Memory Management in Real-Time Operating System

A THESIS

Submitted in partial fulfillment of the

requirements for the award of the degree

of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE & ENGINEERING

By

SHAH VATSALKUMAR HASMUKHBHAI

FOTE/878

Guided By,

Dr. APURVA SHAH

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

FACULTY OF TECHNOLOGY & ENGINEERING

THE MAHARAJA SAYAJIRAO UNIVERSITY OF BARODA

VADODARA-390002 (INDIA)

AUGUST 2018

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING,
FACULTY OF TECHNOLOGY & ENGINEERING,
THE MAHARAJA SAYAJIRAO UNIVERSITY OF BARODA,
VADODARA - 390 001 (INDIA)
+91-265-2434188 (EXT. 405)

CERTIFICATE

This is to certify that **Mr. SHAH VATSALKUMAR HASMUKHBHAI** has worked under my guidance to prepare the thesis entitled **"*Memory Management in Real-Time Operating System*"** which is being submitted herewith towards the requirement for the degree of Doctor of Philosophy in Computer Science & Engineering, Faculty of Technology & Engineering, The Maharaja Sayajirao University of Baroda, Vadodara.

Dr. Apurva Shah,
Guide,
Department of CSE

Head,
Department of CSE

Dean,
Faculty of Technology & Engineering

APPROVAL SHEET

This thesis entitled **"*Memory Management in Real-Time Operating System*"** submitted by **Mr. SHAH VATSALKUMAR HASMUKHBHAI** in the Department of Computer Science & Engineering is hereby approved for the degree of Doctor of Philosophy in Computer Science & Engineering.

Examiner's Signature

Dr. Apurva Shah

Guide

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in the thesis entitled **"*Memory Management in Real-time Operating System*"** towards the partial fulfillment of the requirement for the award of the degree of **Doctor of Philosophy (Ph.D.)** in **Computer Science & Engineering** submitted in the Department of Computer Science & Engineering, Faculty of Technology & Engineering, The Maharaja Sayajirao University of Baroda, Vadodara (India) is an authentic record of my own work carried out during the period from May 2015 to June 2018, under the guidance of **Dr. Apurva Shah,** Associate Professor, Department of Computer Science & Engineering, Faculty of Technology & Engineering, and Director, Computer Centre, The Maharaja Sayajirao University of Baroda, Vadodara (India).

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other Institute.

Date:

Place: Vadodara **(SHAH VATSALKUMAR HASMUKHBHAI)**

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date:

Place: Vadodara **(DR. APURVA SHAH)**

Associate Professor,
Department of Computer Science & Engineering,
Faculty of Technology & Engineering,
The Maharaja Sayajirao University of Baroda, Vadodara

ABSTRACT

The memory allocation algorithms have been analyzed and worked upon broadly, but there is less attention given to the multiprocessor architecture and real-time operating system as well. Most of the algorithms are applicable for the general-purpose operating system and do not fulfill the necessities of real-time systems. Moreover, limited allocators designed to support real-time systems which are not completely scalable for multiprocessors. In the 21st century, as we have entered into an era of high-performance computing, the demand for multi-core architecture has gained momentum. NUMA architecture based systems are the outcome of this tendency and offer an organized scalable design. However, existing dynamic memory allocators are not capable of performing on a multiprocessor architecture and do not comply with real-time system requirements as well. Researches have proved that the existing memory allocators for any operating systems which support NUMA architecture are not suitable for real-time applications. Hence, there is a need to have a dynamic memory allocator which can perform well on SMP and NUMA based soft real-time systems, with better execution time and less fragmentation.

This research is carried out in the same direction to achieve the aforementioned goal of a dynamic memory allocator for real-time systems. 1). Dynamic memory allocator **DmRT** has been designed and implemented for symmetric multiprocessing system which provide consistent and optimum execution time, less memory fragmentation as well as satisfying maximum number of memory request with compare to other existing allocator. 2). As per the need of high performance computing, a dynamic memory allocator **DmRT** for NUMA architecture based real-time operating system has been designed and implemented which also provide consistent and optimum execution time, less memory fragmentation as well as satisfying maximum number of memory request. 3). There are so many simulators available to simulate different test cases for scheduling in a real-time operating system like Litmus-RT, Mark3, rtsim, etc., but till date, none of the simulators is available for simulating memory management algorithm for RTOS. Hence, **MemSimRT** has been designed to simulate various memory allocators for both SMP as well as NUMA architecture based RTOS.

ACKNOWLEDGEMENTS

As a firm believer in the powers and blessings of the **ALMIGHTY GOD**, I firmly believe that it is **Lord Krishna** who has done this gigantic task as he is **'karta- the doer'** of everything and whose wishes run supreme to any action or happening in the known and unknown spheres of universe. I certainly feel blessed that my sincere, often heart touching and tear-some prayers have been answered by his presence which I feel was more pertinent during my long hours of work. I personally attribute it as a single point source from whom I have drawn inspiration from.

In pursuing my Ph.D., I have received contributions, help and support from a number of friends, associates, researchers, and my dear students. I would express my sincere gratitude and appreciation to all of them.

I convey my sincere thanks **Dr. Apurva Shah**, my research supervisor for giving me an opportunity to take up the research work under his supervision, helping me in all phases of the research work, providing sound encouragement and guidance throughout this journey. I with all sincerity bow and express my gratitude for his scholarly, inspiring, incessant, painstaking efforts and patronage. As my supervisor and mentor, **Dr. Apurva Shah** has really provided thoughtful and constructive feedback in improving my research work and the research thesis. In spite of his many responsibilities and other commitments, he has helped me at all the odd times to accomplish this research successfully. I am extremely thankful to him, for the technical guidance, help and conveniences provided to me for successful completion of my thesis and research work.

