Chapter 3
DmRT for Symmetric Multiprocessor

In this chapter, a new dynamic memory allocator for the Real-time operating system is proposed which has been designed and implemented for Symmetric multiprocessing (SMP) architecture. It has been named as **DmRT** (Dynamic memory manager for Real-Time systems). This allocator has been designed to achieve consistent and minimum execution time, low fragmentation and satisfy a maximum number of memory block requests. The DmRT has also been compared with the existing dynamic allocators of the real-time operating system. All the design principals such as strategies, policies, and mechanisms will be explained first and then the structure of DmRT with its results will be discussed in this chapter.

3.1 Design Principals

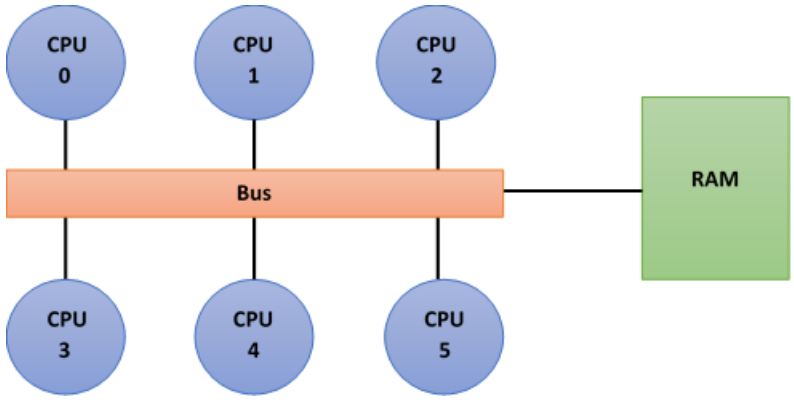

Figure 3.1: SMP Architecture

As shown in Figure 3.1, Symmetric multiprocessing (SMP) system consists of a multiprocessor computer hardware and software architecture in which more than one identical processors are connected to a common or shared main memory. Each processor has full access to all resources like input/output devices which are managed by a single operating system instance treating all processors equally and reserving nothing for special purposes. Nowadays, the majority

of the multiprocessor systems use the SMP architecture. In the multi-core processors, the SMP architecture applies to the cores and treats them as separate processors. There are different strategies for different size of blocks in these allocators which will be explained in this section.

3.1.1 Multiple strategies for different sizes of blocks

As mentioned earlier, various strategies have been used for allocating the different size of blocks to achieve advantages of all policies, strategies, and mechanisms.

I. A small block whose size of memory block < 512 bytes
II. A normal block whose size of memory block < threshold (Some predefined size, i.e. 2Mb)
III. A large block whose size for request exceeding the threshold or (Some predefined size)

3.1.2 Search Policies and Mechanisms

After defining the strategies, the following policies and mechanisms will be used to implement these strategies.

I. For Small blocks, the best-fit policy is used which has been implemented by exact-fit mechanisms to reduce the fragmentation in small sizes of blocks generated by rounding up the request size of memory block [41].
II. For Normal blocks, the good-fit policy is used which has been implemented by segregated lists, which use an array of unallocated block lists**.**
III. For Large blocks, the worst-fit policy has been used.

3.1.3 Arrangement of blocks

DmRT implements the exact-fit mechanism to increase the efficiency of small memory block allocation and decrease the internal fragmentation. It also implements the segregated-fit mechanisms to deploy a good-fit and first-fit policy for searching the nearest segregated size class. Thus, it can ignore the requirement of a thorough search. Here, two types of bitmaps have been

used to keep track of unallocated blocks in the implementation of DmRT allocator. Furthermore, this allocator has used a segregated list with bitmap policies and provides a consistent execution time.

