Real-Time Scheduling of Multimedia Data Retrieval to Minimize Buffer Requirement

*Wen-Jiin Tsai** and *Suh-Yin Lee*

Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, 300, Taiwan, R.O.C.
E-mail: wjtsai@info1.csie.nctu.edu.tw

ABSTRACT

Continuous display is an important issue in the domain of multimedia applications. Especially, to ensure this continuity in the presence of multiusers, a feasible scheduling algorithm is prerequisite for real time data retrieval from the I/O subsystem. I/O scheduling techniques can be classified into two types: *meta-I/O scheduling* which arranges the sequence of data retrieval before issuing physical I/O requests, and *disk scheduling* which determines the order of processing I/O requests that have been issued. In *disk scheduling*, there are several elegant algorithms that had been discussed such as Scan, C-Scan, shortest seek time first and Scan-EDF. All of them focused on improving I/O throughput by serving requests closer to disk head first [7][8]. We focus this paper, however, on solving the real time meta-I/O scheduling.

For real-time scheduling, several algorithms had been addressed such as earliest-deadline-first (EDF) [4], least-laxity-first (LLF) [9], earliest-ready-time first (LRF) [6], and so on, which had shown to be elegant for task scheduling to promote system throughput. When applying to meta-I/O scheduling, however, these algorithms would result in large amount of buffer requirement for accommodating the retrieved data. In this paper, we proposed two real-time algorithms and a technique, called *object migration*, to minimize buffer requirement for meta-I/O scheduling. A buffer measurement approach was also proposed in this paper to estimate the performance of a real-time scheduling algorithm, which is based upon the well-known graph coloring technique. Simulation experiments were conducted to analyze the performance of algorithms. The results indicate that our approaches perform much better than existing real-time algorithms in terms of reducing buffer requirement.

Keywords: *Real time*, *Scheduling*, *Data retrieval*, *Buffer management*.

1. Introduction

Advances in information technology have made it feasible to develop multimedia computing systems capable of offering various services. In the domain of multimedia applications, continuous display is an important issue. In designing a multimedia server, to ensure continuous display for multiusers, a feasible scheduling mechanism is prerequisite to retrieve data in real time from storage subsystem. I/O scheduling techniques can be classified into two types: *meta-I/O*

scheduling which arranges the sequence of data retrieval before issuing physical I/O requests, and *disk scheduling* which determines the order of processing I/O requests that have been issued.

For *disk scheduling*, there are several elegant algorithms that had been discussed. Traditional disk scheduling algorithms such as Scan, C-Scan and shortest seek time first (SSTF) focused on improving I/O throughput by serving requests closer to disk head [8]. Current studies such as Scan-EDF [7] tends to take advantages of real-time technique in disk scheduling to meet the real-time requirement. For *meta-I/O scheduling*, several real-time algorithms have been addressed, such as earliest-deadline-first (EDF) [4], least-laxity-first (LLF) [9], earliest-ready-time first (ERF) [6] and so on, which had shown to be elegant for task scheduling to promote system throughput [5]. When applying to meta-I/O scheduling, however, these algorithms would result in large amount of buffer requirement for accommodating the retrieved data. In this paper, two real-time algorithms and a technique, called *object migration*, were proposed for real-time meta-I/O scheduling to reduce buffer requirement.

The rest of the paper is organized as follows. In Section 2, we first address the problems of buffer requirement occurred in traditional real-time algorithms. A buffer measurement approach was then proposed in Section 3 for estimating the performance of scheduling algorithms, which is based upon the well-known graph coloring technique. Section 4 describes the approaches for reducing buffer requirement in meta-I/O scheduling. In this section, we proposed two real-time algorithms and a strategy called *migration* to achieve our goal. Details of the performance tests are presented in Section 5. The simulation results are also discussed in this section. Section 6 concludes the paper with a summary of this work.

2. Background and Problem

In real-time scheduling, a request R consists of a triple $[r, d, c]$, where r is the ready time before which it cannot be start, and d the deadline by which it must be completed, and the c is the processing time of this request. The time interval $[r, d]$ is termed the *time window* of the request. The scheduling algorithm is to find the *start time* s and the *finish time* f of each request such that it can run without interference within time interval $[s, f]$ where $f = s + c$. A schedule is called *feasible* if all requests are scheduled within their time windows, i.e., $r \leq s$ and $f \leq d$. Several studies had shown the elegance of real-time algorithms, such as EDF and LLF, for easily finding a feasible schedule. When applying to meta-I/O scheduling, however, these algorithms suffer from needing a large amount of buffer space for holding the prefetched data. As an example consider four requests given below:

$$\begin{aligned} R_1 &: [r_a, d_a, c_a] = [0, 7, 4] \\ R_2 &: [r_b, d_b, c_b] = [0, 9, 1] \\ R_3 &: [r_c, d_c, c_c] = [0, 10, 3] \\ R_4 &: [r_d, d_d, c_d] = [0, 14, 2] \end{aligned}$$