I would like to thank **BVM Family** and **MSU Team** for giving me the opportunity, facilities and inspiration to complete this thesis on time. Sincere thanks to the executives and management of BVM, CVM and MSU who placed their faith in my ability to deliver innovative research and who stood with me, when I needed them.

I express my special thanks to **Dr. Anjali Jivani,** Head and Associate Professor, CSE Department, Faculty of Technology & Engineering, The Maharaja Sayajirao University of Baroda, Vadodara and **Dr. Viral Kapadia,** Assistant Professor, CSE Department, Faculty of Technology & Engineering, The Maharaja Sayajirao University of Baroda, Vadodara for positively criticizing the work and sparing his time by giving valuable comments and suggestions for the betterment of

this work. I also thank **Prof. Vishwas Raval and Dr. Amit Thakkar** for providing me with timely inputs and helping me overcomes the teething problems.

"The essence of Bhagavat Gita lies in man's devotion to work and service to mankind. If this is ensured, rest all follows". I found it true in **Naineshbhai Sabeb**. His selfless help has not only removed many obstacles from my path but has also given new meaning to my life as a whole. I with full respects accept his love, blessings and best wishes.

My personal thanks to **Mr. Divyang Thakkar** who always helped when I fall in depression**,** as well as my dear friends **Rahul Gordiya and Dhaval Tailor** who all were ready to render me every type of help required at different stages of my research work.

I fell indebted to my respected parents for providing their continuous inspiration and motivation to complete this research work. I certainly feel that person's first Guru is his/her Mother. And it was her blessings only that kept me going. I take this opportunity to pay my heartfelt tributes to my father who has always supported and applauded my academic efforts constantly.

My beloved wife **Ms. Jayna** has been helpful and considerate to me throughout my research work. "Children are innocent and Innocence is loved by Lord". I have felt the divine presence of the almighty whenever my young nephew **Vraj** has prayed for my success and goodwill.

Sentiments from the core of the heart, when translated into words, seldom convey what one truly wishes to express for prayers, encouragement and support provided to me by all the **Saints and Devotees of my Lord**. No formal words can fully convey my thanks to them.

I am highly obliged to all the persons who helped me **visibly or invisibly** for the successful completion of this Ph.D. work.

VATSALKUMAR HASMUKHBHAI SHAH

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

1.1 Introduction to Real-Time Operating System

In our daily life, embedded systems have become an essential part of our life. It can be a mobile phone, a smartcard, a music player, a router or any other electronics devices which are rapidly changing our lives. An embedded system is defined as a group of several devices or parts such as computer software, hardware, and extra mechanical components intended to accomplish a specific function on the fly on time basis. Due to time constraint, they are also called real-time systems. For that, the real-time operating system (RTOS) becomes the foundation of the Embedded Systems. These real-time systems are lacking in memory and processing power. For this reason, memory management is essential and has been a topic of research since the inception of real-time systems.

To define a real-time operating system, **it is a type of an operating system which provides support to the real-time applications by giving an accurate result within a specific time duration. This time duration is called deadline. [11]**

Table 1.1: Difference between General Purpose OS and RTOS

	RTOS	**General Purpose OS**
Determinism	Deterministic	Non-deterministic
Preemptive kernel	All kernel operations are preempt-able	Not Necessary
Priority Inversion	Have mechanisms to prevent priority inversion	No such mechanism is present
Task Scheduling	Scheduling is time-based	Scheduling is process based
Application	Typically used for embedded applications	Generally used for desktop PC or other general purpose PCs

The table 1.1 shows the fundamental dissimilarities between the general purpose operating system and a real-time operating system [86]. Differentiation is prepared based on various characteristics like determinism, kernel, priority inversion, job scheduling and latency. Real-time Operating Systems have implemented in such a way that the scheduling of events is extremely deterministic. A real-time system may have one or multiple real-time tasks. The fundamental dissimilarity between a real-time and non-real-time task is that a real-time task is always considered by the time limit which informs the system that the given task must be finished within the specified time.

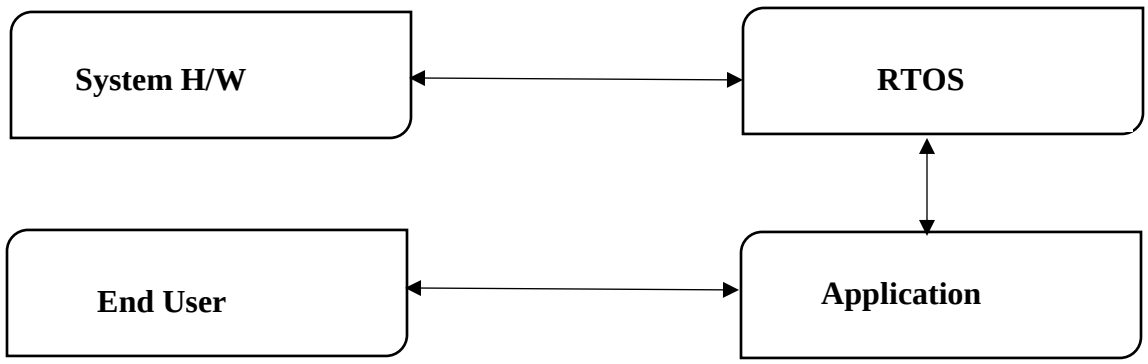

Figure 1.1: Overview of embedded system for real-time applications [21]

