A Lazy Scheduling Scheme for Improving Hypercube Performance*

Prasant Mohapatra, Chansu Yu, Chita R. Das
Dept. of Electrical and Computer Engineering
The Pennsylvania State University
University Park, PA 16802

Jong Kim
Dept. of C. S. E.
POSTECH
P.O. Box 125, Pohang, Korea

Abstract

Processor allocation and job scheduling are complementary techniques to improve the performance of multiprocessors. It has been observed that all the hypercube allocation policies with the FCFS scheduling show little performance difference. 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. Simulation studies show that the hypercube performance is dramatically enhanced by using the lazy scheme as compared to the FCFS scheduling. Comparison with a recently proposed scheme called scan indicates that the lazy scheme performs better than scan 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 [1-3]. 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 it 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 [8]. Several heuristic algorithms reported in literature are buddy [4], modified buddy [5], gray code [6], free list [7], MSS [8], tree collapsing [9], and PC-graph [10]. These schemes differ from each other in terms of the subcube recognition ability and/or time complexity.

Comparison of all the hypercube allocation policies shows that the performance improvement due to better subcube recognition ability is not significant [7,11]. 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. Second, with the FCFS scheme, the scheduler tries to locate a subcube to allocate a job 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. There has been little attention paid towards

*This research was supported in part by the National Science Foundation under grant MIP-9104485.

scheduling of jobs in a distributed system like hypercube. The first effort in this direction was by Krueger et al [11]. They propose a scheme called *scan* which segregates the jobs and maintains a separate queue for each possible cube dimension. The queues are served similar to the c-scan used in disk scheduling. The authors show that 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 the performance gain diminishes.

In this paper, we propose a new strategy called *Lazy Scheduling* for scheduling 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 new subcubes and possibly fragmenting the system. The scheduling is not greedy and is thus named *lazy* . The system maintains separate queues 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. The proposed scheme tries to improve throughput by providing more servers to the queues that have more incoming 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 system load. The performance parameters analyzed here are average queueing delay and system utilization. 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.

The rest of the paper is organized as follows. In Section 2, various hypercube allocation and scheduling policies are summarized. The lazy allocation scheme is described in Section 3. The simulation environment and the workloads are given 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 is based on the buddy scheme for storage allocation [4]. 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 time complexities of allocation and deallocation 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 [12].

Modified Buddy

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 [5]. This scheme has better subcube recognition ability than buddy. The complexities of allocation and deallocation are $O(n2^n)$ and $O(2^k)$, respectively.

Gray Code

The gray code strategy proposed in [6] stores the allocation bits using a binary reflected gray code (BRGC). Here the least integer m is determined such that all the $(i \bmod 2^n)$ 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. 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 [7] 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 [8]. 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. The allocation and deallocation complexities are of the order of $O(2^{3^n})$ and $O(n2^n)$, respectively.

Tree Collapsing

Tree collapsing strategy is introduced in [9]. This involves successive collapsing of the binary tree representation of a hypercube structure. The scheme generates its search space dynamically and has complete subcube recognition ability. The complexities of allocation and deallocation 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 [10]. Subcubes are allocated efficiently by manipulating the PC-graph using linear graph algorithms. This scheme has also 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 [7,11] 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 [11]. Fragmentation of the system overides the advantages obtained from better subcube recognition ability. More sophisticated allocation policies, although show little performance improvement, introduce overheads and higher time complexity. It is for this reason we have adopted the buddy allocation scheme which is simple and has low time complexity.

2.3 Scheduling Strategies

A scheduling strategy called *scan* is proposed in [11]. The concept is similar to the disk c-scan policy. The algorithm maintains $(n+1)$ separate queues, one for each possible cube dimension. A new job joins the end of a queue corresponding to its subcube dimension. All jobs of a given dimension are allocated before the scheduler move on to the jobs of next dimension. It is shown that significant performance improvement can be achieved with this scheme compared to the most sophisticated allocation policies.

