Memory Allocation Technique for Segregated Free List Based on Genetic Algorithm

Manal F. Younis
Computer Department, College of Engineering, University of Baghdad.
E-mail: manal_fadel2@yahoo.com.

Abstract

Dynamic memory management is an important part of computer systems design. Efficient memory allocation, garbage collection and compaction are becoming increasingly more critical in parallel, distributed and real-time applications. The memory efficiency is related to the fragmentation. Segregation is one of the simplest allocation policies which use a set of free lists, where each list holds blocks of a particular size. When the process requests a memory. The free list for the appropriate size is used to satisfy the request. This paper proposes a scheme to reduce the internal fragmentation of a segregated free list for improving memory efficiency using genetic algorithm (GA) to find the optimal configuration. Because the genetic algorithms (GAs) are largely used in optimization problems, they facilitate a good alternative in problem areas where the number of constraints is too large for humans to efficiently evaluate. This GA is tested under five randomly created workloads to find the best configuration. The results are acceptable when compared with optimal configurations of these workloads.

Keyword: Memory allocation, Segregated free list, Genetic algorithm, Crossover, Mutation.

Introduction

Dynamic memory allocation is a classic problem in computer systems. Typically, we start with a large block of memory (sometimes called a heap). When a user process needs memory, the request is granted by carving a piece out of the large block of memory. The user process may free some of the allocated memory explicitly, or the system will reclaim the memory when the process terminates. At any time the large memory block is split into smaller blocks (or chunks), some of which are allocated to a process (live memory), some are freed (available for future allocations), and some are no-longer used by the process but are not available for allocation (garbage). A dynamic memory management system must keep track of these three types of memory blocks and attempt to efficiently satisfy as many of the process's requests for memory as possible [1].

Memory allocation schemes can be classified into Sequential Fit, Buddy System and Segregated free list algorithms. The Sequential Fit approach (including First Fit, Best Fit) keeps track of available chunks of memory on a list. Known sequential techniques differ in how they track the memory blocks and how they allocate memory requests from the free blocks. Normally the chunks of memory (or at least the free chunks) are maintained as a Linear Linked list. When a process releases memory, these chunks are added to the free list, either at the end or in place if the list is sorted by addresses; freed chunk may be **coalesced** with adjoining chunks to form larger chunks of free memory. When an allocation request arrives, the free list is searched until an appropriately sized chunk is found. The memory is allocated either by granting the entire chunk or by **splitting** the chunk (if the chunk is larger than the requested size). Best Fit methods try to find the smallest chunk that is at least as large as the request. First Fit methods will find the first chunk that is at least as large as the request. Best Fit method may involve delays in allocation while First Fit method may lead to more external fragmentation. If the free list is in address order, newly freed chunks may be combined with its surrounding blocks, leading to larger chunks. However, this requires a "linear" search through the free list when inserting a newly freed block of memory (or when searching for a suitable chunk of memory) [1].

Buddy system algorithm maintains free lists of different sized blocks. When a request for memory is made these free lists are

searched. If the appropriate size is not found a larger block is split (variations of this algorithm determine how the block is actually split, for example, in a binary buddy system the block is split by powers of two). It will continue this splitting, and the "buddy" or other half is added to the free list, until the requested size is found. When memory is freed it looks for its buddy, or the block it split from, to regain its original size. For example, if 10 bytes are requested the allocator searches the free list. The only available block is 32 bytes. This block splits into two blocks of 16 bytes. One block of 16 bytes is allocated and the other block is put on the free list. When it frees this memory it then looks to the free list for its buddy and coalescing takes place [2].

The Segregated free list approach maintains multiple linked lists, one for each different sized chunk of available lists. Returning a free chunk from one of the lists satisfies allocation requests (by selecting a list containing chunks, which are at least as large as the request). Freeing memory, likewise, will simply add the chunk to the appropriate list. No coalescing or splitting is performed and the size of chunks remains unaltered. The main advantage of segregated lists is the execution efficiency in allocating and freeing memory chunks. The disadvantage is the inefficient usage of memory. The memory is divided into regions based on the different sized blocks. Since the number and frequency of requests for different sized chunks depends on the application and even inability to satisfy all requests from the application [1].

