution [15]. It will be shown that the scan algorithm does not performs well in this environment. Furthermore, under certain circumstances scan treats jobs unfairly. This happens if there are some queues with large number of jobs and some queues with a few jobs. This is possible with non-uniform arrival pattern. There could be a situation where jobs in a shorter queue have to wait until all the jobs in the longer queues are processed. In such cases, a job arriving early to a shorter queue has to wait even longer than the jobs that arrive much later in the longer queues.

3. Lazy Scheduling Scheme

In this section, we first discuss a simple scheduling algorithm based on static partitioning of the system. Static partitioning scheme is shown to be efficient only for uniform workloads. Next, the concept of *Lazy Scheduling* is discussed. The lazy scheduling is an adaptation of the static scheme in a dynamic environment.

3.1. Static Partitioning

Static partitioning scheme divides an n-cube system into one $(n-1)$-cube, one $(n-2)$-cube,...,one 1-cube, and two 0-cubes. Let S_i denote the partition which accommodates jobs of cube size i, for $0 \leq i \leq n-1$. There is a queue Q_i associated with each S_i. Thus there are n queues in an n-cube system. An incoming job, I_i, requesting an i-cube joins Q_i. Each queue is scheduled on a FCFS basis. The steps for subcube request and release are given below.

Static Partitioning Algorithm

Static_Request (I_k)

1. If Q_k is empty, then allocate S_k to I_k.
2. Else, enqueue I_k to Q_k.

Static_Release (I_k)

1. If Q_k is not empty, then allocate the header of Q_k to S_k.

3.2. Lazy Scheduling

Lazy scheduling is based on two key concepts. First, a job requiring a small subcube is not blocked behind large jobs. The scheduler maintains a separate queue for each dimension. This avoids the blockings incurred in the FCFS scheduling. Second, a job tends to wait for an occupied subcube of the same size instead of using a new cube and possibly fragmenting a larger subcube. The greedy characteristic of the FCFS policy is thus subdued. Both these issues help in reducing fragmentation. If all the jobs of a particular dimension wait for a single subcube executing in the system, then the scheduling resembles the static partitioning scheme. In order to avoid this, we introduce a variable threshold length for each queue. A queue whose length exceeds the threshold value, tries to find another subcube using an allocation policy. This provides more servers for the queues that have more incoming jobs. The threshold value is determined dynamically as is explained next.

Fig. 1. Queue management in lazy scheduling in a 4-cube.

Determination of the threshold value, N_k, is based on the concept that for every subcube allocated in the system, there can be another job waiting to acquire it. For example, if there are two 1-cubes allocated to the system, N_1 is set to 2. Then, the next two jobs requiring 1-cubes are enqueued. Thus, the length of the queue, $|Q_1|$, becomes 2. The jobs waiting in the queue are guaranteed to receive service after the currently occupied 1-cube jobs are executed. Any additional request for a 1-cube makes the length, $|Q_1|$, more than the threshold, N_1. Thus, one more 1-cube is searched for allocation. The scheduler therefore creates more servers for a queue if the number of requests for the corresponding

cube is high. Although the scheme tries to limit the number of jobs waiting for the existing subcubes, there may be more jobs in the queue at high load.

Some jobs may suffer indefinite postponement under certain distribution of workload. Our algorithm is modified to eliminate the possibility of indefinite postponement by using a threshold value for the maximum queueing delay that a job can tolerate. Whenever the waiting time of a job reaches the threshold, it gets priority over all other jobs. No jobs are allocated until the job whose waiting time has reached the threshold is allocated.

The threshold for the queueing delay could be predefined or computed dynamically. Predefined threshold value is useful in imposing deadline for job completion. A dynamic threshold for the queueing delay is derived based on the following heuristic. Let d_t be the average queueing delay for a job at time t. During this waiting period the job 'sees' the arrival of $(d_t \cdot \lambda)$ jobs to the system, where λ denotes the arrival rate. We consider that the maximum delay that a job can tolerate is the processing of these $(d_t \cdot \lambda)$ jobs. This time is equal to $(d_t^2 \cdot \lambda)$ and is used as the threshold value. The average queueing delay and the arrival rate are monitored by the scheduler. The scheduler updates the threshold value periodically or every time a job is allocated to the system.

The algorithm for the lazy scheduling follows is given below. Buddy_Request and Buddy_Release are procedures from the buddy allocation algorithm presented later. The flag *stop_alloc* indicates that the waiting time of a job has reached the threshold value.

Lazy Scheduling Algorithm

Lazy_Request (I_k)

1. Enqueue I_k to Q_k.
2. If ($|Q_k| > N_k$) and (*stop_alloc* is FALSE), then call Buddy_Request(header of Q_k).
3. If succeeds, increment N_k.

Lazy_Release (I_k)

1. Determine the oldest request. If the waiting time exceeds the threshold, then set *stop_alloc* flag to TRUE and save identity of queue (Q_j).
2. If $(|Q_k| > 0)$ and (*stop_alloc* is FALSE), then allocate the released subsystem to the header of Q_k.
3. If *stop_alloc* is TRUE, then call Buddy_Request(header of Q_j). If success, then set *stop_alloc* to FALSE.
4. Call Buddy_Release(I_k).

An incoming job is first handled by the scheduler, which manages the queues and imposes the algorithmic procedure. Jobs are allocated to the system using the underlying allocation policy.