The 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 residence time of the jobs has little variation. In this environment, the system at any instant would have jobs of almost the same size and thus fragmentation is reduced. When the residence times of all the jobs are not the same, the allocations and deallocations eventually create a mixture of jobs from all the queues in the system and lead to fragmentation. The performance of the scan scheme is therefore dependent on the workload. The study in [11] is conducted under the assumption of exponential job residence time distribution with a mean of one time unit. Practically, the residence time distribution resembles close to hyperexponential distribution [13]. 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 short queue have to wait till all the jobs in the longer queues are processed. In such cases, a job arriving early to a short queue has to wait even longer than the jobs that arrive much later in the longer queues.

3 The 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. We conclude the section by analyzing the time complexity of the proposed scheme.

3.1 Static Partitioning Scheme

Static partitioning scheme divides the 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 requesting i-subcubes, for $0 \leq i \leq n-1$. Corresponding to each S_i, there is a queue Q_i. Thus there are n queues in an n-cube system. An incoming job 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 Scheme

Lazy scheduling is based on two key concepts. First, the jobs requiring small subcubes are not blocked behind the 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 decomposing 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 dimension wait for a single subcube executing in the system, then eventually the scheduling will resemble 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 explained later.

The lazy scheduling scheme is illustrated in Figure 1. There are $(n+1)$ queues for an n-cube system represented as Q_k's, for $0 \leq k \leq n$ ($n = 4$ in Figure 1). Each queue has a variable threshold length denoted as N_k, for $0 \leq k \leq n$. N_k is initially set to zero. N_k is incremented with a k-cube allocation and is decremented with a k-cube deallocation. In other words, N_k denotes the number of subcubes of a particular size being occupied at any instant of time. A new job is first enqueued according to its dimension. If the number of jobs waiting, $|Q_k|$, is more than N_k, then the scheduler tries to allocate the job at the head of the queue using an allocation algorithm (we have adopted the buddy scheme although any other scheme could be used). The job is allocated to the system if a suitable subcube is found. Each queue is scheduled on a FCFS basis.

Fig. 1. Queue management in a 4-cube

The determination of the threshold value, N_k, is based on the concept that for every subcube executing 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 executing 1-cube jobs. 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 policy of the scheduler is to create more servers for a queue if the number of requests for the corresponding cube is high. Although the scheduler 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. We modify our algorithm to eliminate the possibility of indefinite postponement by using a threshold value for maximum queueing delay that a job can tolerate. Whenever the waiting time of a job reaches this threshold value, it gets priority over all other jobs. No jobs are allocated until the job whose waiting time has reached the threshold value 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 a job 'sees' the arrival of $(d_t \cdot \lambda)$ jobs to the system, where λ denote 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 formal algorithm for the lazy scheduling is given below. Buddy_Request and Buddy_Release are procedures from the buddy allocation algorithm presented later. The flag *stop_alloc* is used to indicate 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.

We adopt the buddy strategy using an efficient data structure as proposed in [12]. 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 are empty except the list F_n which contains the n-cube. The allocation and deallocation complexities are of $O(n)$. The algorithm is presented below. I_k represents a job that requires a k-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. Put the released subcube into the list F_k.

2. If F_k contains the buddy of I_k, merge them and put in the list F_{k+1}.

3. Repeat step 2 until the corresponding buddy is not available.

3.3 Complexity Analysis

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 of $O(n)$. Determination of the oldest job is of $O(1)$ by keeping track of the waiting time of the earliest generated job. Deallocation with the buddy scheme takes $O(n)$ time. Thus the time complexity of the release process of lazy scheduling is of $O(n)$ 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. 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.

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 size and the 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. 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. 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.

The distribution of the job size is assumed to be uniform or normal. In a 10-cube system, for example, probability that a request is i-cube (p_i) is $\frac{1}{10}$ with uniform distribution. For normal distribution, 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 [13]. Mean residence time is assumed to be 5 time units, and the mean job size is assumed (system size/2). 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.

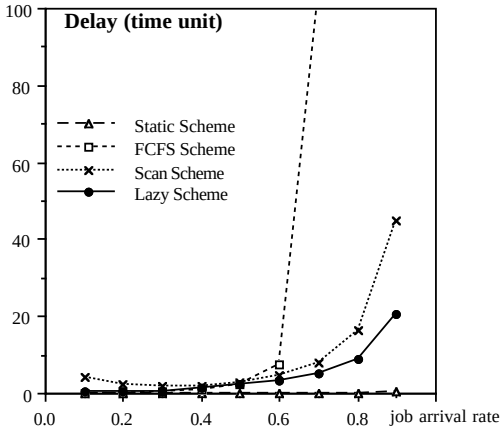
S : Sum of total service demand of generated jobs.

