A genetic based disk scheduling method to decrease makespan and missed tasks

Mohammad Reza Bonyadi, Hossein Rahmani, Mohsen Ebrahimi Moghaddam *

Electrical and Computer Engineering Department, Shahid Beheshti University, G.C, Tehran, Iran

A R T I C L E   I N F O

Article history:
Received 25 September 2009
Received in revised form
20 April 2010
Accepted 21 April 2010

Keywords:
Disk scheduling
Makespan
Genetic algorithm
Missed tasks

A B S T R A C T

Disk scheduling is an operating system process to service disk requests. It has an important role in QOS guarantee of soft real-time environments such as video-on-demand and multimedia servers. Since now, some disk scheduling algorithms have been proposed to schedule disk requests in an optimized manner. Most of these methods try to minimize makespan by decreasing the number of disk head seeks as one of the slowest operations in modern computers and crucial for system performance because it usually takes some milli-seconds. In this paper, we propose a new disk scheduling method based on genetic algorithm that considers makespan and number of missed tasks simultaneously. In the proposed method, a new coding scheme is presented which employs simple GA procedures such as crossover and mutation and a penalty function in fitness. To get the best performance of the proposed method, its parameters such as number of chromosomes in initial population, mutation, and crossover probabilities, etc have been adjusted by applying it on some sample problems. The algorithm has been tested on several problems and its results were compared with well-known related methods. Experimental results showed that the proposed method worked very well and excelled most related works in terms of miss ratio and average seeks.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

The system performance of modern computers is affected by different factors such as processor speed, memory, disk capacity, and disc speed. Evolution of disc speed increases by the slow rate of only 7% per year as compared with 40% annual increase of processor speed and disk capacity [1]. In addition, many modern computer applications require huge amounts of data and in some applications; this data must be retrieved in real-time. Therefore, disk scheduling has great importance in such systems. Examples of such applications may be found in the multi-media field such as video delivery service known as video-on-demand and audio playback systems.

An operating system does not issue block I/O request to the disk in the order or as soon as they are received. Instead, it performs merging and sorting of disk operations to improve the system performance. The sub-system of kernel that performs these operations is called disk scheduler.

On the other hand, one of the requirements of real-time applications is quality of service (QOS) guarantee by operating system [2]. These applications are categorized based on the strictness of their QOS requirements as soft or hard real-time applications [3]. In the soft real-time applications such as video/audio playback, the most important QOS requirements are minimizing the number of missed deadlines [4,5] instead of maximizing the system throughput. Due to relatively slow speed of disks,
the role of an efficient disk scheduler algorithm is very crucial to deliver a smooth playback experience to users. Therefore, real-time disk scheduling algorithms are considered as soft real time and their goal is to find a feasible schedule with maximal throughput. Failure in servicing real-time disk requests may result in disk buffer overflow or playback jitters [6].

Most traditional disk scheduling algorithms, such as FCFS, SCAN, C-SCAN, LOOK, C-LOOK, and SSTF [7–13] are designed to reduce disk seek-time and increase its throughput. In the SCAN algorithm that is also known as "elevator", the arm moves in a predefined direction (e.g. left to right); after servicing all the requests in this direction, the opposite direction (i.e. right to left) is selected and algorithm repeats this procedure. The Circular SCAN (C-SCAN) [14,15] is a variation of the SCAN algorithm. It works in the same way as SCAN except that it always scans in one direction. After serving the last request in the scan direction, the disk arm returns to the start position without servicing any requests and repeats scanning. The advantage of C-SCAN algorithm is that it provides a more uniform wait time for I/O requests, as compared to SCAN [16]. LOOK algorithm is a variant of SCAN where the disk arm does not reach the inner or outer most tracks. The arm changes its direction as long as there are no more outstanding requests in the current direction. LOOK performs better than SCAN under low load condition and is nearly equivalent to SCAN when the load is high. A variant of LOOK scheduling that is designed to provide more uniform wait time is Circular LOOK (C-LOOK) scheduling [17]. C-LOOK moves the head towards one direction from its present position, after serving the last request in the current direction it reverses its direction and immediately returns to the first request of the other end without servicing any request on the return trip. The Shortest Seek Time First (SSTF) algorithm improves throughput by servicing a request that results in shortest seek distance. This algorithm provides high throughput but it may cause starvation problem. These algorithms do not consider real-time constraints of I/O tasks and, therefore, are not suitable to be applied directly on a real-time system [18,6]. On the other hand, some other disk scheduling algorithms such as earliest-deadline-first (EDF) [19] address this issue without considering disk seek-time. EDF Policy is optimal if the tasks are independent [20,21], therefore, it is not proper for real time disk tasks because in the disk-scheduling problem, the service time of a task depends on the track location of the previous task. An advantage of this algorithm is that it is simple and easy to implement. However, a weakness of EDF is that the disk access time on average is very high because the algorithm does not consider the arm position and perform no optimization in the seek time [22].