The real-time system can be categorized into three different categories on the basis of its criticality of meeting the deadline [44]:

1) Hard real-time systems: A real-time task/system is considered to be hard, if not generating the outcomes within the given deadline may create terrible significances on the system under control. For example, automotive systems, nuclear-plant controllers etc.
2) Firm real-time systems: A real-time task/system is considered to be firm, if generating the outcomes after its deadline is of no use for the system, but it may not create any destruction or catastrophes. For example, railway ticket reservation system.
3) Soft real-time system: A real-time task/system is considered to be soft if the outcome of the system is still useful even after missing the deadline with slight performance degradation. For example, multimedia applications on the mobile phone.

1.2 Features of RTOS

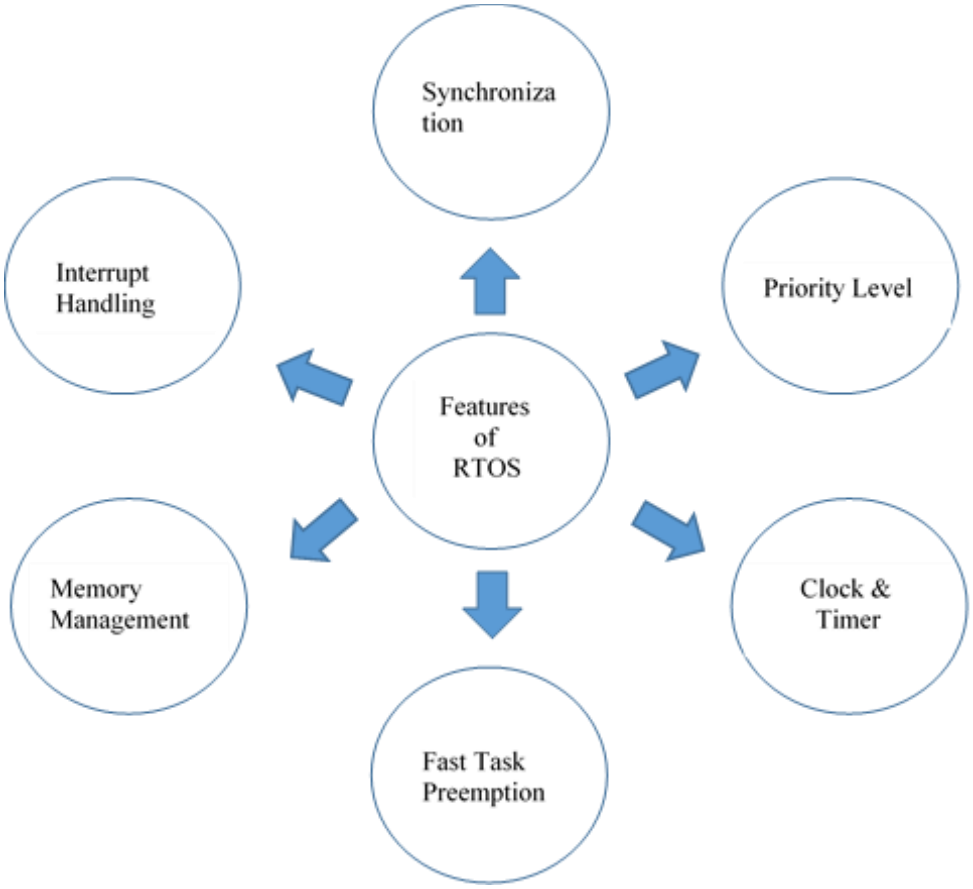

Figure 1.2: Features of RTOS [21] [44]

- **Synchronization:**

  Synchronization is essential for tasks of the real-time system to share mutually exclusive resources. Here, predictable inter-task communication and synchronization strategies are mandatory for multithreading communication in an appropriate manner [27] [61].

- **Interrupt Handling:**

  For handling interrupt, an Interrupt Service Routine is required. Here, interrupt latency is considered as the time duration between an interrupt generation and the execution of the related Interrupt Service Routine [36].

- **Timer and clock:**

  The Clock and timer services with suitable determination are important aspects of each real-time operating system.

- **Real-Time Priority Levels:**

  Each real-time operating system must have the provision of real-time priority levels which are allocated by the developers and the operating system cannot modify it [33].

- **Fast Task Preemption:**

  For the effective procedure of a real-time application, whenever a high priority critical task comes to the critical region, and if a running task has low priority, then the high priority task should be allocated to the CPU immediately.

- **Memory Management:**

  Real-time operating systems for huge and standard sized applications are predictable to offer virtual memory [25]. They not only have to achieve the demands of memory but to also have to serve the memory requests of non-real-time applications as well like different types of editors, browsers, etc. A Real-time Operating System usually has a small memory size with only essential features for an application [42] [43].

1.3 Memory Management

There are two types of memory management carried out in real-time operating system using a stack and using a heap. At the time of context switching of tasks, the stack management is used whereas for dynamically allocating memory to the tasks, the heap management is used [77] [97].

The memory management of the real-time operating system is categorized as static memory management and dynamic memory management [92]. Figure 1.3 shows the classification of Memory Management Technique.