All the schemes considered in this paper use buddy strategy for allocating jobs. We implement the buddy strategy using an efficient data structure as proposed in [11]. This structure maintains a separate list, F_i, for each cube size i. Each list maintains the available subcubes for the corresponding size. Initially all the lists except F_n are empty. List F_n contains the n-cube. The allocation and deallocation complexities are $O(n)$. The algorithm is presented below. I_k represents a job that requires a k-cube. The complete processor management structure is illustrated in Figure 2.

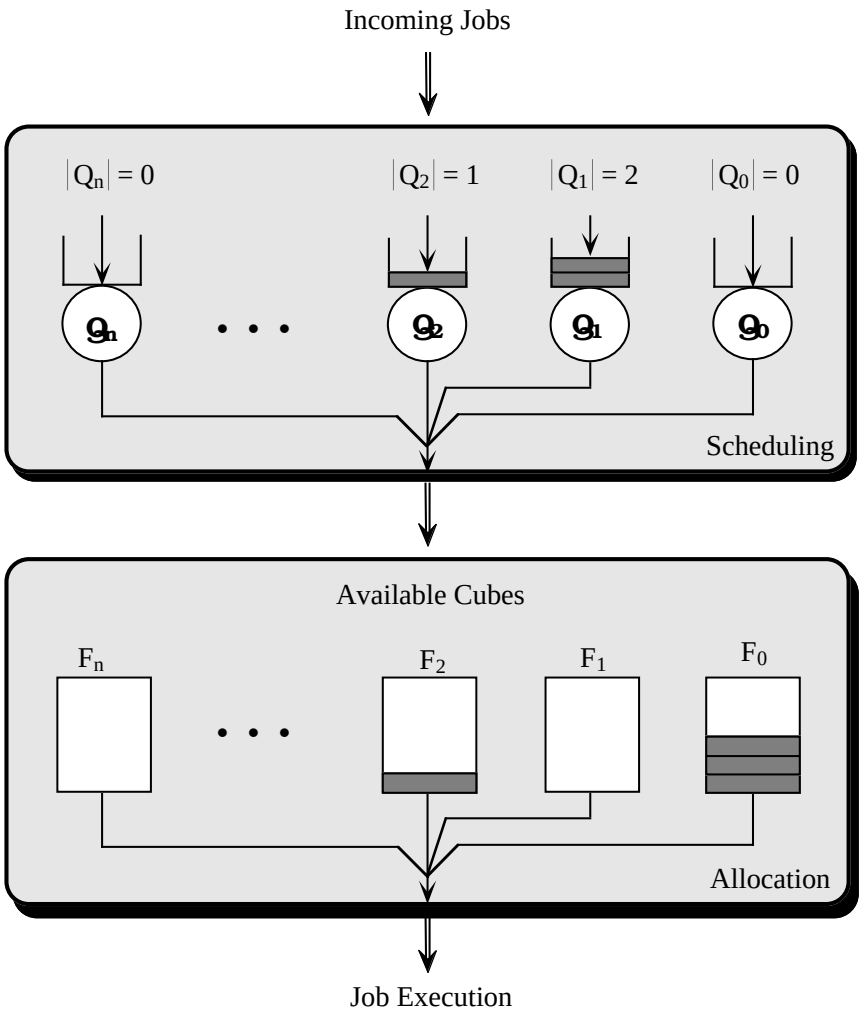


Fig. 2. Processor management in an n-cube.

Buddy Strategy

Buddy_Request (I_k)

1. If F_k is not empty, allocate a subcube in F_k to I_k.
2. Otherwise, search $F_{k+1}, F_{k+2}, ..., F_n$, in order until a free subcube is found.
3. If found, decompose it using the buddy rule until a k-cube is obtained and allocate it to I_k. Update the corresponding lists after decomposition.
4. Else enqueue the job to the corresponding queue.

Buddy_Release (I_k)

1. Insert the released subcube into the list F_k.
2. If F_k contains the buddy of I_k, merge them and insert the new cube into the list F_{k+1}.
3. Repeat step 2 until the corresponding buddy is not available.

3.3. Complexity Analysis

Time complexity of the lazy scheduling scheme is dependent on the complexity of the underlying allocation policy. For job allocation with the lazy scheduling, when a new job arrives, the queue length is compared with the threshold value and the *stop_alloc* flag is checked. These operations take constant time. Thus, the allocation complexity is the same as the buddy allocation which is $O(n)$. Determination of the oldest job is done by keeping track of the waiting time of the earliest generated job. This step need a constant time. Deallocation with the buddy scheme takes $O(n)$ time. Therefore the time complexity of the release process of lazy scheduling is $O(n)$ for an n-cube.

4. Simulation Environment

A simulation study is conducted to evaluate the performance of the proposed strategy and for comparison with other allocation and scheduling policies. The other schemes simulated are FCFS, scan, and static partitioning. Buddy allocation policy is employed for all the scheduling schemes. The job arrival rate (λ) is based on the system capacity. This is done to avoid saturation by ensuring that the arrival rate to the system does not exceeds the service rate. For example, an arrival rate of 0.3 translates to $\frac{system\ size}{mean\ job\ size \times mean\ residence\ time} \times rate = \frac{2^6}{10 \times 5} \times 0.3 = 0.32$ arrivals per time unit on a 64-node hypercube with mean residence time of 5 time units and mean job size of 10 processes. The observation interval T is 10000 time units, which is sufficient to obtain the steady state parameters. The simulation results are obtained by taking average of 1000 iterations. The 95% confidence level is within 5% of the mean.

4.1. Workload

The workload is characterized by the job interarrival time distribution, distribution of the job size (subcube size), and distribution of the job service demands. Job arrival pattern is assumed to follow Poisson distribution with a rate λ. The job sizes and their total service demand could be either independent or dependent (proportionally related) of each other.