Figure 1 exhibits the schedule resulted from employing EDF algorithm, where the retrievals of object **a**, **b**, **c** and **d** are scheduled within time interval [0, 4], [4, 5], [5, 8] and [8, 10], respectively. All the objects in Figure 1 are *prefetched* because all the finish times are scheduled before their

deadlines. Data prefetching may result in a large amount of buffer requirement in the system. Note that, in the area of multimedia presentation, it would be meaningless if related data cannot be synchronous to display, e.g., video and audio. For multimedia data scheduling, we define the deadline of an object as its *synchronization point* meaning that this object cannot be displayed before or after the deadline for the purpose of synchronization. That is, the retrieval of an object must be completed before its deadline and the buffer that allocated for holding this object must be preserved until its deadline. We refer to the buffer space reserved for an object from its start time to its deadline as the *extra-buffer*. The result in Figure 1 indicates that, by using EDF algorithm, we need a large amount of extra-buffer to accommodate all the prefetched data. As an example in Figure 1, we need an extra-buffer of four units from time 0 to 7 for accommodating object **a** and another of one unit from time 4 to 9 for object **b**, where we assume that the sizes of object **a** and **b** are four and one units, respectively. Without considering the prefetched object **c** and **d**, the extra-buffer of 5 units is required for holding object **a** and **b** in such a schedule. Note that, if object **b** is scheduled within time interval [8, 9] then only an extra-buffer of four units is needed. The reason is that the extra-buffer which object **a** resides has been released at time 7 such that it can be reused for holding object **b** from time 8. However, this may not be good if object **c** and **d** are also taken into consideration. Most of the current real-time algorithms, such as EDF, are designed for easily finding a feasible schedule (schedule all the requests within their time windows). In this paper, we focus on designing algorithms both to find feasible schedules and to minimize the extra-buffer requirement.

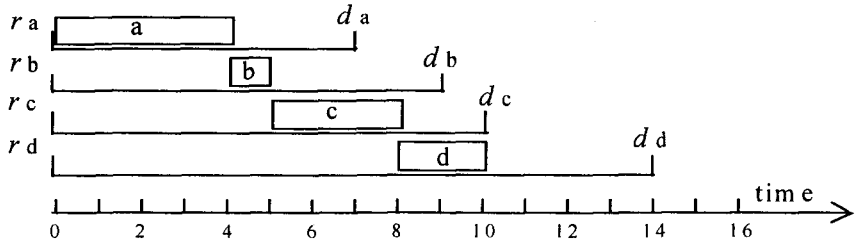


Figure 1: Schedule by using EDF algorithm

In the following, we explored two real-time algorithms and a technique called *migration* to achieve this goal. Before that, an approach is first proposed in the next section to measure the extra-buffer requirement for a schedule, which is based upon the well-known graph-coloring technique.

2. The Buffer Measurement Approach

Let *Pi* denote the time interval between the start time and the deadline of an object i, that is, $Pi = [s_i, d_i]$, meaning that within which extra-buffer is required for holding object i. We refer to *Pi* as the *prefetching interval* of object *i*. Figure 2 shows all the prefetching intervals of the objects in Figure 1. Note that, two objects with overlapped *prefetching intervals* need to be located to different extra-buffer space such that their contents will not be corrupted by each other. As an example in Figure 2, we must allocate distinct extra-buffer space for object **a** and **b** because the

prefetching intervals of **a** and **b** are overlapped. However, the extra-buffer reserved for object a and d need not to be different because object **d** can use the extra-buffer released from object **a**.

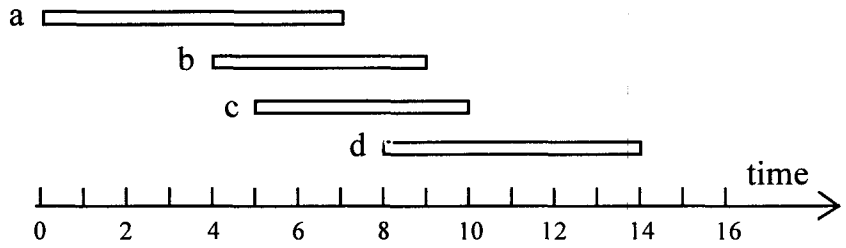


Figure 2: Prefetching Interval of schedule in Fig.1

We formulate such an extra-buffer allocation problem as a graph coloring problem: each node in the graph stands for an object, and two nodes are connected by an arc if two objects have overlapped *prefetching interval*. Such a graph is called an *interference graph*. Figure 3.(a) delineates the interference graph of Figure 2. The problem then is to color the graph using a number of colors less than or equal to the number of available extra-buffer units in the system. In graph coloring algorithm, no adjacent nodes may have the same color, meaning that no two objects with overlapped prefetching intervals may be assigned to the same extra-buffer (e.g., node **a** and **b** in Figure3.(a)). However, nodes that are not connected by an arc may have the same color, allowing objects whose prefetching interval do not overlap occupy the same extra-buffer space (e.g., node **a** and **d**). To better fit the problem, we further modify the graph coloring technique by allowing a node to be assigned more than one color. This enables those objects with larger size to be hold in more buffer units. The number of colors assigned to a node is based on the *degree* of the object (say S) associated with that node, which is defined by:

$$L = \text{size of object S} / \text{size of } \textit{extra-buffer unit}$$

where *extra-buffer unit* is defined by the *greatest common divider* (gcd) of all the object sizes. This means that L buffer units should be allocated to accommodate object S. Assuming that the degrees of object **a**, **b**, **c** and **d** are 4, 1, 3 and 2, respectively, a possible colored interference graph of Figure 3.(a) is depicted in Figure 3.(b), where nodes associated with object **a** is assigned four colors and nodes associated with object **b**, **c** and **d** are assigned one, three and two colors, respectively. It can be seen that the four nodes in Figure 3.(b) are colored with eight colors in total, meaning that at least eight extra-buffer units are required to accommodate all the objects.

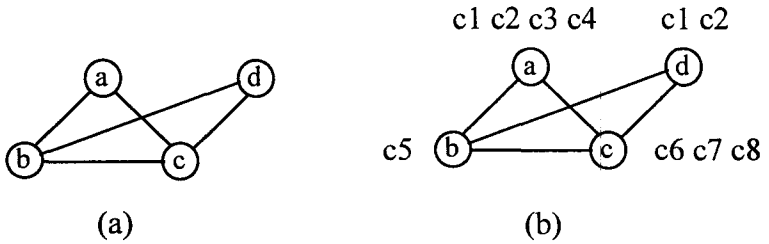


Figure 3: Interference Graph of Fig. 2

Note that the number of available colors is equal to the number of extra-buffer units for accommodating objects. As long as the available colors are sufficient to color the entire interference graph, the system is capable of accommodating all the retrieved objects until their deadlines. Although Graph Coloring is an NP-complete problem, there are heuristic algorithms that work well in practice. In the domain of compiler, for example, graph coloring technique has been widely employed in solving the problem of register allocation, which appeared wonderfully elegant and simple [1][2][3].

3. The Real-Time Scheduling Algorithms

In buffer measurement approach, it can be seen that extra-buffer requirement for an object S is dominated by two factors: the prefetching interval and the size of object S. This implies that extra-buffer requirement for all the objects can be reduced by taking these two factors into consideration. In the following, two real-time algorithms were exploited.

3.1 Latest-Deadline-Last Algorithm

The idea behind the Latest-Deadline-Last (LDL) is to schedule each task as much as close to its deadline such that the prefetching interval for each object is shorten and total amount of extra-buffer requirement can be reduced. With LDL algorithm, tasks are scheduled from the latest deadline backwards to the current time. While scheduling tasks, we keep a time called *current deadline* so that a task can be scheduled only before this time. A set of tasks called *active task set* is used to keep all the tasks whose deadline are greater than or equal to the current deadline. In other words, all the tasks in the active task set are the only one considered for scheduling among all unscheduled tasks. We select a task with the latest deadline to schedule its finish time at current deadline. The current deadline is then updated to the start time of the newly scheduled task. Whenever the current deadline is changed, the active task set is also updated by including more unscheduled tasks whose deadlines are behind the new current deadline. If there is no task in the active task set, the current deadline is shifted to the latest deadline of the unscheduled tasks.

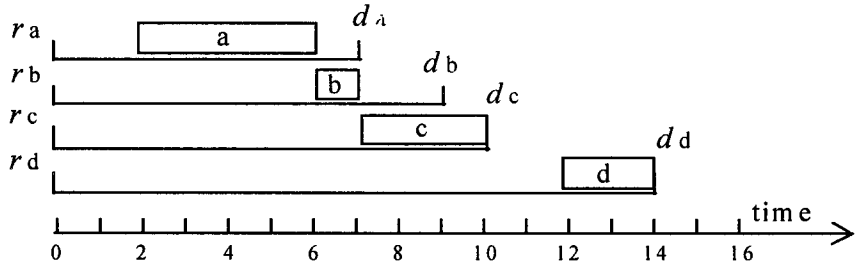


Figure 4: Schedule by using LDL algorithm

Figure 4 shows the schedule by using the LDL algorithm, where only object **a** and **b** are need to be prefetched. By using graph coloring technique, five buffer units are required to accommodate all the objects, where the degrees of objects are assumed to be the same with Figure 3. Compared with EDF in Figure 3, the extra-buffer requirement is reduced significantly when

LDL algorithm is employed. Given a set of real time tasks ordered by deadline, we can recursively derive the start time s_i and finish time f_i of each tasks by using the following algorithm.

LDL Algorithm:

Input: A set of tasks (1, 2, ..., n) sorted in an ascending order of deadlines.