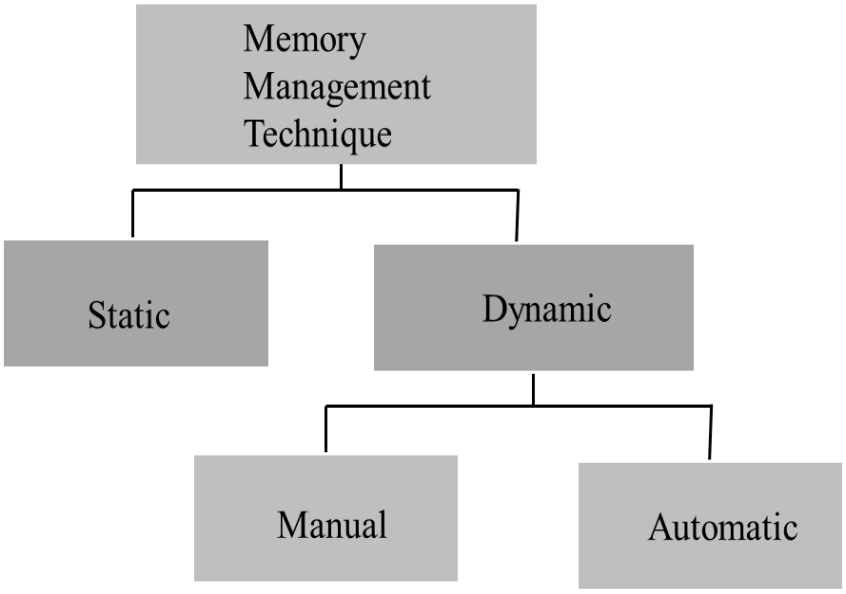

Figure 1.3: Classification of Memory Management Technique [92]

The following table Table1.2 [21] [82] shows the fundamental differences between static and dynamic memory management.

Table 1.2: The Fundamental difference between static and dynamic memory management

	Static Memory management	Dynamic Memory Management
1	Memory allocation is done at compile or design time.	Memory allocation is done at runtime or during execution.
2	Static memory allocation is a fix process which means requisite memory for a specific process is already identified, and after allocating memory, no modifications can be done during the execution.	Dynamic memory allocation needs memory manager to maintain which portion of the memory is allocated and which one is unallocated. So, when a process requests memory, it can allocate and deallocate memory whenever needed.
3	Allocation and deallocation of memory are not performed during execution.	Memory bindings are established and demolished during execution.
4	Extra memory space required.	Less memory space required.

Further, Dynamic memory management is classified into two categories, Manual and automatic [21] [92].

1. Manual memory management

In manual memory management, the developer has direct control over the memory allocation and deallocation. Typically, this is carried out either by heap management functions or by the language constructs that affect the stack. The advantage of manual memory management is it is simple for developers to comprehend and use. However, the disadvantage is that memory management errors are very ordinary.

2. Automatic memory management

In Automatic memory management, the task of memory manager is to manage memory by reusing the memory blocks which are out of scope from the program objects. These automatic memory allocators are also called garbage collectors. The benefit of using it is it removes the majority of the memory management errors. The demerit is many times it misses the deallocation of unusable memory blocks which may lead to memory leakage problem. It also consumes a large amount of the processor time and generally has non-deterministic performance.

1.4 Problem Statement

To design and develop a dynamic memory allocator for real-time operating systems which can reduce the memory fragmentation and satisfy the maximum number of memory block request in the consistent execution time.

1.5 Motivation

The majority of the operating systems use dynamic memory allocation for processing which requires communicating explicitly with memory allocator component. Though memory allocation algorithms have been analyzed and worked upon broadly since 1960, it has been observed that there is less attention given to the multiprocessor architecture and real-time operating system as well. Most of the algorithms have been designed such that, they are applicable for the general-purpose operating system and do not fulfill the necessities of the real-time systems [37].

Moreover, the limited allocators have been designed to support real-time systems but they are not completely scalable for the multiprocessors. As the 21st century is an era of high-performance computing, the demand for multi-core architecture has gained momentum. NUMA architecture based systems are the outcome of this tendency and offer an organized scalable design [50] [62]. However, these dynamic memory allocators are not capable of performing on a multiprocessor architecture and do not comply with real-time system requirements as well. Researchers have proved that the existing memory allocators for any operating systems which support NUMA architecture are not suitable for the real-time applications [71] [81]. Hence, there is a need to have a dynamic memory allocator which can perform well on SMP and NUMA based soft real-time systems, with better execution time and less fragmentation. This research is carried out in the same direction to achieve the aforementioned goal.

Memory is an important aspect when implementing real-time applications because of its costly management in terms of time and memory. Therefore, worst-case execution time and utilization of memory are the prime aspects of real-time system designers. To increase the performance of the real-time system, a cost-effective memory manager is required [79] [84]. As a result, the majority of the real-time systems use static memory allocator for random allocation and de-allocation of memory blocks. It means the allocation of memory is done at the compilation time or during initialization of program before the application arrives into core real-time stage where the entire physical memory is accessible as a single block and can be used as per requirement. Due to the unavailability of automatic memory management, an issue regarding excessive use of memory space arises as the memory which is used for keeping objects cannot be simply reclaimed. The developers have to employ and maintain their private pools of memory to decrease the excessive usage of memory space and reclaim the memory space. The real-time systems have been growing in size and complexity with the multi-core and multiprocessor architecture based systems and hence they require the flexible use of the existing resources comprising memory as well [107]. The static memory and pool of memory are not applicable in the era of the multiprocessor system and the implementation of real-time applications will gradually have to use dynamic memory management allocator to achieve the predictable performance and flexibility [95].