Independent distribution means that a large job (large subcube) has the same distribution of total service demand as that of a small job. Job residence time for larger jobs is less compared to smaller jobs since the larger jobs use more number of processors. The residence time of a job I_k is computed as $x_k/2^k$, where x_k is the required service time of job I_k, and 2^k is the number of processors needed for the k-cube job. x_k is determined from the total service demand distribution. The mean of the total service demand is computed by multiplying the mean job size with the mean residence time.

With dependent distribution, a large job has more total service demand than a small job. So the residence times of all the jobs have the same distribution. In this case, the job size is first obtained using the given distribution. The residence times of the jobs are obtained from a given distribution and the mean residence time, irrespective of the job size.

Distribution of the job size is assumed to be uniform or normal. In a 10-cube system, for example, probability that a request is for an i-cube (p_i) is $\frac{1}{10}$ with uniform distribution. For normal distribution, area under the standard normal density function from -2.5σ to $+2.5\sigma$ is divided into 10 equally-spaced areas. p_i can be obtained by integrating the ith area and normalizing it in order to compensate the outside area (below -2.5σ and above $+2.5\sigma$). The probabilities for a 10-cube system are, $p_0 = p_9 = 0.017$, $p_1 = p_8 = 0.044$, $p_2 = p_7 = 0.093$, $p_3 = p_6 = 0.152$, $p_4 = p_5 = 0.194$. These numbers are used for simulating normal distribution in our study.

The total service demand follows uniform or bimodal hyperexponential distribution. Bimodal hyperexponential distribution is considered as the most probable distribution for processor service time [15]. Bimodal hyperexponential distribution function is a combination of two exponential distribution functions - one is with a large mean ($\frac{1}{\lambda_l}$) and the other is with a small mean ($\frac{1}{\lambda_s}$). λ_l and λ_s can be determined by three parameters - mean ($\frac{1}{\lambda_m}$), coefficient of variation (C_x) and α, where $\frac{1}{\lambda_l} = \frac{1}{\lambda_m} + \frac{1}{\lambda_m}\sqrt{\frac{(C_x^2-1)\alpha}{2(1-\alpha)}}$ and $\frac{1}{\lambda_s} = \frac{1}{\lambda_m} - \frac{1}{\lambda_m}\sqrt{\frac{(C_x^2-1)(1-\alpha)}{2\alpha}}$. With a probability α, the total service demand is exponentially distributed with mean $\frac{1}{\lambda_s}$, and with a probability $(1-\alpha)$, it is exponentially distributed with mean $\frac{1}{\lambda_l}$. Mean residence time is assumed to be 5 time units. For hyperexponential distribution, α is taken as 0.95, and the coefficient of variation (C_x) for the residence time is set to 4.0.

4.2. Performance Parameters

The following parameters are measured during simulation of various n-cubes for T time units.

G : Number of jobs generated during the observation time T.

C : Number of jobs completed during the observation time T.

A : Number of jobs allocated during the observation time T.

R : Total queueing delay of allocated jobs.

S : Sum of total service demand of generated jobs.

The performance parameters obtained from the simulator are utilization (U) and mean queueing delay (M). Mean queueing delay, M, is equal to R/A. We have also computed throughput(X) as X=C/T. The actual job request rate is measured by S/($2^n \times$ T). System utilization, U, is computed from $\sum_{i=1}^{A} 2^{|I_i|} t_i / 2^n T$, where t_i is the residence time of job I_i. The input request rate is different from the measured job request rate due to round off effect while limiting the job size to cubic.