Output: start time s and finish time *f* of each task. The variable *feasible* is set to be TRUE if a feasible schedule is found; otherwise it is FALSE.

Begin

```
/* Initialization. */
f_n := d_n ;
s_n = f_n — c_n ;
feasible := TRUE ;

for i := n—1 to 1 { /* Go over the task list from */
        f_i := min (d_i, s_i+1) ;
        s_i := f_i — c_i ;
        if (s_i < r_i) then {
                feasible := FALSE ;
                break ;
        }
} /* end of for loop */
```

End

3.2 Largest-Size-Last Algorithm

With largest-size-last (LSL), the objects with larger size have higher priority to be scheduled closer to their deadline. The idea behind is to recognize that objects of larger size would occupy larger amount of buffer space when prefetching is required. This implies that if objects of large size can be scheduled as much as close to its deadline, the total requirement of extra-buffer would be reduced. By LSL algorithm, we schedule tasks in a descending order of object size with objects of larger size being scheduled first. While scheduling, we keep a *reservation list* which maintain a set of reserved intervals. A reserved interval in reservation list is a time interval that is assigned to a task in this schedule; it is reserved for tasks that have been scheduled. By LSL algorithm, we first select the task with largest size and schedule it by aligning the finish time to its deadline. The reservation list is then updated by attaching the reserved interval of time between start time and finish time of the newly scheduled task. Each time the reservation list is changed, all the unscheduled task with time window overlapped with reservation list are modified by excluding the reserved interval from its time window. We refer to the resulting windows as the *surviving window*. Each unscheduled task is then scheduled within one of its own surviving time window with finish time as much as close to deadline.

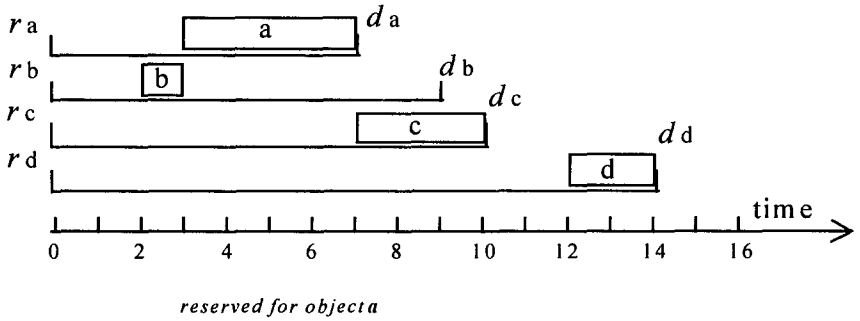


Figure 5: Schedule by using LSL algorithm

Figure 5 delineates the schedule by using LSL algorithm, where only object **b** is prefetched. In Figure 5, object **a** with largest size of four units has the highest priority to be scheduled closest to its deadline, and then is object **c, d** and **b**. When object **a** is scheduled within [3, 7], the reservation list is modified by attaching time interval [3, 7] to it and the time windows of all the unscheduled object **b**, **c** and **d** are also updated. This results in two surviving windows, [0, 3] and [7, 10], for object **c** as depicted in Figure 5. In order to have finish time closest to deadline, object **c** is scheduled within surviving window [7, 10]. The same processes last until all the objects are scheduled. By using graph-coloring technique, five buffer units are required for accommodating all the objects, where the degrees of objects are assumed to be the same with Figure 3. Compared with EDF in Figure 3, the extra-buffer requirement is reduced significantly when LSL algorithm is employed. Given a set of real time tasks ordered by their sizes, we can recursively derive the start time s_i and finish time f_i of each tasks by using the following algorithm.

LSL Algorithm:

Input: A set of tasks (1, 2, ..., n) sorted in an ascending order of their sizes.
Output: start time s and finish time *f* of each task. The variable *feasible* is set to be TRUE if a feasible schedule is found; otherwise it is FALSE.
Begin
/* Initialization. *rlist* denotes reservation list, w_i the surviving windows for task i, and $\#(w_i)$ the number of the surviving windows */

rlist = Nil ; /* set reservation list *rlist* to be empty */
$\#(w_1) := \#(w_2) := ... := \#(w_n) := 1$;
for i := 1 to n {
$r_{i1} := r_i$;
$d_{i1} := d_i$;
}
i := n ;
While ((*feasible* = TRUE) Λ (i > 0)) {
feasible := FALSE;

for j := $\#(w_i)$ **to** 1 { /* go over w_i to find a time interval for task i */

```
                    f_i := d_ij ;
                    s_i := f_i — c_i ;
                    if ( s_i > r_ij ) {        /* a feasible schedule is found */
                            feasible := TRUE;
                            attach (s_i, f_i) to rlist ;  /* update the reservation list*/
                            for k:= 1 to i—1
                                    /* update serviving windows of all the
                                       unscheduled objects */

                                    exclude interval (s_i, f_i) from w_k;
                            }
                    }
                    if (feasible = TRUE) break;
            }  /* end of loop j */
    i := i — 1 ;
    }  /* end of while loop */
End
```