Among the bitmaps, use of one bitmap is to keep the tracks of small memory blocks and is implemented as a two-dimensional array for holding unallocated memory blocks as per the memory blocks size. In DmRT, for effective memory allocation, the block size ranging from 4 bytes to 512 bytes are arranged with a difference of 4 bytes apart. Two different mechanisms have been deployed to check whether a specific size of a memory block is unallocated or not. The first mechanism is that it maintains two bitmaps of 64-bits and the second is maintaining a pointer array to hold the unallocated blocks as shown in Figure 3.2.

As shown in Figure 3.3, The second type of bitmap comprises of a two-dimensional bitmap array pointing to the unallocated memory blocks. The primary bitmap, which is indexed by i, specifies the unallocated memory blocks whose sizes available between 2^i to $2^{i+1} - 1$, and the secondary bitmap, which is indexed by j, splits each primary level range in similar width of a number of ranges. For the simplicity, the number of ranges in the secondary level is specified in power of two: $\mathbf{2}^{\boldsymbol{range}}$. For this allocator, the default value of ***range*** is taken as 6. The variable *range* splits the primary level ranges in an equal number of ranges. For example, if the value of *range* is 4 then there will be 16 segregated lists inside the provided size ranges. Similarly, if value of *range* is 5 then there will be 32 segregated lists inside the provided size ranges. If the value of *range* is 1 then the allocator accomplishes unallocated blocks as powerfully as the binary buddy allocator.

Here, the value of *range* is crucial to the performance of the allocator and hence, it is important to decide the minimum size of the memory block. If the value of *range* is big, it would cause more consumption of memory space to store the information like extra bits and pointers. Conversely, if the value of *range* is too small, then it would increase the internal fragmentation.

The index i denotes the existing maximum size of a memory block: $2^{i+1} - 1$, whereas the number of segregated lists in the provided sizes can be defined by the number of ranges: 2^{range}. Furthermore, a specific segregated list can be identified by the value of index I (i, j), which specifies whether the list (i, j) encompasses any unallocated blocks or not. Hence, all bitmaps do not comprise unallocated memory blocks, but they specify the probable availability of a particular

size of the memory block. All the pointers to the unallocated memory blocks are kept in a two-dimensional pointer array which is known as matrix.