Dynamic memory allocator has been the most significant and essential part in the general-purpose software field because it is more proficient and flexible than the static memory allocator.

Since last five decades, the research scope of the dynamic memory allocation algorithms has been analyzed for its significance and attractiveness. Earlier, a considerable amount of research work has been carried out on the issue of the good average response time of Dynamic Memory Allocation algorithms to check their speed and efficiency in allocation and deallocation of memory blocks. Along with the ways to decrease fragmentation in memory, in case of dynamic memory allocators, there are numerous faster and competent algorithms available for the general purpose operating system, for instance, DLmalloc [53], tcmalloc [99] and Hoard [6]. However, the worst-case execution time (WCET) of these algorithms has not been analyzed thoroughly. For real-time systems, no significant analysis has been carried out for dynamic memory allocation algorithms. Due to this, the majority of the application designers of real-time systems usually ignore the dynamic memory allocators. The reason why the designers are worried is that the worst-case execution time of dynamic memory allocator routines is not constrained [88] [98]. As the lifespan of a real-time application is typically longer than the lifespan of a general-purpose application, not only WCET of the dynamic storage algorithm but also memory utilization efficiency should be considered for the algorithms [106]. Throughout the long lifespan of the application, dynamic memory allocator creates holes in memory, which cannot be reclaimed because of their small size. These holes result in slow offensive response time or they lead to failure in meeting the deadlines. Such holes are called memory fragmentation.

1.6 Objectives of Memory Management Algorithm

The research in dynamic memory management for real-time systems is one of the areas which is still unconquered because real-time applications impose different requirements on memory allocators than the general-purpose applications. The most significant requirement in real-time systems is the investigation of scheduling which is used to decide if the response time of real-time application can be bound to fulfill the timing restriction of the execution. This investigation should consider the impression of multiprocessor architecture settings like concurrency, lock contention, cache misses and traffic on the bus. The effects of all these problems on NUMA architecture systems, related to dynamic memory management could be described as follows:

1) **Reduce memory fragmentation**: As stated above that the period of real-time applications is normally longer than that of simple applications. It may be running from one day or one month while the period of simple application is very short. During the lifespan of a real-time application, the systems may randomly release memory segments of any size, which may cause holes (free memory block) in the memory [7]. In addition to this, these holes are used to be of very small size and hence they cannot be reused. For this, reducing fragmentation of memory can be considered as the main objective.

2) **Restricted execution time**: Applications of real-time systems should meet their timing criteria. For this, the dynamic memory management algorithm should be designed in such a way so that the application can be completed within the given deadline.

3) **Increase node-based locality**: In NUMA based architectures; the memory is shared in the system so that its processors can access remote memory. Here, the time required to access memory will be high as compared to access the local memory. For this reason, it would cause performance degradation of the system. Hence, the node-based locality which tries to access the local memory is the main features to improve the system performance [35] [47].

4) **Reduce false sharing**: Due to real-time systems are cache coherent in nature [72] [78], it introduces "False sharing". When several processors try to access different data objects inside similar cache or memory block, this problem may arise [8]. This false sharing may lead to performance degradation of the application when working on multiprocessor shared memory architecture.

5) **Reduce memory access to the remote nodes**: The time consumption for accessing a remote node is very high [100] which is almost twice than for the simple one, the application developer need to consider this point and develop it in such a way that it can reduce the memory access of remote node.

6) **Reduce lock conflicts**: Applications which are executing on the NUMA based systems and which run concurrently cause the problem of lock conflicts. This issue should be addressed to maintain the performance of the system.

1.7 Research Contributions

For last four-five decades, the majority of the operating systems have started using dynamic memory allocation algorithm. In this, every time the memory allocation needs explicit communication with an allocator. The memory allocation algorithms have been analyzed broadly since 1960, but inadequate devotion has been given to the real-time properties. Majority of the algorithms are for the general-purpose operating system and do not fulfill the requirements of real-time systems. Moreover, the limited allocators which provide support to real-time systems are not properly scalable for multiprocessors. The demand for high-performance computational processing has been increased gradually which lead to development of multi-cores. NUMA architecture based systems are a portion of this tendency and offer an organized scalable design. In summary, there are a few dynamic memory allocators are available, but they do not perform properly on multiprocessor architecture and do not satisfy real-time system requirement as well. The existing memory allocators for any operating systems which support NUMA architecture are not suitable for the real-time applications. So there is a requirement of a dynamic storage allocator which can perform well on both SMP as well as NUMA based soft real-time systems. This allocator must give better execution time and less fragmentation. Through this research, the following research contributions have been made.

1. A Dynamic memory allocator **DmRT** has been designed and implemented for Symmetric Multiprocessor System which provides consistent and optimum execution time along with less memory fragmentation and satisfies maximum number of memory requests as compared to the other existing allocators.
2. A dynamic memory allocator **DmRT** for NUMA (Non-Uniform Memory Access) architecture based real-time operating system has also been designed and implemented. It provides consistent and optimum execution time, less memory fragmentation and it also serves a maximum number of the memory requests.