This paper interested with the segregated free list layout which does not have the problem of external fragmentation, but rather of internal fragmentation when a small block is allocated into larger blocks. In segregated free lists, the requests are served using bins (i.e., an array of free lists) where each bin contains blocks (chunks) of the same size, see Fig. (1). However, if the exact size does not exist or that bin does not have free chunks of memory, the request is served by the next larger block than is necessary, but the remaining part is not split and no coalesces. When this occurs, we have memory waste and an increase in internal fragmentation.

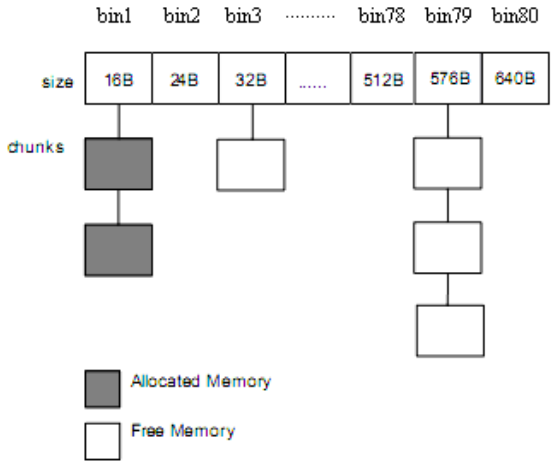


Fig.(1) Data structure of segregated free list.

The rest of this paper is organized as follows: Section 2 illustrates the related work. Section 3 describes genetic algorithm. Section 4 presents the proposed memory allocation approach. Section 5 presents the experimental results. Finally, section 6 illustrates the conclusion and some future work.

Related Work

Rezaei M., Cytron R. K. presented how to exploit Intelligent Memory Devices to decouple the memory management from the central processing unit, and show how segregated binary trees can be embedded in intelligent memory devices [3]. Rosso C. D. presented an approach for improving the internal memory fragmentation by finding the optimal configuration of a segregated free lists data structure using genetic algorithm. The genetic algorithm used the workload as input to generate the optimal configuration among the huge number of potential solutions by evolving an initial population [4]. Rosso C. D. presented a case study of the evaluation and the analysis of dynamic memory management in embedded real-time systems. They have used a scenario-based approach and used a simulation environment to evaluate the performance of different dynamic memory management systems [5]. Rezaei M. and Kavi1 K. M. presented a technique that uses a Binary tree for the list of available memory blocks and show how this method can manage memory more efficiently and facilitate easy implementation of well known garbage collection techniques, [6]. Masmano M., Ripoll I., Balbastre P., Crespo A. proposed a new allocator called Two Level Segregated Fit

(TLSF) which can be represented as a two-dimensional array. The first dimension splits free blocks in size-ranges a power of two apart from each other, so that first-level index I refers to free blocks of sizes in the range $[2^i,2^{i+1}]$. The second dimension splits each first-level range linearly in a number of ranges of an equal width, [7].

Genetic algorithm (GA)

Genetic algorithms (GAs) are adaptive methods which may be used to solve search and optimization problems. By starting with a population of possible solutions and changing them during several iterations, GAs hope to converge to the fittest solution. Each solution is represented through a chromosome, which is just an abstract representation. The process begins with a set of potential solutions or chromosomes that are randomly generated or selected. Over many generations, natural populations evolve according to the principles of natural selection and survival of the fittest. For generating new chromosomes, GA can use both crossover and mutation techniques. Crossover involves splitting two chromosomes and then, combining one half of each chromosome with the other pair. The idea behind crossover is that the new chromosome may be better than both of the parents if it takes the best characteristics from each of the parents. Crossover occurs during evolution according to a user-definable crossover probability (Pc). Pc normally set to high, e.g., 0.6 [7]. Mutation involves flipping a single bit of a chromosome [8]. Mutation is an important part of the genetic search as help helps to prevent the population from stagnating at any local optima. Mutation occurs during evolution according to a user-definable mutation probability (Pm). This probability should usually be set fairly low (0.01 is a good first choice). If it is set to high, the search will turn into a primitive random search [7]. The chromosomes are then evaluated using a certain fitness criterion and the ones which satisfy the most this criterion are kept while the others are discarded. This process repeats until the population converges toward the optimal solution. The basic genetic algorithm is summarized in Fig. (2), [8].

```
SELECT random population of n chromosomes.
EVALUATE the fitness f(x) of each chromosome
x in the population.
LOOP
   SELECT two parent chromosomes from a
   population.
   CROSSOVER the parents to form new children
   with a crossover probability Pc.
   MUTATE new children with a mutation
   probability Pm.
   Place new offspring in the new population.
   Use new generated population for a further
   Sum of the algorithm.
   EXIT if the end condition is satisfied and
   Return best solution.
END LOOP
```

Fig.(2) A basic Genetic Algorithm.