Group-Sweeping Scheduling (GSS) [23,24] is a disk scheduling strategy in which requests are served in cycles with a round-robin manner. To reduce disk arm movements, the set of streams is divided into some groups that are served in a fixed order. Streams within a group are served according to SCAN. If all requests are assigned to one group, GSS reduces to SCAN, and if all requests are assigned to separate groups, GSS effectively becomes round-robin scheduling. The service order within one group is not fixed, and a stream may be first one in the current cycle while it is last in the next cycle.

Earliest-Deadline-SCAN (D-SCAN) [25] is a modification of the traditional SCAN algorithm to consider request deadlines. In D-SCAN, the track location of the request with earliest deadline is used to determine the scan direction. In Feasible-Deadline-Scan (FD-SCAN) algorithm [25], the track of the request with the feasible deadline is used to determine the scan direction. A deadline is feasible if it is estimated that it does not miss. The disk arm moves toward the direction and services all requests along the way.

To improve both disk seek-time and miss ratio, SCAN-EDF [20,22] algorithm groups the disk requests that have the same deadlines, and then uses SCAN algorithm in each group. The advantage of SCAN-EDF is that it attempts to provide both seek optimization and earliest deadline first service. However, the effectiveness of the SCAN-EDF algorithm depends on how many requests carry the same deadline. If a server dynamically issues I/O requests, then the chance for more than two requests to have the same deadline is small or zero. In that case, the algorithm reduces to EDF. On the other hand, if a server uses service round and all requests during a round are assigned the same deadline, then the algorithm reduces to SCAN. Therefore, the behavior of this algorithm really depends on how deadlines are assigned to I/O requests. To overcome this problem, Deadline-Modification-SCAN (DM-SCAN) [26] suggests the use of Maximum-scannable-group (MSG). MSG is a set of consequent requests that can be successfully rescheduled by SCAN with guaranteed real-time requirements. In this algorithm, input disk requests must be sorted by their deadlines, and then request deadlines are reduced several times during the process of rescheduling to preserve EDF sequence. Unlike DM-SCAN, Reschedulable-group-SCAN (RG-SCAN) [27] does not require its input disk requests to be sorted by their deadlines. It also forms larger groups without any deadline modification.

In SCAN-EDF, DM-SCAN, and RG-SCAN algorithms, rescheduling is only possible within a local group of requests. Chang et al. in [21] suggests Global Seek-optimizing Real-time (GSR) disk scheduling algorithm that groups the EDF input tasks based on their scan direction. These tasks are moved to suitable groups to improve the system performance in terms of increased disk throughput and decreased number of missed deadlines. Output schedules of these algorithms are always feasible if the input real-time disk requests are feasible sequences. But with an infeasible input, it is very unlikely to have a feasible output. This is due to the fact that after each regrouping of input tasks, GSR checks the feasibility of the new schedule. If the new schedule is infeasible, GSR algorithm ignores the movement and selects another request to regroup and this continues until it reaches the last request.

The disk scheduling problem without the real-time constraints has been shown to be NP-complete [28] if the seek-cost function is not linear. The NP-complete proof comes from the general traveling salesman problem.
Also, the general real-time disk scheduling with linear seek-cost function is NP-complete [29,30]. Genetic algorithms propose a way to find a good but not the best solution of NP-complete problems. Therefore, Genetic algorithm may be employed in solving disk scheduling problem. However, there are some genetic disk scheduling methods in literature; for example, Turton and Arsalan [31] presented an algorithm that uses the concept of parallel genetic algorithms by employing order-based crossovers. Also, the effect of different order-based crossovers on the performance of an anticipatory disk scheduling algorithm that is tuned by a genetic algorithm has been studied in [32]. In another algorithm presented by Ökdem and Karaboğa [33], ant colony (ACO) approach has been used to schedule disk requests. This method works based on the idea in traveling salesman problem (TSP) and aims to reduce the response time. However in these heuristic methods no solution has been considered to reduce the number of missed requests.