3. There are so many simulators available to simulate different test cases for scheduling in a real-time operating system, e.g. Litmus-RT, Mark3, rtsim, etc., but no simulator is available for simulating memory management algorithm for RTOS so far. Therefore, MemSimRT has been designed to simulate various memory allocators for both SMP as well as NUMA architecture based RTOS. This will be useful for all existing allocators as well as the allocators which would be proposed in future. The MemSimRT can be downloaded using the following QRcode

1.8 Applications

The main requirement of any real-time operating system is to finish the task within the given deadline. If it misses the deadline then either it would lead to catastrophic events or may lead to failure in performance. To accomplish this criterion, it is mandatory that the task should have resources in time whenever it requires. Here, resources may be anything CPU, Input/Output or memory etc.

To execute a process it must have CPU in time as and when needed, but at the same time, it should have an on-demand memory as well. To provide memory on time, the strong and perfect memory allocator will help the task to achieve the deadline. Hence, a memory allocator is designed which can run on Symmetric Multiprocessor (SMP) architecture as well as Non-uniform Memory access (NUMA) architecture based real-time operating system.

1.9 Organization of Thesis

The remaining chapters of the thesis are organized into six chapters as given below:

Chapter 2. Literature Review: This chapter explores the dynamic storage algorithm models provided by general-purpose and real-time systems, including their policies and mechanisms. Furthermore, the conventional memory management algorithms have also been investigated, with specific emphasis on managing the memory blocks. This chapter also highlights hybrid memory

management algorithms which perform better on multiprocessor architecture systems as compared to the conventional dynamic storage allocators.

Chapter 3. DmRT for SMP: This chapter focuses on providing and implementing a more efficient dynamic memory management algorithm for SMP architecture. Also it includes the results of this allocator and compares it with the existing allocators.

Chapter 4. DmRT for NUMA: This chapter discusses an efficient and dynamic memory management algorithm for NUMA architecture especially with its results analysis in context of existing allocators.

Chapter 5. Memory Management Simulator: MemSimRT: This chapter introduces a simulator of memory management algorithm which has been designed for the real-time operating system.

Chapter 6. Conclusions and Future Work: The conclusions derived based on the results of this research work are given in this chapter. Additionally, some directions for further possible research are also presented.

Chapter 2
Literature Review

2.1 Dynamic Memory Management

It is one of the significant procedures in the present world of engineering to accomplish tasks generated during execution of programme in any of the high-level programming. It is about to arrange unallocated or allocated memory blocks which have a smaller life scope than their parent task, job or process. It is highly challenging to fulfill the criteria of timing restriction of real-time applications. The reason behind this is any real-time application requires the prediction of worst-case execution time (WCET) in dynamic memory management [96]. Furthermore, searching the ideal location for allocating a memory block is an NP-hard problem if certain memory blocks are previously allocated [30] [38] [94]; and if a memory management algorithm cannot satisfy the request of memory block, it will lead to fragmentation, though the total size of available memory might be more than the requested block of memory size.

If dynamic memory management is used on multiprocessor architecture, some new complications are raised like false sharing, synchronization between threads. The reason behind this is that the needs of real-time applications vary from one architecture to another [31] [32].

In this section, the above issues will be explained based on the research work that has been carried out so far along with the literature review of memory management algorithms. Section 2.2 introduces some essential problems in memory management like fragmentation, false sharing and heap [93]. In Section 2.3, a range of traditional and nontraditional memory management algorithms have been discussed.

2.1.1 Jargons

In 1995, Wilson [116] and his team have given certain terms or jargons that are commonly used in the area of memory management. The following text discusses a few jargons used like Strategy, Policy and Mechanism.

1) Strategy:

A strategy is a fundamental methodology used in any memory allocation algorithm. It considers different configurations in the program and defines a variety of suitable rules for memory blocks allocation dynamically. The goals of an allocation algorithm can be described as being a correspondent in significance to the allocation algorithm's strategy, For example, "reducing lock contentions", "increasing data locality" etc. These all strategies can be accomplished by policies.

2) Policy:

A policy is a conclusion method for allocating blocks of memory dynamically. It decides precisely from where an allocated block will be removed or at what place in memory an unallocated block will be put in. For example, one policy defines like: "each time discover the minimum block of memory which is large enough to fulfill the memory request". These selected policies are employed by different mechanisms. Some policies are Best-Fit, Exact-Fit, First-Fit, Next-Fit, Good-Fit, and Worst-Fit.

3) Mechanism:

A mechanism is nothing but the implantation of policy. It is a group of different algorithms and data structures. For example, "use a singly linked-list and find the location of unallocated memory block list from where the previous request was fulfilled; the unallocated blocks are inserted at the end of the singly linked-list". Normally, the mechanism may be defined as Sequential Fit, Segregated Fit, Buddy Systems, Indexed Fit and Bitmapped Fit [64] [67].

These all jargons and their definitions are significant for accepting and planning a dynamic memory management algorithm. For instance, for a specific strategy, various policies can cause diverse effects. If a policy creates better locality with huge fragmentation, an application designer has to select an alternative policy under the similar strategy, which can generate low fragmentation. Any policy can be employed by a variety of mechanisms. If a specific policy achieves a good result, but its implementation is not well-organized, designers can employ other policy by selecting an altered or alternative mechanism [101]. Hypothetically, low fragmentation is one of the major characteristics of any dynamic memory allocator which can be achieved by a robust allocation policy. The allocation policy is the selection of memory block from the unallocated memory block list. It can be accomplished by two methods [44] [89] [109]: a) Splitting b) Merging

a) Splitting:

In this allocation policy, large size memory block will be split into a number of small size memory blocks, and it will use a large separated memory block to fulfill a specific memory block request. Normally, the remaining memory blocks will be traced as unallocated memory blocks, and will be used to fulfill upcoming memory block requests [49].

b) Merging:

Merging is used when any applications, jobs or tasks release allocated memory blocks. When applications release any memory blocks, a virtual component called as memory manager finds whether any neighboring memory blocks are released, unallocated or not. And if they are released then it coalesces these memory blocks into a single large size memory block [49]. This is required as a large size memory block is more convenient than two small-sized memory blocks.