There are several advantages to the Genetic Algorithm such as their parallelism and their liability. They require no knowledge or gradient information about the response surface, they are resistant to becoming trapped in local optima and they perform very well for large-scale optimization problems. GAs have been used as heuristics to solve difficult problems (such as NP-hard problems) for machine learning and also for evolving simple programs. Applications of Genetic Algorithms include: nonlinear programming, stochastic programming, signal processing and combinatorial optimization problems such as the Traveling Salesman Problem, Knapsack Problem, sequence scheduling, graph coloring, [8].

The proposed memory allocation approach

The proposed work uses genetic algorithm to reduce the internal fragmentation for a segregated free list. Five samples of memory allocation are used where each workload has a random number of requests. An attempt to find the optimal configuration for segregated free list data structure by giving the minimum and maximum values of the bins, and the processor word size. The number of the total bins of each configuration is calculated by the following equation [4]:

$$n = \frac{Max - Min}{Word\ size} + 1 \quad \text{.....(1)}$$

Where:

n represents the number of bins between max and min values.

Max is the maximum number of bins

Min is the minimum number of bins

When the number of bins is high then the search space for the solution is large and a brute force approach is not feasible. Therefore, an approach based on genetic algorithm is used to find an optimal or near optimal solution. Genetic algorithm provides a heuristic approach to function optimization problems, which have the concepts of fitness, crossover, mutation, populations and genes. In this paper we take five workloads and applied the GA for each of them in an attempt to find the optimal configuration.

The first workload is randomly selected between 8 to 400 bytes; the number of bins equals to 50 is calculated using formula (1) as shown in Fig. (3).

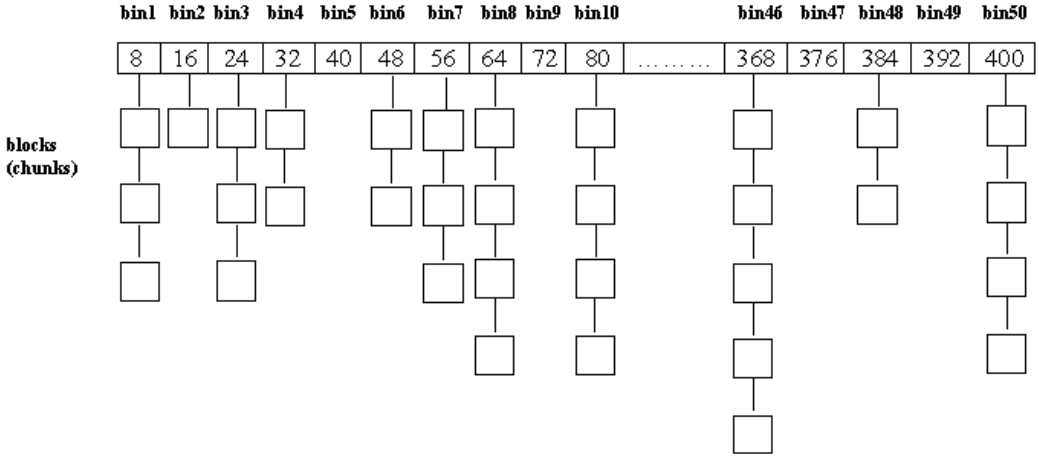


Fig. (3) First workload.

For example, if we need 8 bins to represent the optimal configuration of a segregated free list, there are about 536,878,650 times to find it. It is a large solution space to get the optimal 8 bins among 50 bins therefore; we use GA.

In this paper we represent each bin as a gene which is an encoded parameter. A chromosome is the string produced by concatenating all the encoded parameters. Each chromosome is an individual and member of a population which represents the segregated free list. The chromosome includes n genes for example in first workload (n=8), where each gene holds the size of a given bin as in Fig.(4). Note that each gene represents one bin of a particular size (e.g., gene1 indicates the existence of bin2 of size 16).

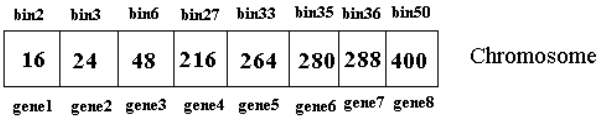


Fig. (4) Chromosome representation.