One of the main concerns of heuristic algorithms is its running time, if these methods are used in a real time environment, high running time may cause some problems. To overcome this problem, these methods should be implemented by hardware. In this paper, we used a genetic based method to schedule disk requests. The reason for selecting genetic algorithm was that it is simple, it may be implemented with low time complexity and it is easy to be implemented by hardware with lower resources rather than other heuristic algorithms such as ACO (It is proved that the convergence time of hardware implemented GA has 99.6% reduction as compared with that of the implemented in C language software simulation [34,35]).

In the proposed method, a novel coding approach is presented and genetic operators have been adjusted to get best performance. Because of simple coding, the genetic operators are simple and fast. In the scheduling phase, to overcome the overhead of proposed method running time in real time tasks, the scheduling process is considered as an independent request in the system. Experimental results were satisfactory and they showed that the proposed method worked better than the related ones in terms of miss ratio and average seeks.

The rest of the paper is organized as follows: in Section 2 a brief explanation of the real-time disk scheduling problem is provided and in Section 3, the proposed approach is introduced. Section 4 describes the evaluation results and simulation way and in Section 5, paper is concluded.

2. Problem description

Each disk request $T_i$ in a real-time environment is defined by its ready time $r_i$, deadline time $d_i$, sector number $l_i$, data size $b_i$, and its corresponding track location $a_i$. Ready time is the earliest time at which a disk task can start. Deadline time is the latest time at which disk task should be completed. The actual starting and completing time of a disk task are called start time $s_i$ and fulfill (finish) time $f_i$, respectively. The start time and finish time of a real-time task $T_i$ with schedule sequence $T_j$ are computed by $s_i = \max \{r_i, f_j\}$ and $f_i = s_i + c_{j,i}$, respectively.

Assume that the schedule sequence consists of two sequential tasks $T_j$ and $T_i$. To serve the disk request $T_i$, the disk-head moves from previous task cylinder ($a_j$) to the requested one ($a_i$) by a seek-time cost. Then a rotational latency is used for the desired sector. Finally, the requested data ($b_i$) are transferred from disk to buffer in a transfer time. The service time of task $T_i$ is calculated as follows:

$$c_{j,i} = \text{seek}_\text{time}(a_i - a_j) + \text{rotational\_latency}(l_i) + \text{transfer\_time}(b_i)$$ (1)

One of the main problems in disk scheduling algorithms is to determine hard disk seek time. Extracting these parameters has been the subject of intense researches for more than a decade [36–40]. For example in a HP 97560 hard disk, the seek-time with movement distance $D_{j,i} = |a_i - a_j|$ is calculated as follows [36]:

$$\text{Seek\_time}(D_{j,i}) = \begin{cases} 3.24 + 0.4 \sqrt{D_{j,i}} & D_{j,i} \leq 383 \\ 8.00 + 0.008 \cdot D_{j,i} & D_{j,i} > 383 \end{cases}$$ (2)

**Definition 1.** A schedule $\tau$: $T_1 T_2 ... T_i ... T_n$ is called feasible if all real-time disk tasks $T_i$, for all ($i$: $1$ ... $n$), satisfy real-time requirements: $r_i \leq s_i$ and $f_i \leq d_i$.

Consider the schedule sequence $\tau': T_w(1) T_w(2) ... T_w(i) ... T_w(n)$; schedule fulfill time ($f_w(n)$) is the finish time of the latest task ($T_w(n)$) and $b_{w(i)}$ the data size of request $T_w(i)$. Therefore, the disk throughput is calculated as follows [27]:

$$\text{Disk Throughput} = \sum_{i=1}^{n} \frac{b_{w(i)}}{f_w(n)} \propto f_w(n)^{-1}$$ (3)

Therefore, the problem objective that is defined to maximize throughput can be achieved by minimizing the schedule fulfill time while the number of missed tasks is minimized. In overall, a goal of real-time disk scheduling problem is defined as follows:

Consider a set of $n$ real-time disk tasks $\tau$: $T_1 T_2 ... T_i ... T_n$. Finding a feasible schedule $\tau': T_w(1) T_w(2) ... T_w(i) ... T_w(n)$ with maximal throughput, is the goal of real-time disk schedulers. The index function $w(i)$, for $i$: $1$ to $n$, is a permutation of $\{1,2,...,n\}$.