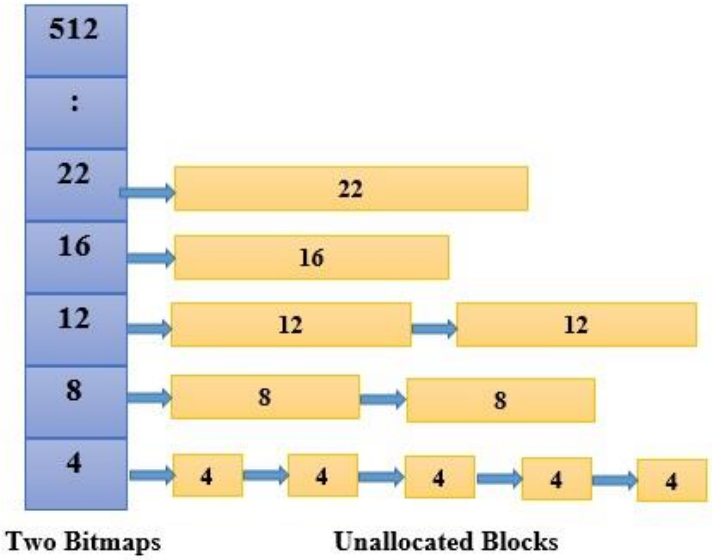

Figure 3.2: DmRT Structure for Small Block Allocation

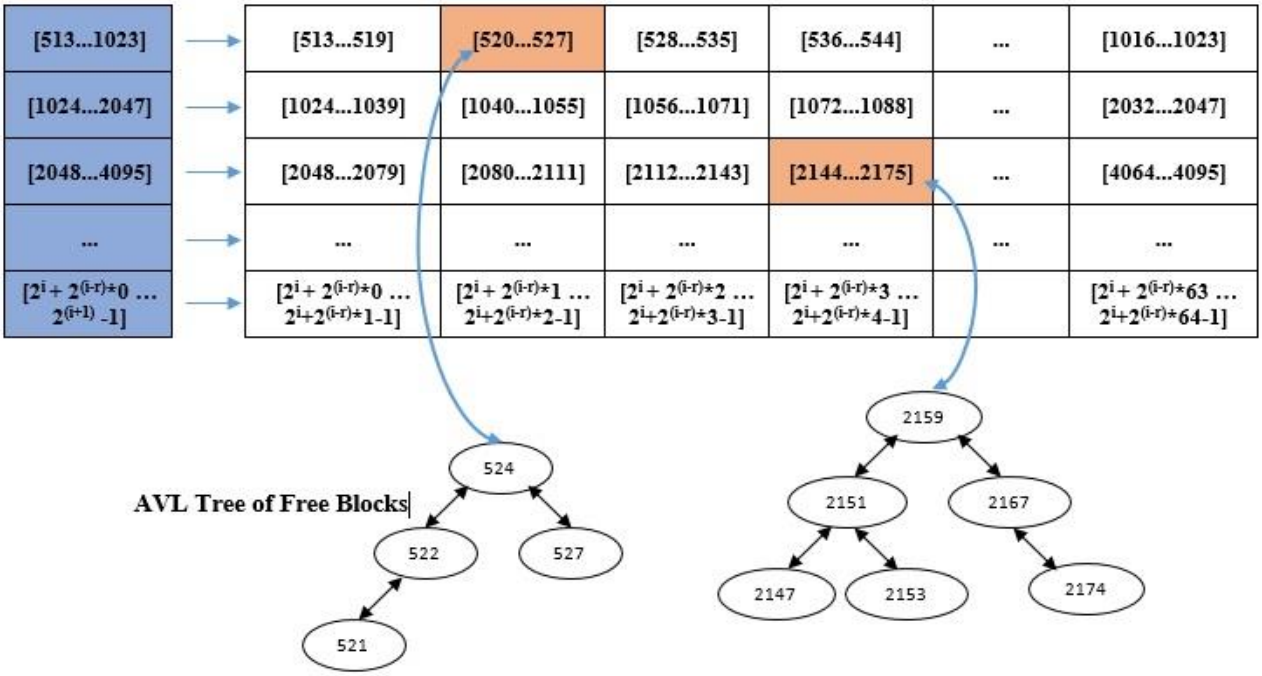

Figure 3.3: DmRT Structure for Normal Block Allocation

As discussed previously, for DmRT, the value of *range* is set to 6 by default. Every component of the array points to a list which has unallocated memory blocks of sizes, in a range, from $2^i + 2^{(i-range)} \times j$ to $2^i + 2^{(i-range)} \times (j+1) - 1$.

In the implementation of this allocator, it uses a two-dimensional bitmap array, which needs a 64-bit variable for the primary bitmap and 64*64-bit variables for the secondary bitmaps. Hence, total 66 variables of 64-bit are required to specify the unallocated block lists. Also, in each secondary level range, all the available memory blocks are arranged in the AVL tree to balance the tree structure.

$$\text{ISL (PI, SI)} = \begin{cases} PI = \lfloor \log_2 RB \rfloor & where\ PI \in [9,31] \\ SI = \left\lfloor \frac{(RB - 2^{PI})}{2^{PI-range}} \right\rfloor & where\ SI \in [0,63] \end{cases} \quad 3.1$$

The Primary Level is intended to accomplish the time of execution in a constant manner for the allocation of the memory blocks. Each and every segregated list keeps the specific size of unallocated memory blocks, and the DmRT can find an unallocated block by an index computed using equation 3.1. The Primary Level is designed using bitmaps and singular linked lists which contain small sizes of memory blocks. It has been designed using a bitmap, arrays of pointers to unallocated blocks and doubly-linked lists for the normal sizes of memory blocks. Having a single global heap between more than one thread may lead to the possibility of lock conflicts. To decrease the lock conflicts, each and every thread of the application should have a private thread heap.