The following steps show how the GA used to obtain the optimal solution:

- **Step one**: is generation of a population randomly, and then calculates the fitness of each chromosome; which represents the summation of number of requests for each bin (the genes in the same chromosome are not duplicated).
- **Step two**: Select two parents 'chromosomes' from the population and take the parent whose fitness is greater than the other.
- **Step three**: Produce a child as offspring from the parents using crossover by taking crossover probability (Pc) = 0.6 which decides if the parts of two chromosomes will be interchanged. This is determined between two points. For example, if we take $point_1$ = 3 and $point_2$ = 6 the genes from location 3 to location 6 are changed between two chromosomes, see Fig.(5).

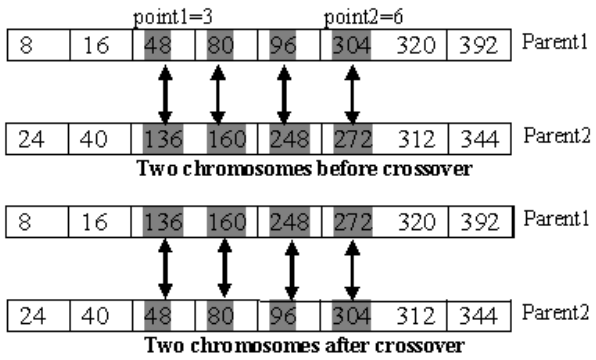


Fig. (5) Crossover between two chromosomes.

The genes which had been undergo crossover and swapped between the two parent chromosomes must not be duplicated (i.e., not

similar to any other gene in the same child chromosome). If duplicated, we have to replace it randomly with a different value from the original set of bin sizes (8-400).

- **Step four**: Mutate the child by taking probability mutation (Pm) = 0.01 to decide which gene(s) is/are changed randomly. A mutation operator that replaces the value of the chosen gene with a random value selects. The gene which is replaced must not similar to any other gene in the same chromosome. Then calculate the fitness of the created child.
- **Step five**: Put new offspring in the population and use this new generated population for further.
- **Step six**: If the number of generation is equal to end of loop (in this example we take it 50 iterations) then return the best solution with maximum fitness value.

Experimental results

This paper uses five randomly created workloads; each of them represents a particular segregated free list. The characteristics of these workloads, together with their memory allocation structures are given in table1 and 2, respectively. In Tables (2-1, 2-2, 2-3, 2-4, 2-5) alloc. size is the size of memory allocation (bin) and #requests is the number of requests for an allocation.

Table (1)

Characteristics of five workloads.

Workload No.	*Min – Max bins*	*Word size*	*No. of bins*	*No. of bins required*
1	8-400	8	8	8
2	16-536	4	131	13
3	32-400	16	5	5
4	4-1024	4	15	15
5	4-80	4	5	5

Table (2-1)

The memory allocation structure of the first workload.

No.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Alloc. Size	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128	136	144	152	160
# requests	22	24	10	4	10	10	0	6	10	12	5	12	20	5	6	10	20	8	5	20

No.	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**
Alloc. Size	168	176	184	192	200	208	216	224	232	240	248	256	264	272	280	288	296	304	312	320
# requests	6	10	16	8	12	5	14	10	12	5	25	10	8	30	20	10	5	15	13	20

No.	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**
Alloc. Size	328	336	344	352	360	368	376	384	392	400
# requests	8	20	15	20	4	8	13	5	21	8

Table (2-2)

The memory allocation structure of the second workload.

No.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Alloc. Size	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92
# requests	0	4	10	0	12	0	3	0	0	8	1	4	0	0	2	0	4	0	0	0

No.	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**
Alloc. Size	96	100	104	108	112	116	120	124	128	132	136	140	144	148	152	156	160	164	168	172
# requests	0	4	0	1	2	1	3	5	0	0	3	0	0	1	0	6	1	0	3	6

No.	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**
Alloc. Size	176	180	184	188	192	196	200	204	208	212	216	220	224	228	232	236	240	244	248	252
# requests	0	0	2	0	2	6	5	2	5	4	2	4	0	6	0	0	4	0	0	0

No.	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**	**75**	**76**	**77**	**78**	**79**	**80**
Alloc. Size	256	260	264	268	272	276	280	284	288	292	296	300	304	308	312	316	320	324	328	332
# requests	0	3	1	7	0	7	0	2	1	0	3	8	4	5	4	6	2	0	0	0