5. Results and Discussion

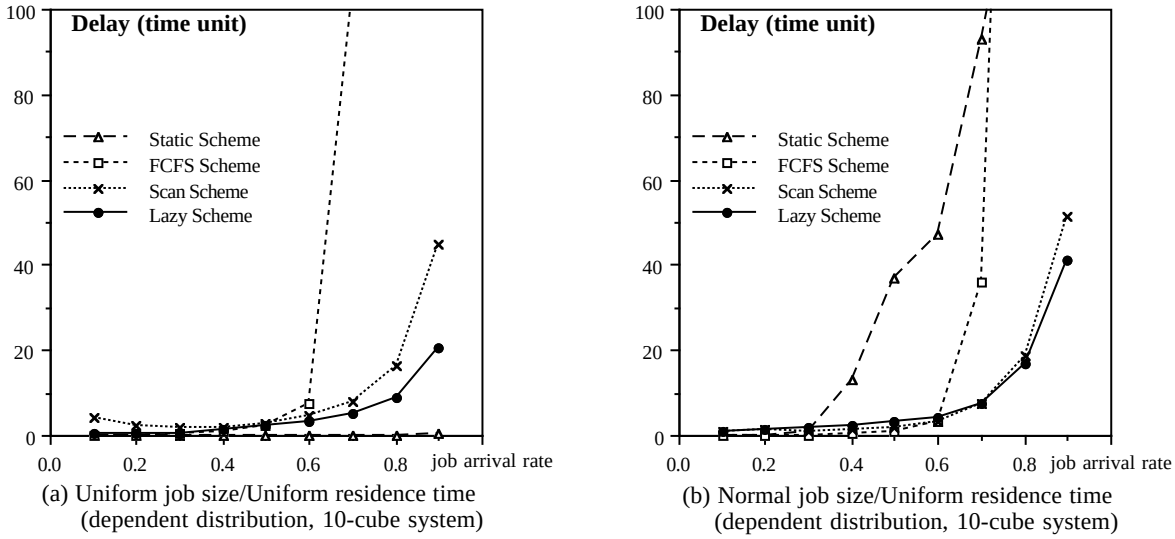
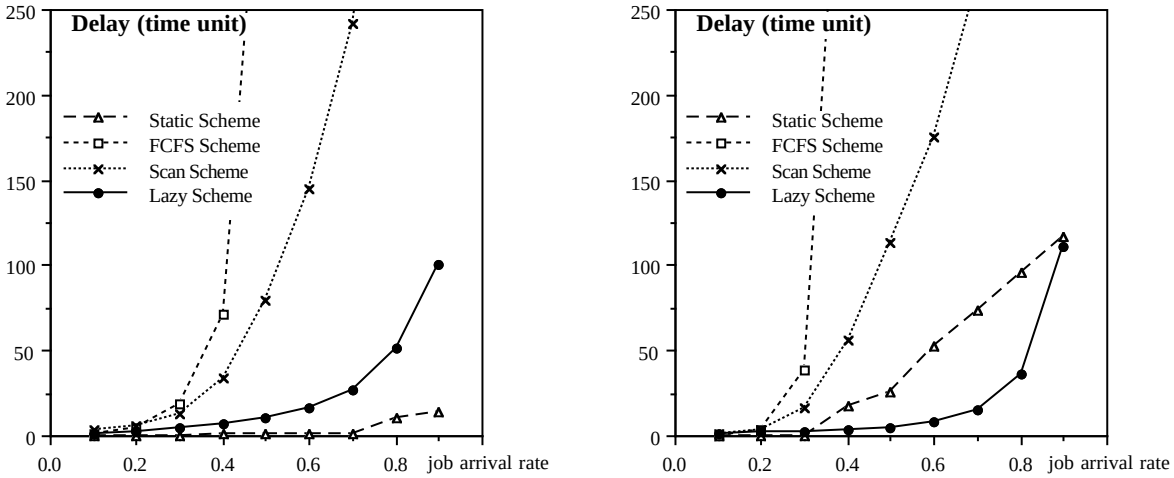


Fig. 3. Variation of average queueing delay for a 10-cube
Mean job size = 5-cube, Mean residence time = 5 time units

Figure 3 shows the variation of queueing delay with respect to input load for a 10-cube system. The job residence time is uniformly distributed. The job size is uniformly distributed in Figure 3(a), and Figure 3(b) shows the variations for a normal job size distribution. We compare static, FCFS, scan, and lazy schemes. The delay saturates early for the system employing the FCFS scheduling strategy in both cases. Static partitioning performs very well in case of uniform job size distribution. System utilization is high and there is almost no fragmentation in static partitioning with uniform distribution. But the performance improvement is not consistent and deteriorates for other distributions. This can be inferred from Figure 3(b). Scan and lazy scheduling show better performance than the FCFS strategy for both uniform and normal distributions. Figures 3(a) and 3(b) show that the average queueing delay with the lazy scheduling is less than that of the scan. Moreover, the lazy scheduling performs close to the static scheme for uniform job size distribution (Figure 3(a)). This is because by delaying the allocation of a job for which a cube is already busy, the lazy scheme tries to divide the system uniformly for all cube sizes.

It was mentioned in Section 2 that the scan policy is workload dependent. It does not perform well when the job residence time exhibits wide variation. The difference in the delay between lazy and scan becomes more prominent when the job residence time is hyperexponentially distributed. This is demonstrated in Figure 4. The job size is uniformly distributed in Figure 4(a), and is normally distributed in Figure 4(b). Performance deterioration with the scan scheme is due to the high variability of residence time. Allocation of equal-sized jobs is not maintained in these situations. On the other hand, the lazy scheme performs well with all workloads, particularly with hyperexponential distribution.

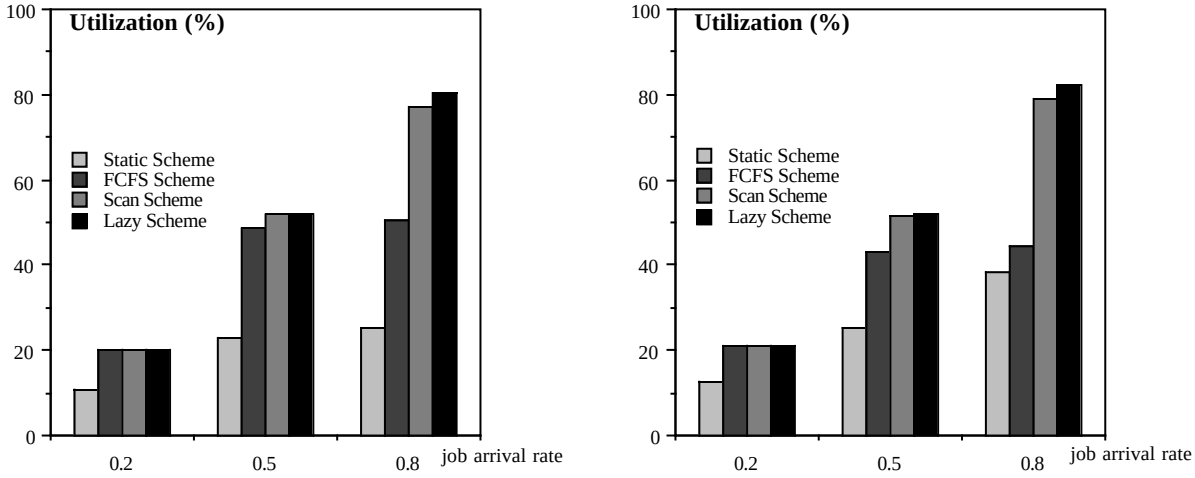


(a) Uniform job size/Hyperexp.residence time (dependent distribution, 10-cube system)