3.2 Pseudocode of Proposed Allocator for SMP: DmRT

In this section pseudocode of proposed allocator DmRT has been shown. The first part described the pseudocode of "Arrangement of Blocks" and then pseudocode of "Allocation of different types of memory blocks."

3.2.1 Arrangement of Blocks

BEGIN

IF Block Size <= 512 bytes THEN

Hashing data structure where each key is multiple of 4 up to 512 bytes

At each key, link list of 64 nodes of same Size

ELSE IF Block Size > 512 bytes AND Block Size <= 2 Mb THEN

Create Two level list

Primary Index which Stores range of 2^{PI} to 2^{PI+1} -1 where $PI \in [9, 21]$

Each primary index is divided in ranges by 2^{range}, where ***range*** =6

ELSE IF Block Size > 2 Mb THEN

Block will be arranged in descending order of Size

ENDIF

END

3.2.2 Block Allocation

RB = Requested Block Size

PI = Primary Index

SI = Secondary Index

range = divides the Primary level ranges in a number of ranges linearly

ISL = Index of Segregated list which holds the Free block Tree

FR = Fragmentation

RS = Number of Request Satisfied (Initialize with 0)

RN = Root node of AVL Tree

BS = Block Size

MAB = Maximum Available Blocks

BEGIN

IF ***RB***<= 512 bytes THEN

$\boldsymbol{PI} = \left\lfloor \frac{RB-1}{4} \right\rfloor$

WHILE true

IF ***SI***> -1 THEN

CALL smallBlockAllocation(***PI, SI, RB***)

BREAK

ELSE

INCREMENT***PI***

IF ***PI*** EQUAL 9 AND ***SI*** EQUAL -1 THEN

PRINT "Block Allocation Failed"

RETURN

ENDIF

ENDIF

ENDWHILE

ELSE IF ***RB***> 512 b AND ***RB***<= 2 Mb THEN

$$\boldsymbol{ISL}\ (\boldsymbol{PI}, \boldsymbol{SI}) = \begin{cases} \boldsymbol{PI} = \lfloor \log_2 \boldsymbol{RB} \rfloor & where\ \boldsymbol{PI} \in [9, 21] \\ \boldsymbol{SI} = \left\lfloor \frac{(\boldsymbol{RB} - 2^{\boldsymbol{PI}})}{2^{\boldsymbol{PI-range}}} \right\rfloor & where\ \boldsymbol{SI} \in [0, 63] \end{cases}$$

```
                    CALL normalBlockAllocation(PI, SI, RB)
        ELSE
            Blocks are arranged in descending order of Size
            PI index starts with 0 up to MAB.
            CALL largeBlockAllocation(PI, RB)
        ENDIF
END

smallBlockAllocation (PI, SI, RB)
BEGIN
        PRINT "Small Block Allocated"
        Compute FR as (PI+1)*4 – RB
        DECREMENT SI
        INCREMENT RS
END

normalBlockAllocation (PI, SI, RB)
BEGIN
        WHILE true
            IF RN>= RB THEN
                Allocate RN;
                PRINT "Normal Block Allocated"
                BREAK
            ELSE
                SET RN as Right Child of RN
                IF RN reach to Leaf node AND RN<= RB THEN
                    INCREMENT SI
                    IF SI EQUAL 2^range -1 THEN
                        INCREMENT PI
                        IF PI EQUAL 21 AND SI EQUAL 2^range -1 THEN
```

```
                                        PRINT "Block Allocation Failed"
                                        RETURN
                                    ENDIF
                                ENDIF
                            ENDIF
                        ENDIF
                ENDWHILE
                Balance AVL tree to maintain level -1, 0, +1
                Compute FR as RN –RB
                INCREMENT RS
END

largeBlockAllocation(PI, RB)
BEGIN
        IF RB<= BS at PIth index THEN
                Divide block intoRB and (BS at PIth index - RB)
                Compute BS at PIth index as BS at PIth index - RB
        ELSE
                INCEREMENT PI
                IF PI>MAB THEN
                        PRINT  "Block Allocation Failed"
                        RETURN
                ENDIF
        ENDIF
END
```