No.	**81**	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**	**95**	**96**	**97**	**98**	**99**	**100**
Alloc. Size	336	340	344	348	352	356	360	364	368	372	376	380	384	388	392	396	400	404	408	412
# requests	0	0	12	0	0	0	0	0	1	10	10	0	3	4	0	0	0	0	0	6

No.	**101**	**102**	**103**	**104**	**105**	**106**	**107**	**108**	**109**	**110**	**111**	**112**	**113**	**114**	**115**	**116**	**117**	**118**	**119**	**120**
Alloc. Size	416	420	424	428	432	436	440	444	448	452	456	460	464	468	472	476	480	484	488	492
# requests	0	7	12	0	3	0	0	0	0	12	0	6	0	10	1	2	0	0	0	3

No.	**121**	**122**	**123**	**124**	**125**	**126**	**127**	**128**	**129**	**130**	**131**
Alloc. Size	496	500	504	508	512	516	520	524	528	532	536
# requests	3	0	11	12	0	2	0	4	1	2	6

Table (2-3)
The memory allocation structure of the third workload.

No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Alloc. Size	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	272	288	304	320	336
# requests	24	10	10	8	20	0	6	0	12	5	0	23	50	6	10	20	0	5	20	6

No.	21	22	23	24
Alloc. Size	352	368	384	400
# requests	10	17	0	24

Table (2-4)
The memory allocation structure of the fourth workload.

No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Alloc. Size	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80
# requests	10	14	0	20	40	0	0	0	0	0	10	0	8	1	4	0	7	0	12	4

No.	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Alloc. Size	84	88	92	96	100	104	108	112	116	120	124	128	132	136	140	144	148	152	156	160
# requests	10	0	4	7	4	0	1	2	6	3	5	17	0	3	0	0	0	9	6	1

No.	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
Alloc. Size	164	168	172	176	180	184	188	192	196	200	204	208	212	216	220	224	228	232	236	240
# requests	0	3	6	0	0	2	0	2	6	5	2	5	4	2	4	0	6	0	0	4

No.	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
Alloc. Size	244	248	252	256	260	264	268	272	276	280	284	288	292	296	300	304	308	312	316	320
# requests	0	8	0	0	3	1	7	0	7	0	2	1	0	13	8	4	5	4	6	2

No.	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
Alloc. Size	324	328	332	336	340	344	348	352	356	360	364	368	372	376	380	384	388	392	396	400
# requests	0	0	0	23	13	12	0	0	0	0	0	1	10	10	0	3	4	0	0	0

No.	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
Alloc. Size	404	408	412	416	420	424	428	432	436	440	444	448	452	456	460	464	468	472	476	480
# requests	0	0	6	0	7	13	0	3	0	0	0	0	12	0	6	0	10	1	2	0

No.	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
Alloc. Size	484	488	492	496	500	504	508	512	516	520	524	528	532	536	540	544	548	552	556	560
# requests	0	0	3	3	0	11	12	0	2	0	4	1	2	6	12	0	3	0	0	0

No.	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
Alloc. Size	564	568	572	576	580	584	588	592	596	600	604	608	612	616	620	624	628	632	636	640
# requests	0	12	0	6	0	10	1	2	0	12	0	3	3	0	11	12	0	2	20	4

No.	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180
Alloc. Size	644	648	652	656	660	664	668	672	676	680	684	688	692	696	700	704	708	712	716	720
# requests	1	20	12	0	30	0	0	0	0	12	0	6	0	10	1	2	0	0	0	3

No.	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
Alloc. Size	724	728	732	736	740	744	748	752	756	760	764	768	772	776	780	784	788	792	796	800
# requests	3	0	11	12	0	2	0	4	1	2	12	0	3	18	11	0	0	12	0	6