4. Object Migration Strategy

The Object Migration Strategy reduces the extra-buffer requirement by regarding the disk storage as a special buffer, that is, adopt the disk storage instead of memory buffer to accommodate objects. This strategy works well in a multi-disk environment. In a multi-disk environment, there are distinct meta-I/O schedules for different disks. Assuming that object *A* need to be prefetched into extra-buffer as a result of meta-I/O schedule in some disk (say source disk, denoted by D_s). The extra-buffer can be released if object *A* can be written to another disk (say target disk, denoted by D_t) during its prefetching interval so that it can be retrieved directly from D_t instead of D_s before its deadline. Such an operation is called the *migration* which involves one *read* operation at source disk and two operations at target disk, a *write* and a *read* operation. For a object *i*, a migration operation can be performed only when there is a feasible schedule in target disk for writing and reading object *i* within its prefetching interval $[s_i, d_i]$, where s_i and d_i is the start time and deadline of object i yielded from the schedule at source disk. As an example in Figure 6, the migration for object **a** (prefetched from disk Ds) can be performed iff a feasible schedule is found for writing and reading object **a** at disk Dd within time interval $[s_a, d_a]$.

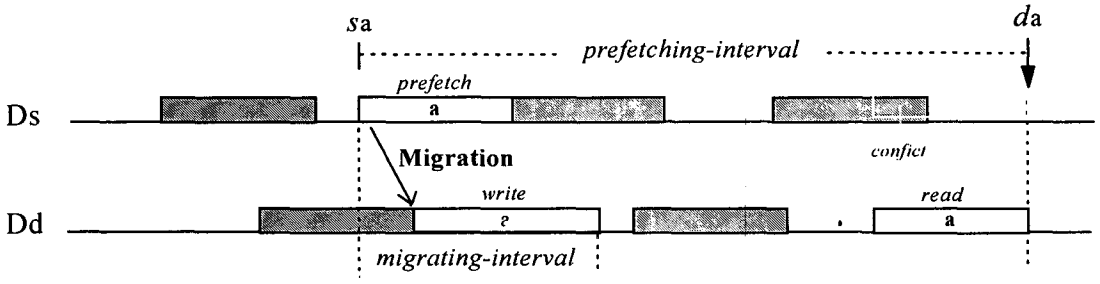


Figure 6: Object Migration Strategy

The migration strategy can help reduce buffer requirement by reducing the prefetching intervals of objects. With migration strategy, extra-buffer requirement starts from the time object is prefetched at source disk and it can be released when the object has been moved to the target disk. We refer to the interval between start time of reading operation at source disk and finish time of writing operation at target disk as the *migrating-interval*. This means that extra-buffer holding time for accommodating the object is reduced to migrating-interval instead of prefetching-interval. Owing to shorter extra-buffer holding time, the buffer can be reused soon for another object such that total amount of extra-buffer requirement is reduced. Another benefit yielded from performing migration operation is *load balancing* among disks in the system. Although several researches have focus on load balancing storage design based upon data popularity, it is hardly to precisely catch all the user behaviors that may depends on several unpredictable factors. At this moment, real-time scheduling with on-line object migration can help system afford more unpredictable traffics and achieve a better performance.

Our problem at hand is to determine a candidate for migration such that more buffer can be released. For the purpose, following two strategies are proposed: 1. *Large Object Migrating*, 2. *Long-Prefetched Object Migrating*. For Large Object Migrating strategy, the objects with largest size have the highest priority to be migrated first, while, for Long-Prefetched Object Migrating strategy, the objects with longest prefetching interval are first considered to be migrated. The reason is that the objects with larger size will need larger buffer space for accommodating and the objects with longer prefetching interval will occupy the buffer resource for a longer time. Both of them lead to more buffer requirement. Namely, when the two types of objects are migrated to an available disk, we can reduce buffer requirement significantly for the system. Migration strategy can be employed together with any real-time scheduling algorithm in a multi-disk environment.

5. Simulation

In our simulations, two criteria are used to compare the performance of various scheduling algorithms: 1. *Average Delay Time*: Upon request arrival, the start time of the request is generated if a feasible schedule is found, or the request is delayed. The delay time of a request is measured as the difference between arrival time and start time of the request. 2. *Buffer Requirement*: The amount of buffer required to hold the data, measured by the proposed buffer measurement approach.