3. Proposed approach

In this section, a method for disk scheduling based on GA is proposed. In the following sub-section, initialization procedure has been described. At first population initialization way is introduced. A new coding scheme is proposed and evaluated in Section 3.2. Afterward, a penalty function which is utilized in the fitness function is introduced. Also, a new parameter (called $P$) that has
been inspired from artificial immune systems (AIS) is presented in Section 3.4.

3.1. Population initialization

Population initialization is the first step of each algorithm that works based on GA. In the proposed method, the population is initialized via a random process. Algorithm 1 shows the initialization procedure.

Algorithm 1. Initialize population (Pop)

Input: Number of chromosomes (NOC), Number of tasks (Nt)
Output: Pop (Initialized population)
Consider a two dimensional array Pop that has NOC columns and Nt row
For i:0 to NOC-1
For j: 0 to Nt-1
Pop[i][j]=rand(Nt);
Return Pop;

This algorithm gets the number of chromosomes and number of tasks as its inputs and initializes the population randomly. Indeed, the population (Pop) is an array that its ith row shows the ith chromosome and the jth column of the ith row shows the jth genome in ith chromosome. Also, the rand(Nt) is a function that returns a random integer no in the interval [0, Nt-1]. The structure of chromosomes is presented in the next sub-section.

3.2. Coding scheme

The proposed coding scheme consists of a set of integer numbers in the interval [0, Nt-1] where Nt is the number of tasks, therefore, the length of each individual is equal to the number of tasks. As an example, Table 1 shows a disk scheduling problem which contains 5 tasks (the task with bigger value in priority has higher priority). Fig. 1 depicts a sample chromosome example, which is used to code a five task scheduling problem such as problem in Table 1. As shown in Fig. 2, the cell values are between 0 and 4.

In fact, each chromosome shows a sequence of numbers between 0 and Nt-1 and each genome points to the index of an array which includes unscheduled tasks. Algorithm 2 shows the pseudo code for the decoding procedure.

Algorithm 2. Decodes the input chromosome (C) and prepares the corresponding order of tasks (T)

Input: Chromosome C
Output: Order T
T: empty sequence ( < Null > )
Q:{0, 1, 2, 3, ..., Nt-1}
For i:0 to length(C)-1
    S=Mod(C[i], length(Q));
    T=T+Q[S];
    Eliminate Sth element of Q;
End for
Return T;

In this algorithm, C is an input chromosome which should be decoded to show tasks order. Also, Mod(a, b) is a function that returns the reminder of a when it is divided by b. As an example, Table 2 exhibits the results of applying Algorithm 2 on the chromosome in Fig. 1. As it was mentioned, five tasks are considered for this example. Note that the index for all arrays is considered from 0.

According to the table, the value of T after applying the Algorithm 2 on the chromosome in Fig. 1 is <3, 0, 2, 4, 1> which means the tasks should be executed in this order (left to right).

If we schedule the tasks in Table 1 using the order in Table 2 we have to start from T3 which needs one seek changes (assume that the start position of the hard head is in position 0) and 3 ms to perform its job. Assume that each seek needs 1 ms. Hence, it is completed in time 4 and it is not missed. After that, we have to schedule T0. Therefore, the position of hard head is 1 (last task) and we have to change it to 5 which needs 4 ms for seeking and 3 ms to transfer data. So it is completed in time 11. Table 3 shows all calculations for the chromosome in Fig. 1 according to Table 1.

As it seems from Table 3, if the tasks in the Table 1 are scheduled according to the chromosome in Figs. 1 and 2 tasks are missed and the procedure ends in the time 16. It is worth mentioning that if a task is missed, it is not considered in next calculations for current head position and current time.