(b) Normal job size/Hyperexp.residence time (dependent distribution, 10-cube system)

Fig. 4. Average queueing delay variation for a 10-cube
Mean job size = 5-cube, Mean residence time = 5 time units

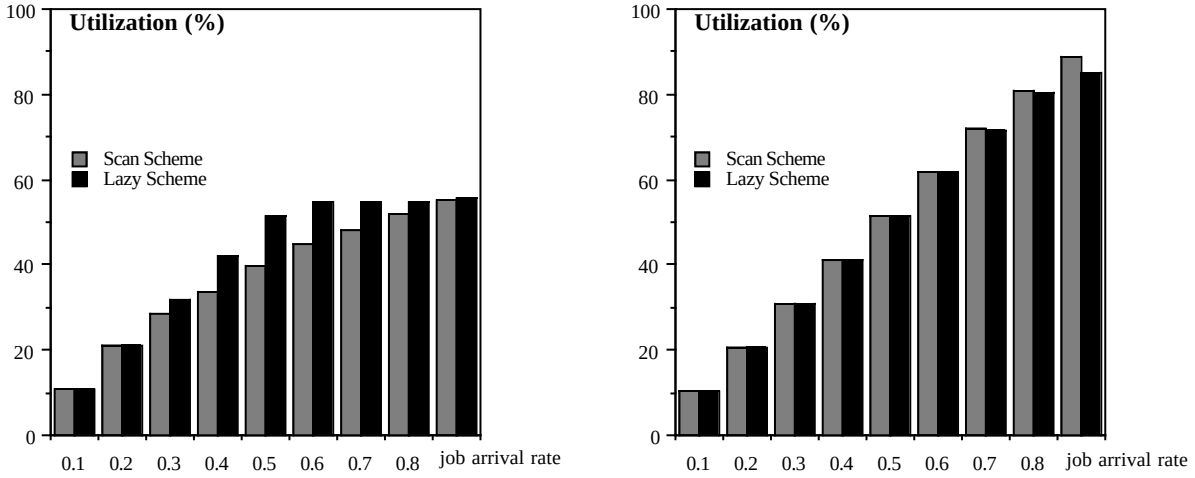
Figure 5 shows the comparison of system utilization at three different input loads. The job size is uniformly distributed in Figure 5(a), and is normally distributed in Figure 5(b). The job residence time is hyperexponentially distributed for both the plots. This graph primarily shows that the utilization of the static partitioning scheme is very low compared to the other schemes. Because of fixed partitioning, a large part of the system may be idle while there are a number of jobs in the queue for a different partition. This leads to poor system utilization. Thus, low queueing delay does not necessarily means better system utilization. The system utilization of buddy, scan, and lazy schemes are almost the same for light load. Lazy scheduling can provide better system utilization than others as the system load increases.



(a) Uniform job size/Hyperexp.residence time (dependent distribution, 10-cube system)

(b) Normal job size/Hyperexp.residence time (dependent distribution, 10-cube system)

Fig. 5. System utilization vs input load of a 10-cube
Mean job size = 5-cube, Mean residence time = 5 time units

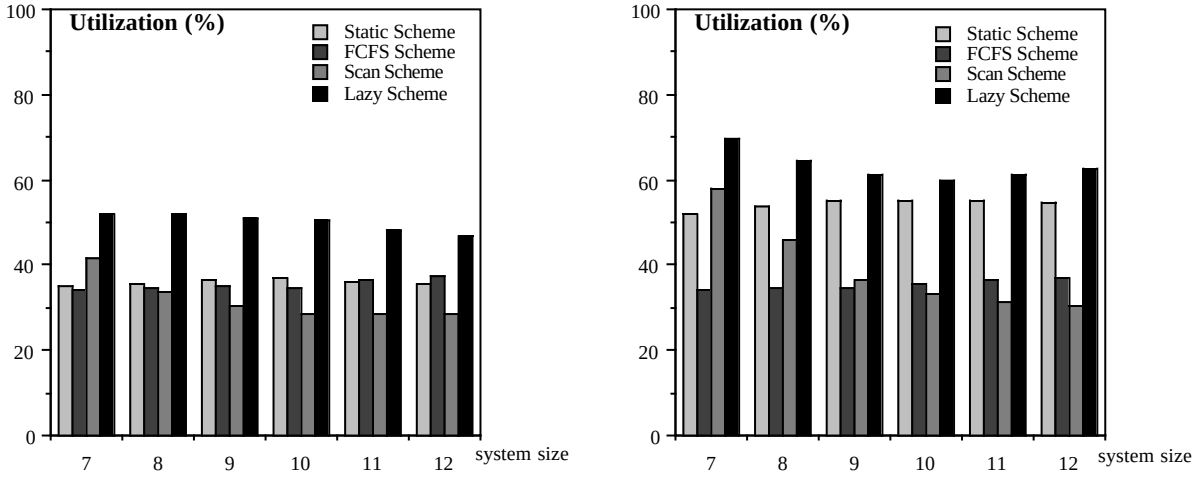


(a) Normal job size/Hyperexp.residence time (independent distr., 10-cube system)

(b) Normal job size/Uniform residence time (independent distr., 10-cube system)

Fig. 6. Utilization comparison of the two scheduling schemes
Mean job size = 5-cube, Mean residence time = 5 time units

Comparison of system utilizations with the lazy and the scan schemes as a function of input load is shown in Figure 6. Figure 6(a) shows the difference in utilization of the two scheduling policies when the job residence times are hyperexponentially distributed. System utilization with the lazy scheduling is relatively better than that with the scan scheduling. Lazy and scan schemes provide almost identical performance with uniformly distributed job residence time. This is demonstrated in Figure 6(b).