3.3 Results

Case 1: Existing allocators and DmRT allocate from Local Memory

In a symmetric multiprocessor architecture, all processors will share the same memory which is known as local memory for them. Whenever any request for the memory block is raised, the memory manager will search and allocate memory block from the same local memory.

There are three different test categories have been selected.

1. Best case, i.e., the test has been taken for 100 memory blocks request.
2. Average case, i.e., the test has been taken for 1000 memory blocks request.
3. Worst case, i.e., the test has been taken for 2000 memory blocks request.

There are three main parameters considered for the results.

Parameter 1: The execution time should be consistent and minimum.

Parameter 2: Fragmentation should be as low as possible.

Parameter 3: Number of Requests Satisfied should be as high as possible.

Here, following four different memory management algorithms have been compared.

1. Dlmalloc
2. tcmalloc
3. TLSF
4. Proposed Memory Allocator

All the tests have been taken on MemSimRT simulator - A Memory Management Simulator for Real-Time operating system. The details about MemSimRT will be discussed in Chapter 5.

The results mentioned here is the average of 100 attempts. 100 attempts for each case have been mentioned in Annexure I.

3.3.1 Existing Allocators and DmRT allocate from Local Memory

1. Average of 100 attempts (Best Case: for 100 memory block requests)

Table 3.1: Existing Allocators and DmRT allocate from Local Memory (Best Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	DmRT
Execution Time (ms)	287.8581	330.3003	268.598	234.6128
Fragmentation in (%)	43.6472	29.684	22.4791	17.5031
Request Satisfied in (%)	56.6156	62.5883	81.5737	87.6169

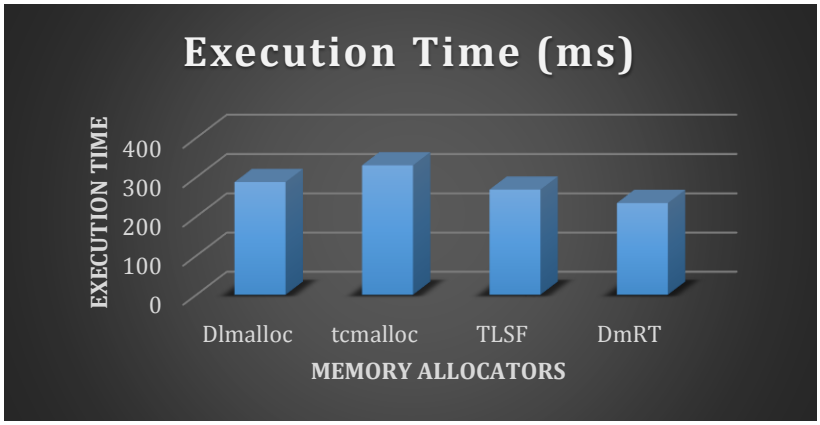

Figure 3.4: Execution time of Memory allocators in Best case

As shown in Figure 3.4, the DmRT takes **minimum execution time as** compared to all other dynamic memory allocators, and the tcmalloc takes maximum execution time.