![Fig. 1. An example of proposed chromosome structure which codes a 5 task problem such as problem in Table 1.](image-url)
3.3. Fitness function

In this sub-section the proposed fitness function is presented. At first, term "Makespan" is defined and then the fitness evaluation procedure is introduced.

Makespan is the time that the last task in the execution order completes its work. As an instance, the makespan for the order in Table 3 is 16 because the last task (\(T_4\)) completes its work in time 16 and the average seeks is \((1+4+2)/(3)=2.33\). Note that the value of makespan (16 in the mentioned example) is always smaller than or equal to worst deadline among all tasks (18 in the mentioned example).

In disk scheduling problem, the goal is to find an order for executing some tasks such that the makespan and the number of missed tasks to be minimized. In this way, a penalty function has been designed which guarantees the best makespan with regard to the number of missed tasks. In this function, with increasing the number of missed tasks the penalty value is increased such that the value of fitness becomes different for the chromosomes with the same number of missed tasks and different values of makespan. Eq. (4) shows the formula that can be used to evaluate the fitness of each chromosome.

\[
\text{Fitness}(C) = \text{makespan}(C) + \frac{\text{Miss}}{\text{WD}}
\]

where Miss is the number of misses that occurs in the sequence proposed by chromosome C and WD the maximum value for deadline among all tasks in the problem at hand. As this equation shows, the fitness function is minimized when the value of Miss is zero and the makespan has its minimum value. Indeed, the penalty value is big enough to distinguish a feasible order (an order with no miss) and an order with just one miss, because the value of makespan is always smaller than or equal to WD. This causes that a schedule with or without missed task is distinguished. By using this fitness function, when missed tasks are minimized, number of completed tasks is maximized. On the other hand, by minimizing the makespan, the throughput increases and the average seek decreases. Therefore, the proposed fitness function models increasing throughput, decreasing miss ratio, and decreasing makespan simultaneously.

### Table 2
Results of applying Algorithm 2 on the chromosome in Fig. 2.

<table>
<thead>
<tr>
<th>i</th>
<th>Q</th>
<th>C[i]</th>
<th>Length(Q)</th>
<th>S</th>
<th>Q[s]</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1,2,3,4}</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>(&lt;3&gt;)</td>
</tr>
<tr>
<td>1</td>
<td>{0,1,2,4}</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>(&lt;3,0&gt;)</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,4}</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>(&lt;3,0,2&gt;)</td>
</tr>
<tr>
<td>3</td>
<td>{1,4}</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>(&lt;3,0,2,4&gt;)</td>
</tr>
<tr>
<td>4</td>
<td>{1}</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(&lt;3,0,2,4,1&gt;)</td>
</tr>
</tbody>
</table>
However, in some application, the user may want to give a high priority to its request. In this situation, the scheduling method should employ a fitness function that considers the priority of each request. Therefore, to solve this problem, the following function is proposed for such cases:

\[
\text{Fitness}(C) = \text{makespan}(C) + (P_{i_1} \cdot \text{State}_{T_1} + P_{i_2} \cdot \text{State}_{T_2} + \ldots + P_{i_n} \cdot \text{State}_{T_{n-1}}) \cdot WD
\]

where \(P_{i_1}\) is the priority of task \(T_1\) in sequence \(C\). The value of this parameter may be set by user; if user specifies no value, its default value is 1 (that means all tasks have same priority).

\[
\text{State}_{T_i} = \begin{cases} 
1 & \text{if task } T_i \text{ misses its deadline} \\
0 & \text{otherwise}
\end{cases}
\]

When a user task (for example task \(T_j\)) is a high priority task, the user can set the parameter \(P_{i_j}\) with bigger value. As Eq. (5) shows, the fitness function is minimized when the value of \(\text{State}_{T_i}\) (for all \((i:0...Nt-1)\)) is zero and the makespan has its minimum value. It is worthwhile to note that, according to this fitness function, a chromosome with smaller number of missed task (even with big makespan) is preferred to a chromosome, which presents a sequence with bigger number of missed task (even with small makespan). For further visualization, the fitness value for the chromosome in Fig. 1 for the tasks in Table 1 is \(16+(0 \times 2+1 \times 1+1 \times 3+0 \times 1+0 \times 1)\times 18=88\). The proposed function in Eq. (5) has been used as the fitness function of the proposed method, however, if the priorities of all tasks are same, Eq. (5) reduces to Eq. (4).