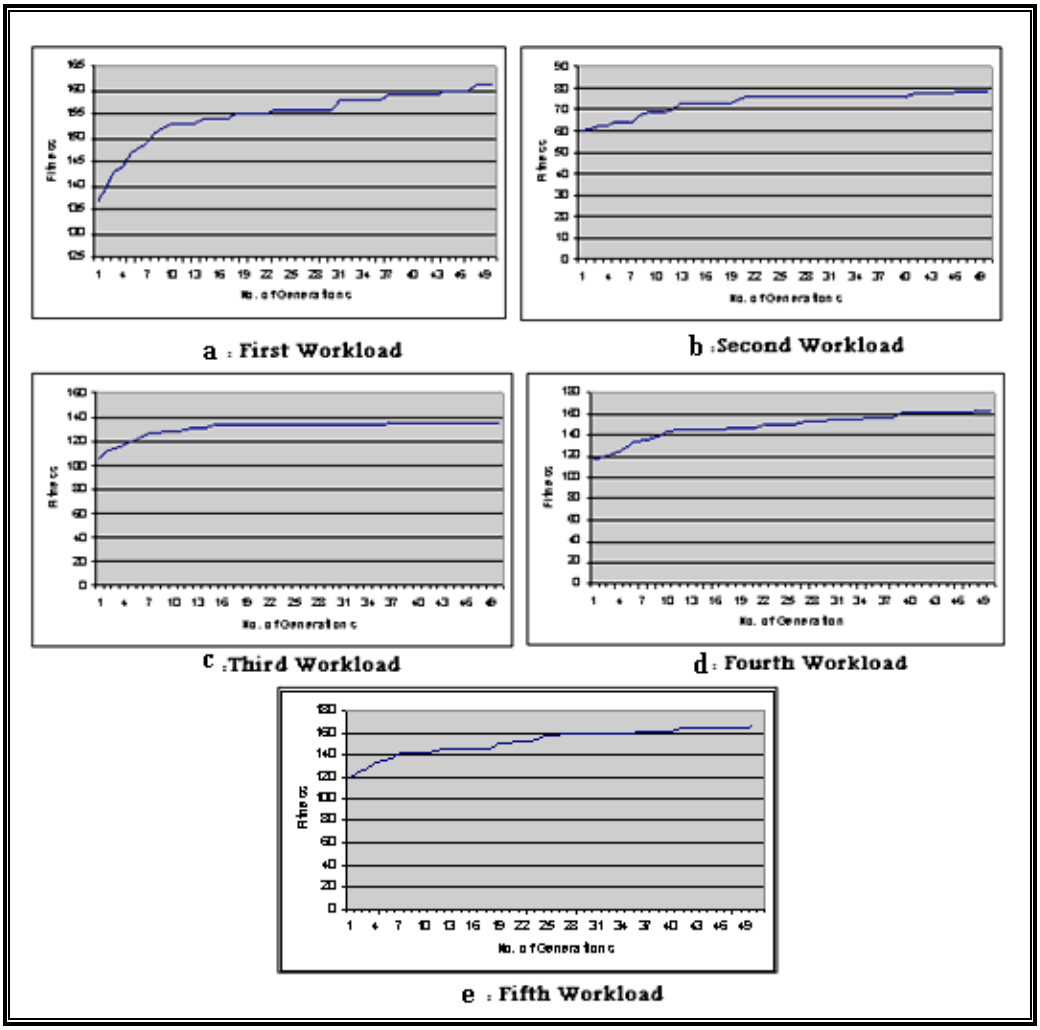
No.	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220
Alloc. Size	804	808	812	816	820	824	828	832	836	840	844	848	852	856	860	864	868	872	876	880
# requests	10	1	2	12	0	20	3	3	0	11	12	0	2	0	4	1	2	12	0	3

No.	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240
Alloc. Size	884	888	892	896	900	904	908	912	916	920	924	928	932	936	940	944	948	952	956	960
# requests	0	0	0	0	12	0	6	0	10	1	2	0	24	0	3	3	11	12	0	2

No.	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
Alloc. Size	964	972	976	980	984	988	992	996	1000	1004	1008	1012	1016	1020	1024
# requests	0	4	1	2	12	0	2	33	4	1	2	12	0	25	10

Table (2-5)
The memory allocation structure of the fifth workload.

No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Alloc. Size	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80
# requests	9	24	10	10	6	20	10	6	10	12	15	0	23	5	6	10	20	0	5	20



Figs. (6) Depict the results of GA for segregated free list for workloads 5-9; respectively.

Fig. (6) shows the relation between fitness and the number of generation graphically, for each workload. The values of the fitness obtained by running the proposed algorithm on each workload 10 times and calculate the average.

Table (3) shows the best solution obtained from the GA for each workload over 10 different runs. In this table the best solution is presented of the five workloads which founded manually with their corresponding fitness values. This table shows how proposed GA converges from the optimal solution founded manually.

Table (3)
The GA results of five workloads.