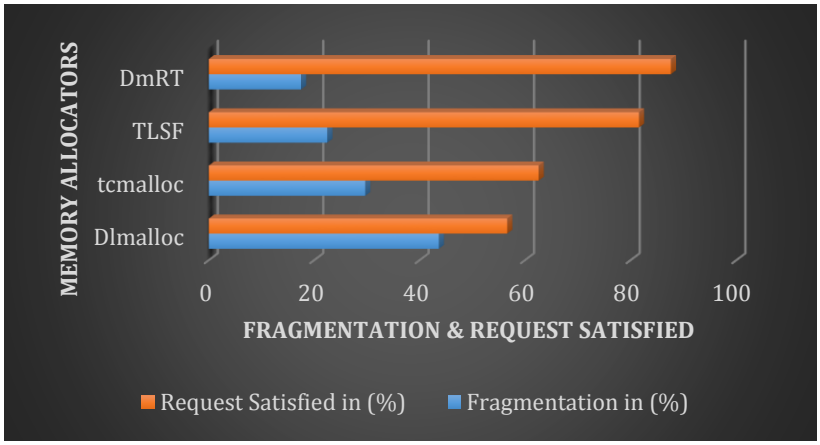

Figure 3.5: Fragmentation & Request Satisfied of Memory allocators in Best case

As shown in figure 3.5, the DmRT **satisfies the maximum requests** and has **lowest fragmentation as** compared to all other dynamic memory allocators, for the same, the Dlmalloc is exactly opposite to it.

2. Average of 100 attempts (Average Case: for 1000 memory block requests)

Table 3.2: Existing Allocators and DmRT allocate from Local Memory (Average Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	DmRT
Execution Time (ms)	1904.826	2890.503	1461.272	1067.995
Fragmentation in (%)	52.3926	35.157	27.0205	22.0902
Request Satisfied in (%)	45.458	57.4617	74.9894	83.109

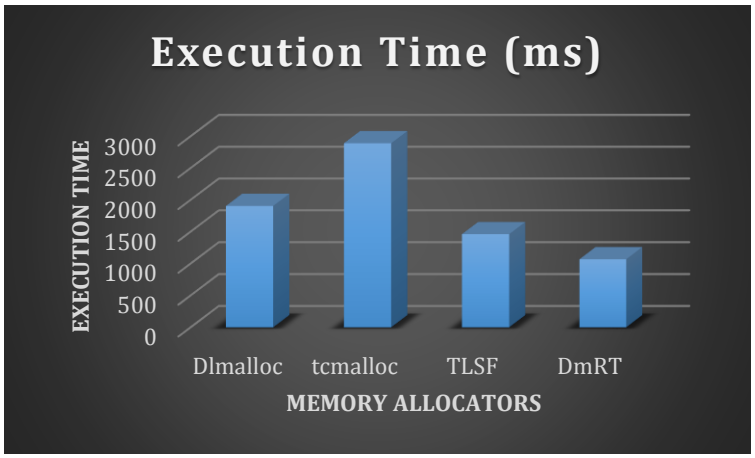

Figure 3.6: Execution time of Memory allocators in Average case

Figure 3.6 shows that the DmRT takes the **minimum execution time as** compared to all other dynamic memory allocators, whereas the tcmalloc takes the maximum execution time.

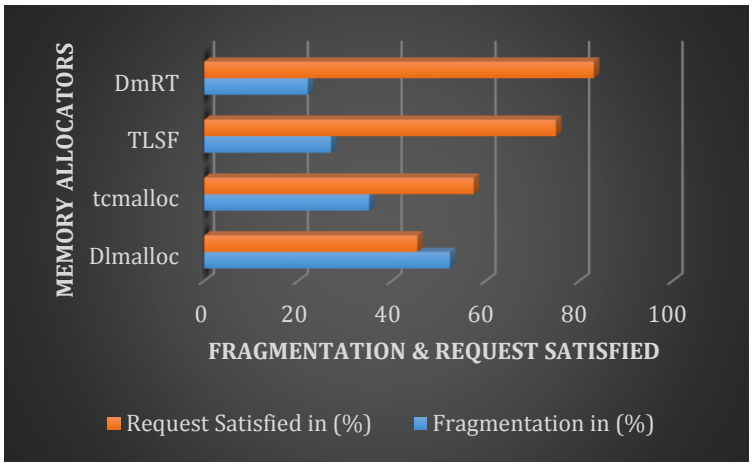

Figure 3.7: Fragmentation & Request Satisfied of Memory allocators in Average case

As shown in figure 3.7, the DmRT **satisfies the maximum requests** and has the **lowest fragmentation** as compared to all other dynamic memory allocators. Here also, the Dlmalloc is performing exactly opposite to it with **more fragmentation** and a non-acceptable number of requests being satisfied.