### 3.4 Genetic Operators

Because of the proposed simple coding scheme, the GA operators (includes crossover and mutation) are very simple and fast. In the implementation, the single point crossover [41] with probability \(P_c\) has been used to combine chromosomes and prepare offspring. Furthermore, the uniform mutation method [41] with probability \(P_m\) has been used as mutation operator and roulette wheel algorithm has been considered as a selection procedure [41].

In addition, to impede the degeneration phenomena, the affinity mechanism is used in GA approach which has been inspired from the artificial immune systems (AIS) [42]. According to the affinity approach, in each generation we seek the current population to perceive similar chromosomes and replace them with new reinitialized chromosomes. This approach helps us to explore the information from a wide territory of problem space. The most significant point that should be considered here is that the above procedures are time consuming. Hence, we define a new parameter called \(P_i\) which controls this process. Indeed, we used this parameter as a probability that determines whether the affinity mechanism should be applied or not. Algorithm 3 shows this procedure.

### Algorithm 3. Apply affinity mechanism on chromosomes

**Inputs:** \(P_i\), Number of Chromosomes (NOC)

**Output:** there is no output for this algorithm. In fact, the changes are applied on the global variable Pop(population)

If \((\text{rand} < P_i)\)

**Begin**

For \(i:0\) to NOC-1

For \(j:i+1\) to NOC-1

**Begin**

If \((\text{the output of Algorithm 2 is equal for both Pop}[i]\) and Pop[j]) then

Reinitialize Pop[j] randomly;

End for

End if

**End**

**End if**

In this algorithm, all chromosomes in the population (Pop) are compared to each other and if the tasks order (decoding result according to Algorithm 2) of two chromosomes were equal, one of them is reinitialized. Also, rand is a function that returns a random value in the interval \([0, 1]\). If this value is less than \(P_i\) then the affinity mechanism is applied on the current population. This parameter has been set via a case study (see Section 4.2).

### 4. Experimental results

In this section, first the used procedure to set the parameters of the algorithm \((P_m, P_c, P_i)\) by changing their values in feasible intervals is described. In the second subsection, experimental results are presented that consists of the comparison results among the proposed method and some other algorithms for disk scheduling. In the third subsection, the effect of task priorities on the results of the proposed method have been described briefly and finally a discussion about running time of proposed method has been presented.
4.1. Parameter setting

The parameters $P_m$, $P_c$, and number of chromosomes have been adjusted by considering several tests. In these tests, some problems with different number of real-time tasks (10, 15, 20, and 30) have been used as typical problems to check the performance of the proposed algorithm when just one parameter has been changed. The requested data of the tasks in all problems was equal to 36 KB and their period times have been considered as random number with uniform distribution. Also, in each test, the maximum generation has been considered equal to 100.

At first, the number of initial chromosomes has been changed from 4 to 50 with the step size 2 and the algorithm has been applied to the mentioned problems. The values of $P_m$, $P_c$, and $P_i$ have been considered as 0.2, 0.8, and 0.2, respectively, for this test and the results have been shown in Fig. 2. Each graph in Fig. 2 shows the average of fitness values over 1000 runs when number of chromosomes is varied. Fig. 2 shows that the number of chromosomes does not have high effects on the fitness value when it is more than the number of tasks. Hence, in all implementations we set the number of chromosomes equal to the number of tasks.

To find the best value for $P_m$, $P_c$, and $P_i$, they were changed in the interval $[0.01, 0.91]$ with step size 0.1, $[0.5, 0.95]$ with step size 0.05, and $[0.01, 0.91]$ with step size 0.1, respectively. In each run, the algorithm was applied 100 times on six problems that each one had 30 tasks, and the average of fitness values was reported.