5.1 Simulation Model

In our simulations, the user request arrivals are modeled by using a *Poisson* process and the file selection for a request is modeled by using *uniform* distribution. Each file in the simulation is characterized by a triple $[O_n, O_s, O_d]$, where O_n is the number of real-time objects in the file, O_s is the object size which is defined by the time for the object to be retrieved from disk, and the O_d is the *inter-object interval* which is defined by the deadline difference between two successive objects in this file. The default values of parameters used in the simulation are given in Table 1, where the second column (M) is the mean value and the third column (V) the variation. All the parameters, except to the request arrival (which is modeled as a Poisson distribution), are generated uniformly

distributed in the interval [M(1—V), M(1+V)]. For example, object sizes in our simulations are uniformly distributed in the interval [0, 20]. The deadlines of all the objects a file are *relative* to the deadline of the first object in this file. In our simulations, the user request for a file can be delay but the individual relative deadline in this file cannot be missed. The reason is that, in the domain of multimedia applications, it would be meaningless if related objects cannot be synchronized for display, e.g., video and audio. The display of the entire file must be delay if any individual deadline cannot be met. Moreover, in our simulations, we strip each file across multiple disks based upon the storage allocation approach proposed in [10], by which two real-time objects with overlapped interval of retrieval time are stored on different disks to avoid retrieval conflicts. In the multidisk environment, the schedule for a user request is said to be feasible iff we can find a feasible schedule for each disk (Note that, the requested file may consist of several objects reside on different disks).

We conduct two sets of experiments. For the first set, four real-time scheduling were examined in our study, which are Earliest-Deadline-First (EDF), Least-Laxity-First (LLF), Latest-Deadline-Last (LDL) and Largest-Size-Last (LSL). For the second set of experiments, two migration strategies, the Large object Migrating (say M1) and the Long-Prefetched Object Migrating (say M2), employed in different scheduling algorithms (EDF, LDL and LSL) were examined.

Table 1: The default value of parameters used in the simulations

Symbol	Mean (M)	Var (V)	Description
d	8	0.0	the number of disks
O_n	6	1.0	the number of real-time objects in a file
O_s	10 sec	1.0	the real-time object size
O_d	20 sec	1.0	the inter-object interval for real-time objects in a file
R_i	60 sec		the interarrival time of requests.

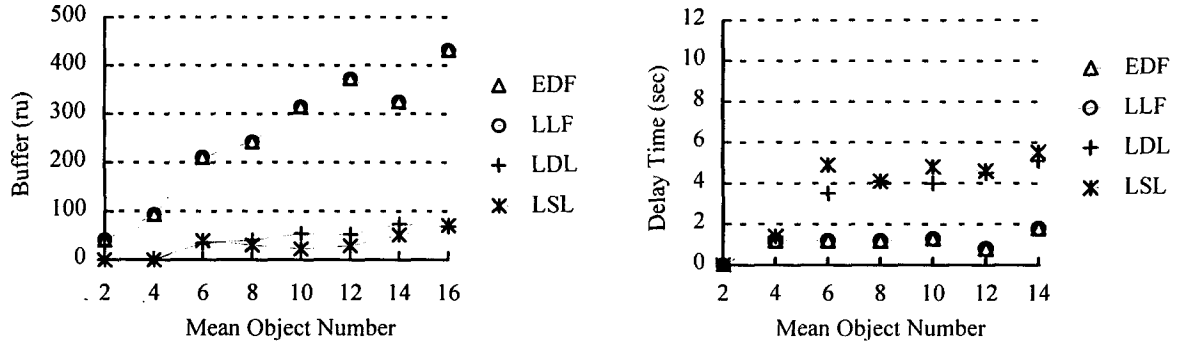
5.2 Simulation Results and Discussion

For the first set of experiments, Figure 7, 8 and 9 show the performance of scheduling with EDF, LLF, LDL and LSL algorithm, parameterized by different number of objects in a file, object size and interarrival time of requests, respectively. Figure 7.(a), 8.(a) and 9.(a) show the buffer requirement for scheduling algorithms, measured in *retrieval units* (ru), while Figure 7.(b), 8.(b) and 9.(b) show the average delay time per request, measured in seconds (sec). The retrieval unit is defined by the amount of data retrieved from storage per second, which is varying with different storage. After comparing the results, we can make the following observations.

1. It can be seen that, with the increase of either the mean number of real-time objects in a file or the average size of objects, the buffer requirement and the average delay time increase for all algorithms as shown in Figure 7 and 8. Furthermore, both buffer requirement and average delay time drops for all algorithms when mean inter-arrival time of requests increases as shown in Figure 9.
2. Figure 7.(b), 8.(b) and 9.(c) show that LDL and LSL have a relatively worse average delay time over most ranges of parameters. Since a request will be delay when a feasible schedule