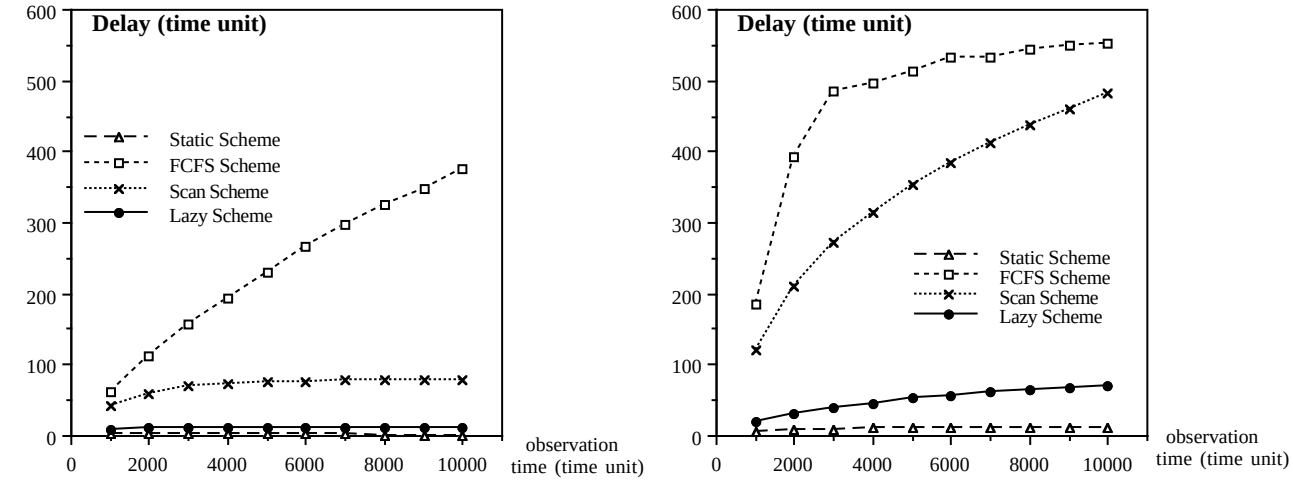


(a) Uniform job size/Hyperexp.residence time (independent distr., job arrival rate of 0.5)

(b) Uniform job size/Hyperexp.residence time (independent distr., job arrival rate of 0.85)

Fig. 7. Utilization versus system size for different schemes
Mean job size = (system size/2)-cube, Mean residence time = 5 time units

Figure 7 shows the utilization of static, FCFS, scan, and lazy schemes for different system sizes. The lazy scheme shows better utilization than all the other schemes irrespective of the system size. Figure 7(a) corresponds to light load whereas, Figure 7(b) represents heavy load condition.



(a) Uniform job size/Hyperexp.residence time (dependent distribution, 10-cube system, job arrival rate of 0.5)

(b) Uniform job size/Hyperexp.residence time (dependent distribution, 10-cube system, job arrival rate of 0.85)

Fig. 8. Delay variation with respect to the observation time
Mean job size = 5-cube, Mean residence time = 5 time units

Average delay with respect to the observation time is plotted in Figure 8. The job size is uniformly distributed and the residence times are hyperexponentially distributed. Figure

8(a) is for moderate load ($\lambda = 0.5$), and average queueing delay for heavy load ($\lambda = 0.85$) is shown in Figure 8(b). The graphs indicate that under moderate load, average queueing delay with the FCFS scheme is very high compared to the other schemes. Average queueing delay with the scan scheduling increases monotonically with time under heavy load, and the delay becomes closer to that of the FCFS scheme. Performance behavior with the static scheme is good because of the uniform job size distribution. Average queueing delay does not increases considerably with time in case of the lazy scheduling. It stays much lower than that of the FCFS and scan scheduling schemes.

We also measure the system throughput, which is the number of jobs completed per cycle. Throughput and average delay are not necessarily sufficient measures of system performance. It is observed that usually higher throughput results in longer delay. A combined metric called *system power* is sometimes more meaningful than either average delay or throughput alone [8]. System power is defined as the ratio of throughput to delay. A higher power means either a higher throughput or lower delay. In either case, a higher power is better than a lower power. Comparison of system powers of FCFS, scan, and lazy schemes is shown in Table I. System power may not be a monotonic function as it depends on both delay and throughput.

Table I. System power of a 10-cube system
Case A : Uniform job size/Uniform residence time (dependent distr.)
Case B : Normal job size/Hyperexponential residence time (dependent distr.)
Mean job size = 5-cube, Mean residence time = 5 time units

Input	Case A			Case B		
Load	FCFS	Scan	Lazy	FCFS	Scan	Lazy
0.1	20.00	0.04	0.50	1.333	0.400	0.250
0.2	4.000	0.16	0.57	0.200	0.235	0.381
0.3	1.500	0.31	0.54	0.031	0.075	0.444
0.4	0.800	0.38	0.50	0.003	0.027	0.444
0.5	0.470	0.34	0.45	0.003	0.017	0.365
0.6	0.162	0.26	0.37	0.003	0.013	0.277
0.7	0.013	0.17	0.27	0.004	0.010	0.175
0.8	0.003	0.09	0.17	0.004	0.007	0.087
0.9	0.002	0.04	0.08	0.005	0.005	0.031