Fig. 3 exhibits some results of this experiment versus four different values of $P_i$ (the graphs correspond with other values of $P_i$ have not been reported here) and Fig. 4 shows the obtained average and standard deviation of fitness values versus different values of $P_i$. It is implied

![Fig. 3](image)

Fig. 3. The values for $P_m$, $P_c$, and $P_i$ were changed and the average of fitness values for six problems with 30 tasks were reported. The algorithm was applied on each problem 100 times and the averages have been shown. To draw each graph $P_i$ had a fix value and $P_m$, $P_c$ were changed. (a) $P_i=0.01$, average of fitness values is 645, (b) $P_i=0.21$, average of fitness values is 619, (c) $P_i=0.61$, average of fitness values is 623, and (d) $P_i=0.91$, average of fitness values is 645.
from Fig. 4 that the different values of $P_i$ have different effects on the performance of the algorithm in both the average of fitness and the variance of found solutions. All in all, with regard to these results, the best performance for the proposed GA appeared when the values for $P_m$, $P_c$, and $P_i$ were 0.51, 0.95, and 0.21, respectively. Hence, in all further tests these values were used.

4.2. Simulation environment

In the simulation environment, set of requests have been considered that their ready times were Poisson process with an average inter-arrival time rate ($\lambda$). In simulations, we made various load traffic sequences. The deadlines of requests are calculated as $TimeMultiplier \times (Timeout \ Base) \times (Timeout \ \times \ (Size/36 \ KB))$ where $TimeMultiplier$ is a uniform random number in interval [1...5] and the value of TimeoutBase is 550 for each task. Also, the value of Timeout is 10 and the size parameter is the size of requested data by each request. The experiment was performed 100 times with different random generator seed at each run. In each of these runs, 400,000 real-time disk requests have been given to the algorithms. Also, the proposed function in Eq. (5) has been considered as a fitness function and to make the results comparable with other methods, the priority of all tasks considered same.

All implementations were performed on a personal computer with 2.26 GHZ of CPU and 3 GB of RAM in the C++ environment. In Table 4, the main parameters of the disk model ($HP$ 97560) have been provided. It is worth mentioning that changing disk type changes the absolute value of results but it has not effect on the relative values, therefore, an arbitrary disk type that is considered in literature has been used to demonstrate the value of results.

The experimental results showed a significant improvement in disk scheduling results in comparison with FIFO, GSR, C-SCAN, EDF, SCAN-EDF, and Ant Colony (ACO) based algorithms.

4.3. Results and comparison with related works

In this sub-section, experimental results are reported and they are compared with related works. To do comparison precisely, we implemented and applied several methods on various load traffic sequences and with different size of requests.

The results have been compared with some well-known methods such as C-SCAN, FIFO, EDF, SCAN-EDF, GSR, and Ant Colony (ACO) Based disk scheduling algorithm [43] in terms of their number of missed tasks and average seeks. It is worth mentioning that the default GA parameters are shown in Table 5 (see Section 4.1).

Figs. 5 and 6 show the miss ratio and the average seeks of requests in FIFO, EDF, C-SCAN, SCAN-EDF, GSR, ACO based and proposed GA algorithm versus a set of requests with size 36, 180, 360 KB, and a random mixture of these sizes, respectively. In this case, $\lambda$ parameter was considered as 0.125 which means, in average, each 8 ms one request arrived to the disk.

It is obvious in the figures that the proposed method has better results in terms of the miss ratio and average seeks (except C-SCAN in term of average seeks) in comparison with other methods in the all cases. The proposed algorithm has improved the miss ratio by about 6.5%, 7%, 7%, and 6% in comparison with GSR when the size of requests in each set is 36, 180, 360 KB, and a mixture of these sizes, respectively. Also, proposed method worked better than GSR, SCAN-EDF, EDF, and ACO based method in terms of miss ratio and average seeks in the all cases.

Figs. 7 and 8 show the miss ratio and the average seeks of requests in some scheduling algorithms when $\lambda$ parameter has been set to 0.09, respectively. From these figures, we can see that, the proposed method improved EDF and SCAN-EDF by about 256 and 188 in average seeks.

![Fig. 4. The average of fitness value and its variance over 100 runs for each value of $P_i$.](image-url)
when the size of each request is 36 KB, respectively. Also, GSR algorithm outperformed miss ratio versus EDF and FIFO by about 2%, and 8%, respectively, while the proposed has 6% improvement in the miss ratio versus GSR. Also, the average seeks of proposed method decreased by about 97 in comparison with GSR when the size of

Fig. 5. Comparing the average miss ratio of the proposed method with other methods versus different block sizes when $\lambda=0.125$.