3. **Average of 100 attempts (Worst Case: for 2000 memory block requests)**

Table 3.1: Existing Allocators and DmRT allocate from Local Memory (Worst Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	DmRT
Execution Time (ms)	3204.577	4352.133	2153.912	1847.152
Fragmentation in (%)	60.4389	43.6719	32.0433	26.9948
Request Satisfied in (%)	35.0845	52.5575	70.5685	77.47

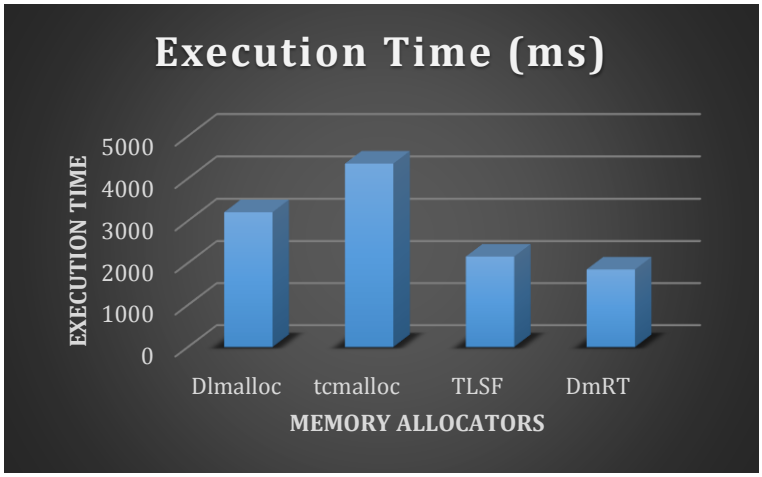

Figure 3.8: Execution time of Memory allocators in Worst case

In the worst-case, the DmRT takes **minimum execution time with reference** to all other dynamic memory allocators, while tcmalloc takes **maximum execution time. This scenario** is shown in Figure 3.8

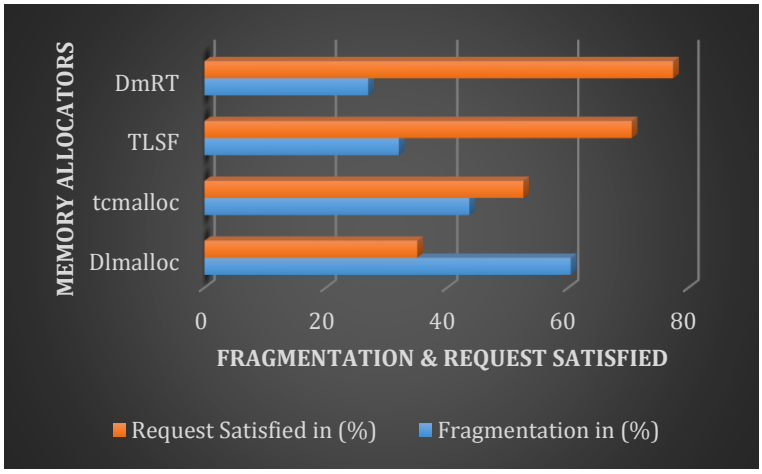

Figure 3.9: Fragmentation & Request Satisfied of Memory allocators in Worst case

Figure 3.9 shows that the DmRT **satisfies the maximum requests** and causes the **lowest fragmentation** among all other dynamic memory allocators. The notable thing here is that the difference among them is more than 50% i.e. even in the worst-case, the DmRT provides the best results and Dlmalloc performs exactly opposite to it.