It is observed that the FCFS scheme has maximum system power under uniform job size and residence time distribution. The lazy scheme shows better power than scan, and under high load it also performs better than the FCFS discipline. When the job residence time is hyperexponentially distributed, and the job size is normally distributed, the FCFS and the scan policies show better power under very light load. This is because the FCFS and the scan schemes have low queueing delay under very light load. In this environment, allocation of jobs are delayed even though the system has enough idle processors with the lazy scheme. So the jobs incur longer delay as compared to the scan or the FCFS schemes. But system power with the lazy scheme improves as the load increases. Furthermore, it should be noted that system power has wide variation from one workload to another for the FCFS and the scan policies. The variation is not that prominent for the lazy scheme. This also demonstrates the adaptiveness of the lazy scheme for varied workloads.

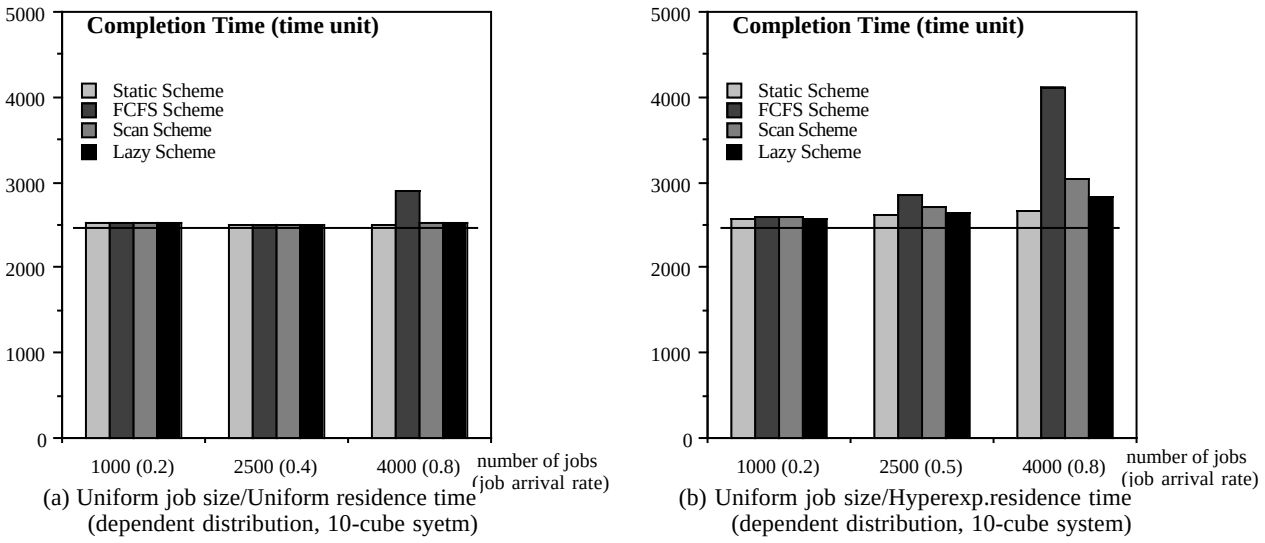
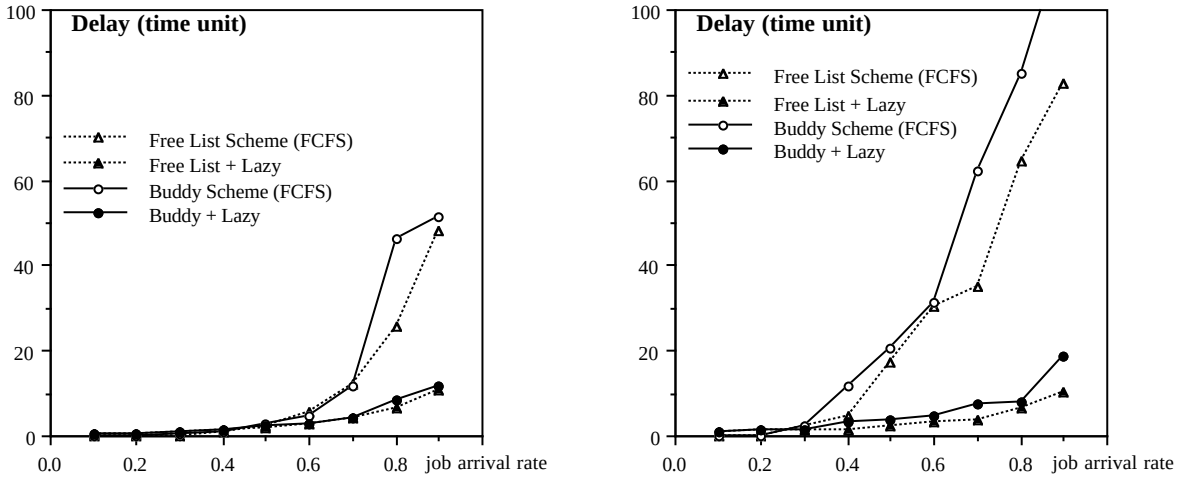


Fig. 9. Completion time using the scheduling schemes
Mean job size = 5-cube, Mean residence time = 5 time units

In order to integrate various performance measures, we generated a fixed number of jobs in a random order. The four scheduling schemes are used to process these jobs and the ordering was maintained for all the schemes. Three sets of jobs (1000, 2500, and 4000) are generated at various input rates (shown in Figure 9) with uniform job size distribution. The results are plotted for uniform residence time distribution in Figure 9(a), and for hyperexponential residence time distribution in Figure 9(b). The horizontal line indicates

the time at which the last job was generated. It is observed that when the job size and residence time are uniformly distributed, both the scan and the lazy scheduling schemes have more or less the same completion time. With hyperexponential distribution, the lazy scheme completes the jobs earlier than the FCFS or scan policies. The result with static scheme is optimal for uniform job size distribution. Job completion time with the FCFS scheme is maximum in both the graphs.