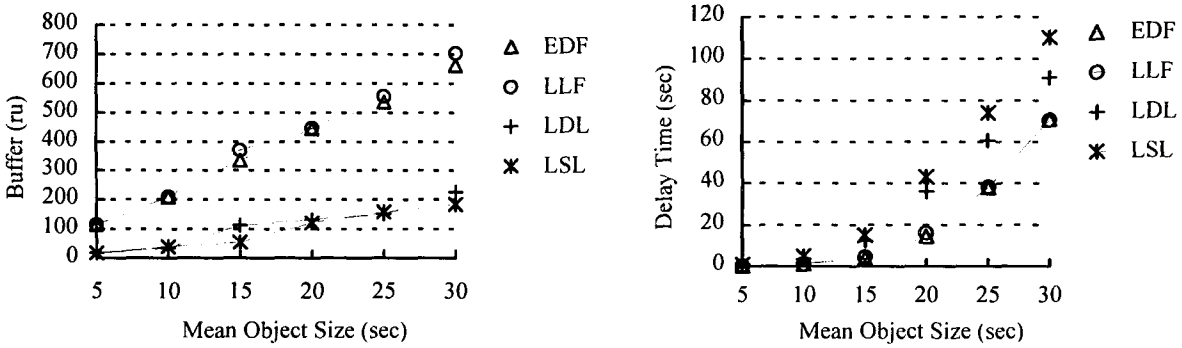
cannot be found, the results implies that both of them have the worse success ratio of generating a feasible schedule.

3. Although LDL and LSL have a relatively worse average delay time, both of them achieve a significant improvement for reducing buffer requirement in comparison with EDF and LLF as can be seen in Figure 7.(a), 8.(a) and 9.(a). The result implies that both LDL and LSL result in fewer objects to be prefetched and hence fewer buffer is required for holding these data. We also notice that, LSL performs better than LDL in terms of reducing buffer requirement.
4. The results demonstrate that, by sacrificing a little average delay time of requests (within tens of seconds), the two proposed LDL and LSL algorithms can reduce buffer requirement significantly (more than hundreds of seconds) for real-time systems.



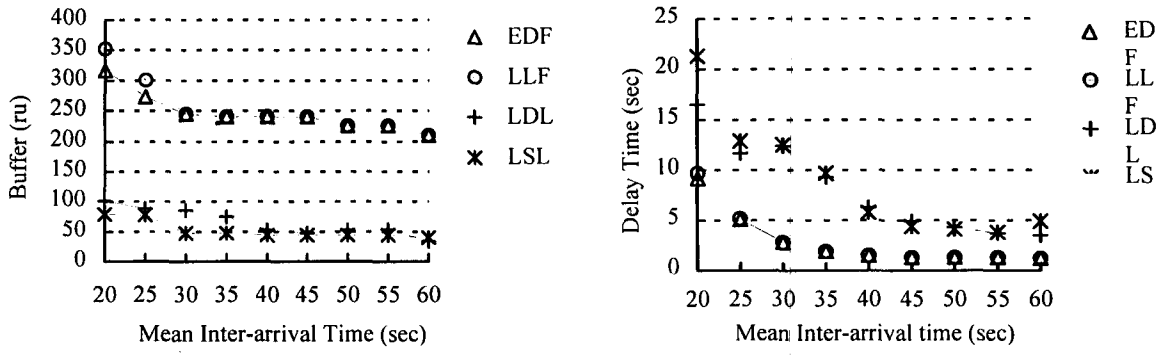
(a) (b)

Figure 7: The effect of the number of real-time objects in a file



(a) (b)

Figure 8: The effect of object size

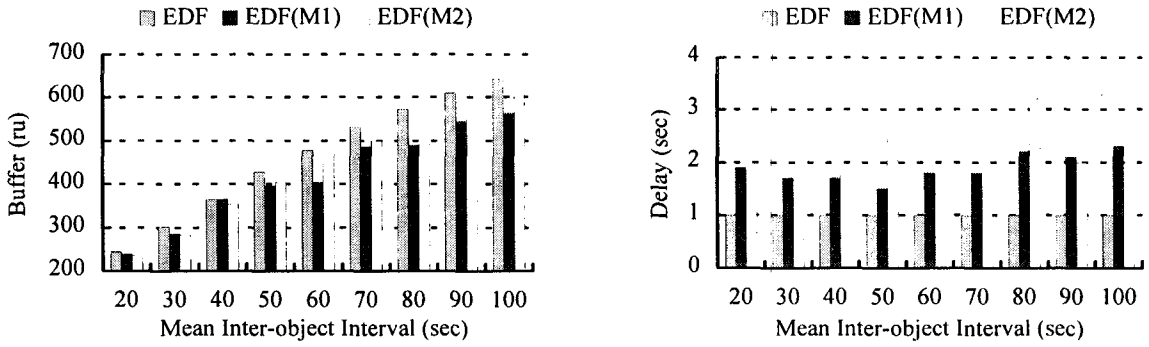


(a) (b)