The performance parameters obtained from the simulator are utilization (U) and mean queueing delay (M). Mean queueing delay, M, is equal to R/A. The actual job request rate is measured by $S/(2^n \times T)$. The 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.

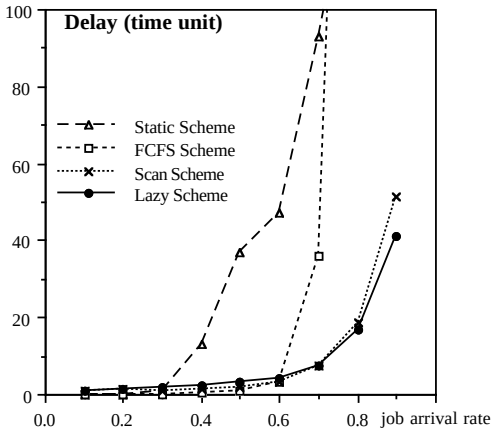
5 Results and Discussion

Figure 2 shows the variation of queueing delay with respect to input load for a 10-cube system. The job size is uniformly distributed in Figure 2(a), and Figure 2(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. The 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 2(b). Scan and lazy scheduling show better performance than the FCFS strategy for both uniform and normal distributions. Figures 2(a) and 2(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 the uniform job size distribution (Figure 2(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 becomes more prominent when the job residence time is hyperexponentially distributed. This is demonstrated in Figure 3. The job size is uniformly distributed in Figure 3(a), and is normally distributed in Figure 3(b). The 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 under all workloads, particularly with hyperexponential distribution.



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

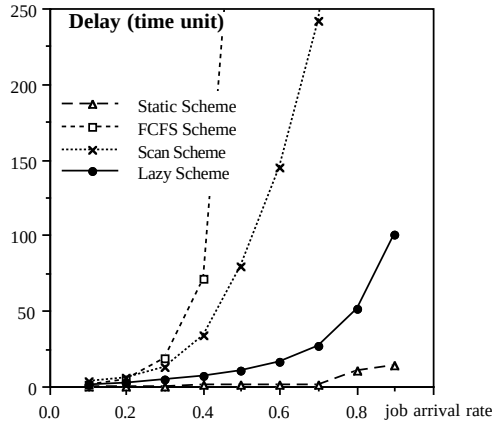


(b) Normal job size/Uniform residence time (dependent distribution, 10-cube system)

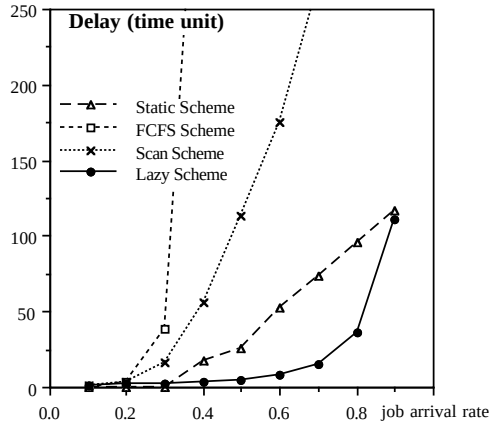
Fig. 2. Average queueing delay for a 10-cube

Figure 4 shows the comparison of system utilization at three different input loads. The job size is uniformly distributed in Figure 4(a) and normally distributed in Figure 4(b). The main observation from this graphs is 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 empty while there are a number of jobs in the queue for a different partition. This leads to the poor utilization in static scheme. Thus, lower queueing delay does not necessarily means better sys-

tem utilization. At low loads the system utilization of buddy, scan, and lazy schemes are almost the same. Lazy scheduling can provide better system utilization than others at higher loads.