Fig. 6. Comparing the average seeks of the proposed method with other methods versus different block sizes when $\lambda=0.125$.

Fig. 7. Comparing the average miss ratio of the proposed method with other methods when $\lambda=0.09$. 

each request was 180 KB. In overall, the proposed method outperformed FIFO, EDF, SCAN-EDF, ACO and GSR in terms of miss ratio and average seeks in all the cases.

Figs. 9 and 10 show the miss ratio and the average seeks of requests in FIFO, EDF, C-SCAN, SCAN-EDF, ACO based, and proposed GA algorithms, respectively. Also, the \( \lambda \) parameter has been considered as 0.06 in these figures.
It is obvious in these figures that GA based algorithm has better results in terms of miss ratio and average seeks in comparison with other methods (except C-SCAN in terms of average seeks) in all cases.

Fig. 9 shows that GA based algorithm has improved the miss ratio by about 0.5%, 5.5% in comparison with ACO and SCAN-EDF, respectively, when the size of each request is different. Also Fig. 10 shows that the proposed algorithm worked better than EDF, SCAN-EDF, ACO, and GSR in average seeks.

From Figs. 5, 7, and 9, we can conclude that by decreasing the $\lambda$, miss ratio has decreased that was expectable, because by decreasing $\lambda$, the arrival time difference between two successive tasks increases and as a result queue length decreases.

4.4. Impact of tasks priorities on the proposed method results

In this subsection, the impact of tasks priorities on the proposed method results has been discussed. The simulation environment was same as last, only a uniform random integer number in interval [1…5] was assigned to each task as its priority and the requests size in the test set considered as a mixture of different values (36, 180, and, 360 KB).

Figs. 11 and 12 show the miss ratio of using two proposed fitness functions when the $\lambda$ parameter has been considered as 0.125 and 0.06, respectively. For example, as shown in Fig. 11, when Eq. (4) is used to calculate the fitness function of the proposed method, 1.08% of high priority tasks miss their deadlines and when Eq. 5 is used, only 0.04% of these tasks miss their deadlines. As another example, in Fig. 12, when Eq. (4) is used to calculate the fitness function of the proposed method, 0.32% of tasks with priority four miss their deadlines and when Eq. 5 is used, no tasks miss its deadline. As a result, the fitness function based on Eq. 5 is better than Eq. (4) in priority based applications.

4.5. Running time of proposed method

The proposed method is not a greedy algorithm; therefore, one of its main concerns is the running time; because it is designed to be used in a real time environment, high running time may cause some
problems in scheduling of other tasks. Fig. 13 shows the average running time of three disk scheduling algorithms in simulation environment: two non-greedy ones: proposed and an ACO based method and one greedy algorithm: SCAN-EDF. As it is shown in this figure, the proposed method needs less running time than ACO based method, but it usually needs more running time than greedy ones. Because the scheduling process is considered as a separate task when it is applying on tasks queue, if its running time is comparable with needed time to handle a typical task, it is usable. The needed time to handle a typical task usually is considered as average seeks time of disks. Since the average seeks time of new hard disks is about 8 ms [44,45], therefore, the proposed method is proper to be applied when number of tasks is about 25 that is a fair number (it is obvious that if a faster computer is used this number increases a little). If we are going to use the proposed method for bigger number of simultaneous tasks, the method should be implemented by hardware to decrease the running time [34,35,46]. It is worth mentioning that the convergence time of pipeline based hardware implementation of GA has 99.6% reduction as compared with that of the implemented in C-language software simulation [34].

5. Conclusion

In this paper, a new approach based on Genetic Algorithm (GA) was proposed to schedule disk requests. In the proposed method, a simple and robust coding technique has been used while simple genetic operators have been considered. To evaluate the proposed method, some scheduling problems have been employed that their parameters were generated randomly. Some of these problems have been used to adjust operator parameters. Also, some famous related works were implemented such that we could compare the proposed method results with them. The simulation results showed that the proposed method has less number of missed tasks versus other related works; however, in some cases it has lower average seeks than C-SCAN algorithm.

To make it possible to use proposed method in every scheduling system, it is better that this method is implemented by hardware in future.