#Workload	Optimal Solution founded manually	Fitness	Best Solution Found by GA	Fitness	Fitness difference
1	32,96,208,224,400	141	32,272,32,208,224	141	0
2	24,32,52,268,300,344,372,376,424, 452,468,504,508	134	24,32,52,156,208,276,344,372, 376,424,452,504,508	121	13
3	8,16,136,160,248,272,320,392	182	8,304,160,272,136,248,352,16	176	6
4	8,16,20,128,336,340,424,636,648, 660,776,824,932,996,1020	330	4,8,16,20,116,300,336,372,376, 460,648,660,776,824,932	268	62
5	52,80,68,8,24	107	52,80,68,8,24	107	0

We found that the fitness difference is increased when the difference between the number of bins and number of required bins is increased as shown in Table (4).

The best configuration of segregated free list of the first workload after GA is applied as shown in Fig. (7).

bin1	bin2	bin3	bin4	bin5
32	96	208	224	400

Fig. (7) Best configuration of segregated free list for first workload.

Table (4)
The comparison between the GA results.

#Workload	No. of bins	No. of required bins	Difference between the no. of bins and the required bins	Fitness difference
3	24	5	19	0
5	20	5	15	0
2	50	8	42	6
1	131	13	118	13
4	256	15	241	62

Conclusions

In this paper genetic algorithm is used as an attempt to find the best configuration of segregated free list. Five workloads with different characteristics. The results of the proposed algorithm are compared with the best solutions calculated manually. According to the results on the five different workloads, we found that the proposed GA is capable of performing well in finding the required number of bins with suitable fitness values.

In the future the proposed genetic algorithm can be enhanced to adopt more heuristic operators for, e.g., crossover and mutation.

References

[1] Rezaei M., Kavi K.M.; "A New Implementation Technique for Memory Management"; Proceedings of the Southeast Con, Nashville, TN; Vol. 10; pp 1-2; 2000.

[2] Paul R. W., Mark S. J., Michael N., and David B.; "Dynamic Storage Allocation"; Vol. 7, pp 733-756; 1993.

[3] Rezaei M. and Cytron R. K.; "Segregated Binary Trees: Decoupling Memory Manager"; Dept. of Electrical and Computer Engineering; Vol. 23; pp 1-3; 2001.

[4] Rosso C D.; "Reducing Internal Fragmentation in Segregated Free Lists using Genetic Algorithms"; Proceedings of the 2nd International ACM Workshop on Interdisciplinary Software Engineering Research, Vol. 12; pp 143 – 150; 2006.

[5] Rosso C. D.; "The method, the tools and rationales for assessing dynamic memory efficiency in embedded real-time systems in practice"; 2006.

[6 Masmano M., Ripoll I., Balbastre P., Crespo; "A Constant-Time Dynamic Storage Allocator for Real-Time Systems"; Department of Computer Engineering; 2008.

[7] Neuro Dimension ,Web Site Design and Implementation Copyright © 2002, Inc.

[8] Aouad M. I., Schott R., and Zendra O.; "Genetic Heuristics for Reducing Memory Energy Consumption in Embedded Systems"; pp. 3-4; 2010.

الخلاصة

ادارة الذاكرة الداينميكية هي جزء مهم في تصميم انظمة الكمبيوتر. اصبح التخصيص الكفوء للذاكرة و جمع الاجزاء الصغيرة وضغطها من الامور الضرورية والحرجة في الانظمة المتوازية والتوزيعية وتطبيقات الانظمة ذات الوقت الحقيقي. كفاءة الذاكرة لها علاقة بتقسيمها الى اجزاء صغيرة غير مستغلة. تجزاة الذاكرة الى مقاطع مختلفة الحجم التي تنظم بشكل قائمة هي ابسط انواع طرق تخصيص الذاكرة. وعندما يطلب البرنامج ذاكرة يفضل تخصيص الحجم المناسب لطلبه. ففي هذا البحث تم اقتراح طريقة لتقليل الاجزاء الفارغة في المقاطع المقسمة لها الذاكرة باستخدام الخوارزمية الجينية وذلك بايجاد افضل تقسيم للذاكرة. فالخوارزمية الجينية تستخدم لايجاذ الحلول الفضلى وخاصة عندما يكون الاختيار حل واحد من مجموعة كبيرة من الحلول المقترحة. وتم اختبار الحل المقترح (باستخدام الخوارزمية الجينية) على خمسة عينات وكانت النتائج مقبولة مقارنة بالحلول الفضلى المحسوبة يدوياً لهذه العينات.