(a) Uniform job size/Uniform residence time (dependent distribution, 10-cube system)

(b) Normal job size/Hyperexp.residence time (dependent distribution,10-cube system)

Fig. 10. Queueing delay with combination of allocation and scheduling
Mean job size = 5-cube, Mean residence time = 5 time units

Finally, a performance study is conducted by combining the allocation and scheduling schemes. Until now, we have considered lazy scheduling with the buddy allocation scheme. Now we combine the lazy scheduling with the free list allocation scheme. Free list s selected as a representative of the policies having perfect subcube recognition ability. Average queueing delay with two different workloads are shown in Figures 10(a) and 10(b). The plots for the buddy and free list schemes with the FCFS scheduling are also shown for comparison. The lazy scheduling with free list allocation shows a little performance improvement than the lazy scheduling with buddy allocation. Performance improvement due to an efficient allocation policy is not prominent as seen from the graphs.

We have done extensive simulations for both dependent and independent workloads. The trends in various performance measures are observed to be similar for the two workloads.

6. Concluding Remarks

We have proposed a new scheduling scheme called *lazy scheduling* for assigning jobs in hypercube computers. This scheduling along with the buddy allocation scheme is used to process jobs in a multiuser environment. Prior research has focussed more on efficient allocation policies for hypercubes, although they provide only incremental performance gain due to the limitations with the FCFS scheduling. The lazy scheme is proposed as an alternative to the FCFS scheduling to improve hypercube performance. Time complexity of the lazy scheduling along with the buddy allocation is $O(n)$.

It is shown that significant improvement in system utilization, delay, and throughput can be achieved with the lazy scheduling compared to the FCFS discipline. Our scheme is compared with another technique, called scan for various workload distributions. It is observed that both scan and lazy schemes provide comparable performance under uniform workload distribution. However, for hyperexponential residence time distribution and varied job sizes, the lazy scheme out performs the scan technique. In summary, the proposed method is adaptable to all workloads where as scan is workload dependent.

The study argues in favor of exploiting various scheduling schemes as opposed to efficient but complex allocation policies. The concept may be applicable for other architectures. We are currently investigating the performance tradeoffs due to scheduling and allocation policies in other multiprocessors.

References

[1] Al-Bassam, S., El-Rewini, H., Bose, B., Lewis, T. G. Processor Allocation for Hypercubes. *J. Parallel Distrib. Comput.* Dec. 1992, pp. 394-401.

[2] Al-Dhelaan, A., and Bose, B. A New Strategy for Processor Allocation in an N-Cube Multiprocessor. *Int. Phoenix Conf. on Computers and Communications*, Mar. 1989, pp. 114-118.

[3] Bhuyan, L. N. and Agrawal, D. P. Generalized Hypercube and Hyperbus Structures for a Computer Network. *IEEE Trans. on Computers*, Apr. 1984, pp. 323-333.

[4] Chen, M. S., and Shin, K. G. Processor Allocation in an N-Cube Multiprocessor Using Gray Codes. *IEEE Trans. on Computers*, Dec. 1987, pp. 1396-1407.

[5] Chuang, P. J. and Tzeng, N. F. Dynamic Processor Allocation in Hypercube Computers. *Int. Symp. on Computer Architecture*, May 1990, pp. 40-49.

[6] Dutt S., and Hayes, J. P. Subcube Allocation in Hypercube Computers. *IEEE Trans. on Computers*, Mar. 1991, pp. 341-352.

[7] Hayes, J. P., Mudge, T. N., *et al.* Architecture of a Hypercube Supercomputer. *Int. Conf. on Parallel Processing*, Aug. 1986, pp. 653-660.

[8] Jain, R. *The Art of Computer System Performance Analysis,* John Wiley and Sons Inc., New York, 1991.

[9] Kim, J., Das, C. R. and Lin, W. A Top-Down Processor Allocation Scheme for Hypercube Computers. *IEEE Trans. on Parallel & Distributed Systems*, Jan. 1991, pp. 20-30.

[10] Knowlton, K. C. A Fast Storage Allocator. *Communications of ACM*, Vol. 8, Oct. 1965, pp. 623-625.

[11] Knuth, D. E. *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, Addison-Wesley, 1973.

[12] Krueger, P., Lai, T. H., Radiya, V. A. Job Scheduling is More Important than Processor Allocation for Hypercube Computers. *IEEE Trans. on Parallel and Distributed Systems*, May 1994, pp. 488-497.

[13] Livingston, M., and Stout, Q. F. Fault Tolerance of the Cyclic Buddy Subcube Location Schemes in Hypercubes. *Proc. Distributed Memory Computing Conference*, 1991, pp. 34-41.

[14] Saad, Y. and Schultz, M. H. Topological Properties of Hypercube. *IEEE Trans. on Computers*, vol. 37, July 1988, pp. 867-872.

[15] Trivedi, K. S. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications,* Prentice-Hall Inc., 1982.

[16] Wang H., and Yang, Q. Prime Cube Graph Approach for Processor Allocation in Hypercube Multiprocessors. *Int. Conf. on Parallel Processing*, Aug. 1991, pp. 25-32.