Figure 9: The effect of inter-arrival time of requests

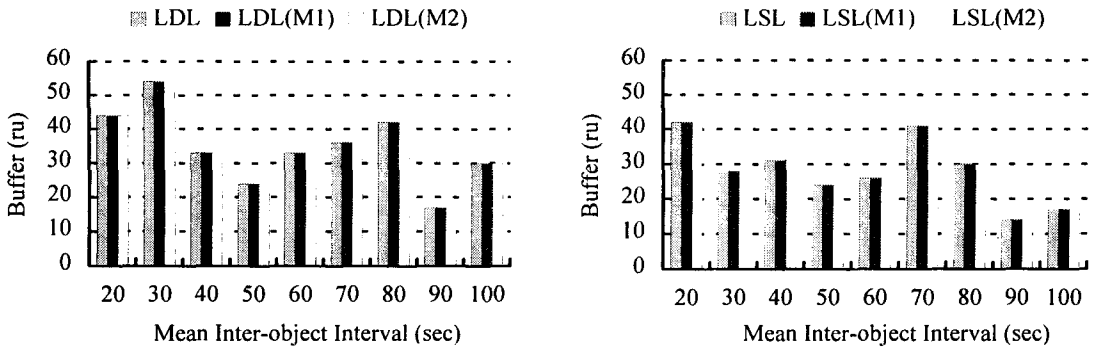
For the second set of experiments, Figure 10 shows the performance of migration when EDF is employed parameterized by the inter-object interval of real-time objects in a file, while Figure 11 shows the performance of migration when LDL and LSL are employed. The results show that migration strategies perform well in reducing buffer requirement when algorithm EDF is employed as depicted in Figure 10.(a). Note that, when the mean of O_d (the inter-object interval) increases, the buffer requirement drops, with M1 dropping faster than M2. The reason is that, fewer migrating candidates can be found for a smaller O_d when large object migrating strategy is used. In addition, Figure 10.(b) shows that migration strategies will lead to a worse average delay time bacon more disk bandwidth are wasted for migration operation and hence fewer feasible schedule can be generated for EDF.

We also notice that, both M1 and M2 do not perform well when LDL or LSL is employed as shown in Figure 11. The reason is that, both LDL and LSL generate schedules with fewer prefetched objects and shorter prefetching interval, therefore no migrating candidates can be found. The results also demonstrate that both LDL and LSL achieve a better performance in reducing buffer requirement than EDF, even though when migration strategies are employed in EDF algorithm.



(a) (b)

Figure 10: The effect of migration in EDF algorithm



(a) (b)

Figure 11: The effect of migration in LDL and LSL algorithms

6. Conclusion

Continuity is an important issue in the domain of multimedia applications. To ensure that continuous retrieval can be guaranteed in the presence of multiusers, a feasible scheduling mechanism is prerequisite for real time data retrieval from storage subsystem. However, several real-time algorithms such as earliest-deadline-first (EDF) suffer from wasting a large amount of buffer when employed in solving meta-I/O scheduling problem. In this paper, several techniques to reduce buffer requirement were explored for real-time meta-I/O scheduling, including latest-deadline-last (LDL), largest-size-last (LSL) and migration strategies. Moreover, we also propose a buffer measurement approach to estimate the performance of various scheduling algorithms.

We conducted a simulation study of the proposed real-time algorithms. By comparing four algorithms, namely earliest-deadline-first (EDF), least-laxity-first (LLF), latest-deadline-last (LDL) and (largest-size-last) LSL, we find that both LDL and LSL perform much than EDF and LLF for real-time meta-I/O scheduling. Although LDL and LSL have a relative worse average delay time for requests, the reduced buffer requirement in significant for scheduling in real-time systems. Furthermore, the simulation results also show that migration strategies achieve an improvement in reducing buffer requirement for some real-time algorithms in multi-disk systems. On-line migrating objects between disks also achieve a potential load-balancing among disks, which promotes the performance of the entire system. In conclusion, we believe that the proposed real-time algorithms and migration strategies show a promising prospect towards solving buffer requirement problem for meta-I/O scheduling in real-time systems.

Reference

[1] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon, "Coloring heuristics for register allocation," *SIGPLAN Notices*, Vol. 24. Iss. 7, July 1989.

[2] D. Callahan and B. Koblenz, "Register allocation via hierarchical graph coloring," *SIGPLAN Notices*, Vol. 26, Iss. 6, June 1991.

[3] F. C. Chow, J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Transactions on Programming Languages and Systems*, Vol. 12, Iss. 4, Oct. 1990.

[4] J. R. Jackson, "Scheduling a production line to minimize maximum tardiness," Technical Report Management Science Res. Rep. 43, UCLA, 1955.
[5] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, Vol. 20, No. 1, Jan. 1973.
[6] R. E. Larson and M. I. Dessouky, "Heuristic procedures for the single machine problem to minimize maximum lateness," AIIE Trans, vol. 10, 1978.
[7] A. L. N. Reddy and J. Wyllie, "Scheduling in a multimedia I/O system," *Pro ACM Multimedia Conf.*, ACM Press, New York, 1992.
[8] A. L. N. Reddy and J. C. Wyllie, "I/O issues in a multimedia system," *IEEE Comput.*, March 1994.
[9] P. G. Sorenson, "A methodology for real-time system development," Ph.D. thesis, University of Maryland, College Park, MD 20742, 1974.
[10] W. J. Tsai and S. Y. Lee, "Storage design and retrieval of continuous multimedia data using multi-disks," ICPADS, 1994.