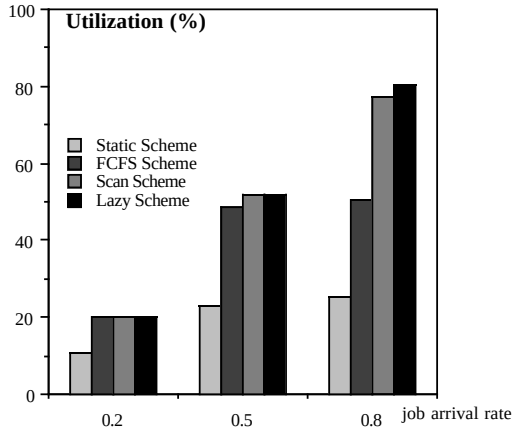
(a) Uniform job size/Hyperexp.residence time (dependent distribution, 10-cube system)



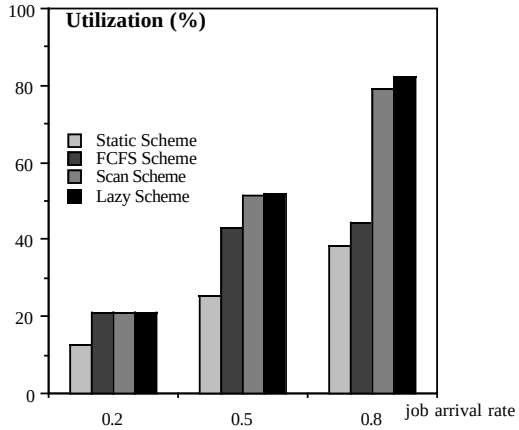
(b) Normal job size/Hyperexp.residence time (dependent distribution, 10-cube system)

Fig. 3. Average queueing delay for a 10-cube

The average delay with respect to the observation time is shown in Figure 5. Figure 5(a) is for moderate load ($\lambda = 0.5$), and average queueing delay for heavy load ($\lambda = 0.85$) is shown in Figure 5(b). The graphs indicate that under moderate load, the 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. The 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.

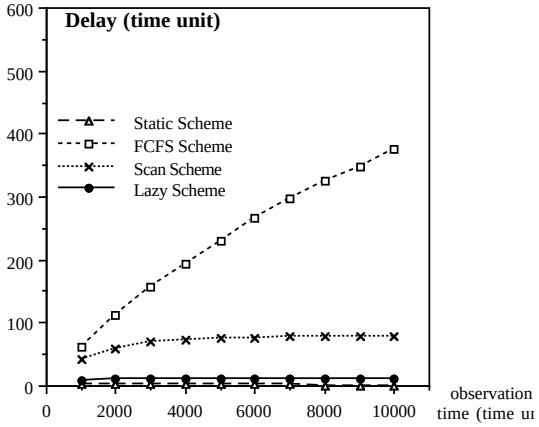


(a) Uniform job size/Hyperexp.residence time (dependent distribution, 10-cube system)

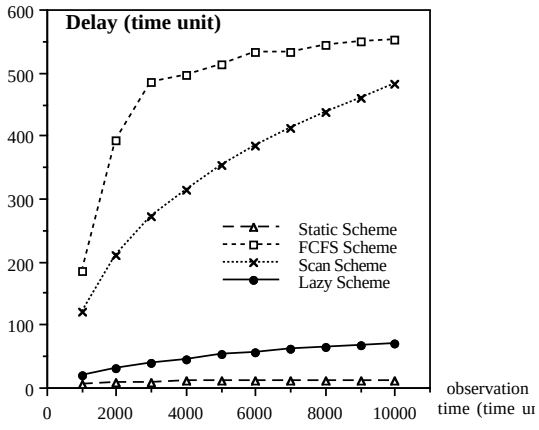


(b) Normal job size/Hyperexp.residence time (dependent distribution, 10-cube system)

Fig. 4. System utilization vs input load of a 10-cube



(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. 5. Delay variation with the observation time

We have done extensive simulations for both dependent and independent workloads. The trends in various performance measures are 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 the hypercube performance.

It is shown that significant improvement in system utilization and delay 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 method. 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. We are currently investigating the performance tradeoffs due to scheduling and allocation policies in other multiprocessors.

References

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

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

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

[4] K. C. Knowlton, "A Fast Storage Allocator," Communications of ACM, vol.8, pp. 623-625, Oct. 1965.

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

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

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

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

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

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

[11] P. Krueger, T. H. Lai, and V. A. Radiya, "Processor Allocation vs. Job Scheduling on Hypercube Computers," Int. Conf. on Distributed Computing Systems, pp. 394-401, 1991.

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

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