Merging process can be divided into two different classes. Primarily, immediate merging tries to merge the unallocated memory blocks instantly whenever a block is released. However, immediate merging is costly because any released memory block will be merged together by continually and regularly coalescing the neighboring unallocated memory blocks. Secondary, deferred merging simply notes a released memory block as “unallocated” or “released” without combining. Because most of the applications frequently generate similar size memory blocks of short life span this type of memory management algorithm maintains memory blocks of a specific size on an unallocated list and reclaims them without merging and splitting. Due to this, if an application demands the similar sized block of memory immediately after one memory block is freed, the memory block request can be fulfilled by simple manipulation; this may enhance if some specific size of memory blocks are frequently allocated and unallocated. But the disadvantage of differed merging technique is that it has unbounded response time which causes fragmentation.

2.2 Fundamental Issues

To design any memory allocator, two important things must be considered, first is the time efficiency and second is the space efficiency. Without making a negotiation between these two, it is hardly possible to develop a memory allocator which is having low fragmentation and low response time for most of the applications [17] [22]. For example, the memory allocator designed

by Kingsley [48] is the best example of simple segregated storage algorithms. In this allocator, each request of the memory block is converted into a nearest large number which is a power of two. In its allocation phase, it tries to remove memory block from the array, and in deallocation phase, it tries to insert memory block into an array. Due to the simple design of this algorithm, its performance is very efficient and fast and it doesn't try to merge the adjacent memory blocks. Hence, the internal fragmentation is very high. So, an important criterion to design a dynamic memory allocator is to maintain a balance between time and space efficiency.

For memory allocators of multiprocessor architecture, several other problems arise like false sharing, heap conflicts, boundless increasing heap problems and conventional fragmentation.

2.2.1 Fragmentation

Among the various issues, the most critical problem of a memory allocation algorithm is memory fragmentation. Randell had categorized fragmentation in two classes [90]: internal and external. These are generated due to splitting and merging of memory blocks as explained earlier.

1) External Fragmentation

External Fragment will be generated when the demanded block of memory cannot be served even though the total unallocated memory is more than demanded memory block size. While allocating and deallocating a memory block, external fragmentation is created due to a large number of small-sized unallocated memory blocks which are known as 'holes'. The large number of small size unallocated memory blocks are not adjacent to each other, hence cannot be combined, and they are too minor to fulfill any demand.

2) Internal Fragmentation

Internal fragmentation is generated when the memory management algorithm provides a large-sized unallocated memory block to fulfill the demand, instead of the actual demanded block size. The remaining part of the allocated memory block will have no use. That's why this scenario is known as internal fragmentation. The remaining memory block remains included in an assigned memory block. In certain memory allocation algorithms, internal fragmentation is allowed up to a certain limit for improved performance or easiness. In contrast to external fragmentation, internal fragmentation is exceptional to the respective implementation of a memory management algorithm

and so it must be analyzed according to the application. Hence, it is difficult to discover a common analysis of internal fragmentation anywhere.

2.2.1.1 Fragmentation Measurement Parameters

For comparison of memory management algorithms, some unit of measure is required. Normally, the cost of time and space are used as a unit of measure. The cost of time means the execution time of process and cost of space means fragmentation in memory. Through various ways, the fragmentation in memory can be demonstrated. For example, there are 15 unallocated memory blocks of size 8Kb and 20 unallocated memory blocks of size 2Kb, and if an application demands 10 unallocated memory blocks of size 8Kb and 15 unallocated memory blocks of size 2Kb. In this situation, there should not be any fragmentation as the demanded memory blocks are available but if some application requires 15 unallocated memory blocks of size 16Kb then it would create a problem and demand will not satisfied because of the huge amount of fragmentation.

In his work, Johnstone [46] proposed four different unit of measure to define the amount of fragmentation.

Method 1: The fragmentation can be measured as the total memory usage by the memory allocator over the amount of memory demanded by the application, aggregated at all points over the time. This method of fragmentation measurement is quite simple but this method hides the spikes in the utilization of memory and due to these spikes fragmentation may create problems.

Method 2: The fragmentation can be measured as the total memory usage by the memory allocator above the highest memory needed by the application at the point of highest utilization of memory. The issue with this method is that the point of highest utilization of memory cannot normally be measured at runtime.

Method 3: The fragmentation can be measured as the highest amount of memory used by the memory allocator over the actual memory needed by the application at the time of maximum memory used by the memory allocator. The problem with this method is that

it will create high fragmentation, even if the applications use small amount of memory more than the size of required memory.

Method 4: The fragmentation can be measured as the highest amount of memory usage by the memory allocator over the highest amount of memory usage by the application. The drawback of this method is that when an application uses small memory, it can show low fragmentation.

Each method stated above is used for measurement of fragmentation generated by the application. The issue with these all measures is that none of the methods differentiated between unallocated memory and allocated memory for its self-arrangement of data, such as maintaining unallocated blocks. In this research, the internal fragmentation and external fragmentation have been considered with space used by data structures. The following equation [46] has been used to compute the exact amount of fragmentation (*frag*):

$$frag = \frac{h-a}{h} \qquad 2.1$$

Where, h is the actual amount of memory provided by the allocator, and points out the amount of allocated memory requested by the application.

2.2.2 False Sharing

The application which includes multiple threads has so many challenges like race conditions between threads, different types of conflict such as contentions, debugging of threads and the most obvious problem, the deadlock conditions. But uniprocessor system cannot cop up in the modern and faster trends of multiprocessor environment. So, memory management algorithm must be smart enough to resolve the issues created due to simultaneous execution of threads which are demanding different or same memory block at the same time.

Sharing resources between parallel executing thread is the most obvious condition in any application with multithreading. Now, the conflict between threads always happens when two or more thread try to access shared resources. But this is not the only reason when the conflict arises, it may arise when all threads try to read or write different resources if multiple resources are

sharing same memory or cache. For example, thread1 modifies one object, say dummy_ob1, and at the same time, thread2 modifies another object dummy_ob2. Let us consider that these objects dummy_ob1 and dummy_ob2 exist in the same cache memory but the threads are executing on two different processors. If of these two objects, one is updated, then cache memory will be put in invalidate condition due to cache coherent rules and because of this situation, the performance of the application will be degraded [40]. This scenario is known as false sharing.

Due to the growing needs of a multi-processor system, the tendency of adding more cache memory size leads to the problem of false sharing [57]. It is also hard to remove false sharing automatically. To address this either the compilers adjust some logic of simultaneous looping or designer should use such data structure to avoid this problem [45]. This technique may break the association between objects of data and false sharing. Though the issue cannot be entirely removed as there is some impact of data structure which is based on an array; still, a better-planned approach can address this issue [6].

2.2.3 Types of Heap

The memory used by the memory allocator for allocation of a block is called heap. In a multiprocessor environment, a single heap will not be work [18] [87]. This section explains different types of the heap with its pros and cons [108]. Of these, anyone can be used according to its application.

1) Sequential Single Heap

Sequential single heap used by the memory management algorithm generally has lower fragmentation and faster execution time. Here, the heap is secured by a universal lock which leads the sequential accesses of memory and creates lock conflict.

2) Synchronized Single Heap

The usefulness of a synchronized single heap is to decrease the sequential accesses of memory and heap conflict. It is typically carried out by a synchronized data-structure such as a B-

tree or an unallocated memory block list with locks. But, its cost of accessing memory is comparatively high. Also, false sharing remains as it is.

3) Absolute Reserved Heap

An Absolute Reserved heap specifies that an isolated heap is assigned to every thread and these threads are distinct from other existing threads. For this, why every thread cannot use other reserved heap for the memory operations like read or write. Therefore, a memory management algorithm having various heaps can decrease the majority of the lock conflicts on the reserved heap, and become scalable. Inappropriately, it creates reserved heap to develop without any bounds. For example, two threads have a producer-consumer association where a thread T1 acts as a producer assigning memory Mem1 and another thread T2 acting as a consumer frees Mem1. In this, Mem1 must be further added into T2's heap.

4) Reserved heaps with rights

In contrast to the memory manager which uses absolute reserved heaps, the memory managers with rights provide unallocated memory blocks to the heap. However, in a producer-consumer association based applications, a round-robin way, the memory manager can remove the generated false sharing. It still generates boundless memory usage.

5) Reserved heaps with thresholds

The memory manager generates an order of heaps, and multiple threads can share some of the heap excluding the reserved heaps. The heap, which is shared, may demonstrate heap conflicts, in rare case. So, the memory managers may be effective and scalable. If reserved heaps cross the threshold value, some part of unallocated memory will be migrated to the shared heap. The cost of managing memory, especially the cost of time, may be huge because so many memory processes are required for this.

Majority memory allocators have a specific issue of boundless heap increments due to multiple heaps, where memory usage cannot be unbounded by any policy even if the requisite memory is fixed. It is known as blowup occurrence [6]. Some of the memory management algorithms like Hoard [6] and Vee [115] demonstrate restricted memory usage as they provide the reserved heaps with thresholds.

2.3 Memory Management Algorithms

Since 1963, several memory management algorithms have been designed and implemented. Majority of the current memory management algorithms are modifications of the traditional allocators. Furthermore, these traditional memory management algorithms are ordinary and very simple to use in minor devices [19]. There are two different approaches to analyze Worst-Case Execution Time of memory allocators: 1) Static Worst-Case Execution Time analysis 2) Worst Case Complexity analysis. But, without prior information of the allocation or deallocation demands, it is difficult to use Static Worst-Case Execution Time analysis [88] [91].

Now, different memory management algorithm such as sequential and segregated fit, buddy allocators, indexed/bitmapped fit and current memory management algorithms will be discussed.

2.3.1 Sequential Fit

Sequential fit allocators can be categorized into four categories: 1) Best-Fit 2) First-Fit 3) Next-Fit and 4) Worst-Fit. Figure 2.1 [100] demonstrates the variances between allocators of sequential fit. It consists of five unallocated memory blocks of dissimilar sizes with six allocated memory blocks. The sizes of the memory blocks are specified in the caption of every memory block which has indicators to represent a doubly link list.

For Example, one application demands 2Kb of unallocated memory. Here, as per their policies, the best-fit will provide the 2Kb unallocated memory block, 3Kb of unallocated memory block will be provided by First-fit and Next-fit whereas 5Kb unallocated memory block will be provided by Worst-fit. This has been demonstrated in Figure 2.1.

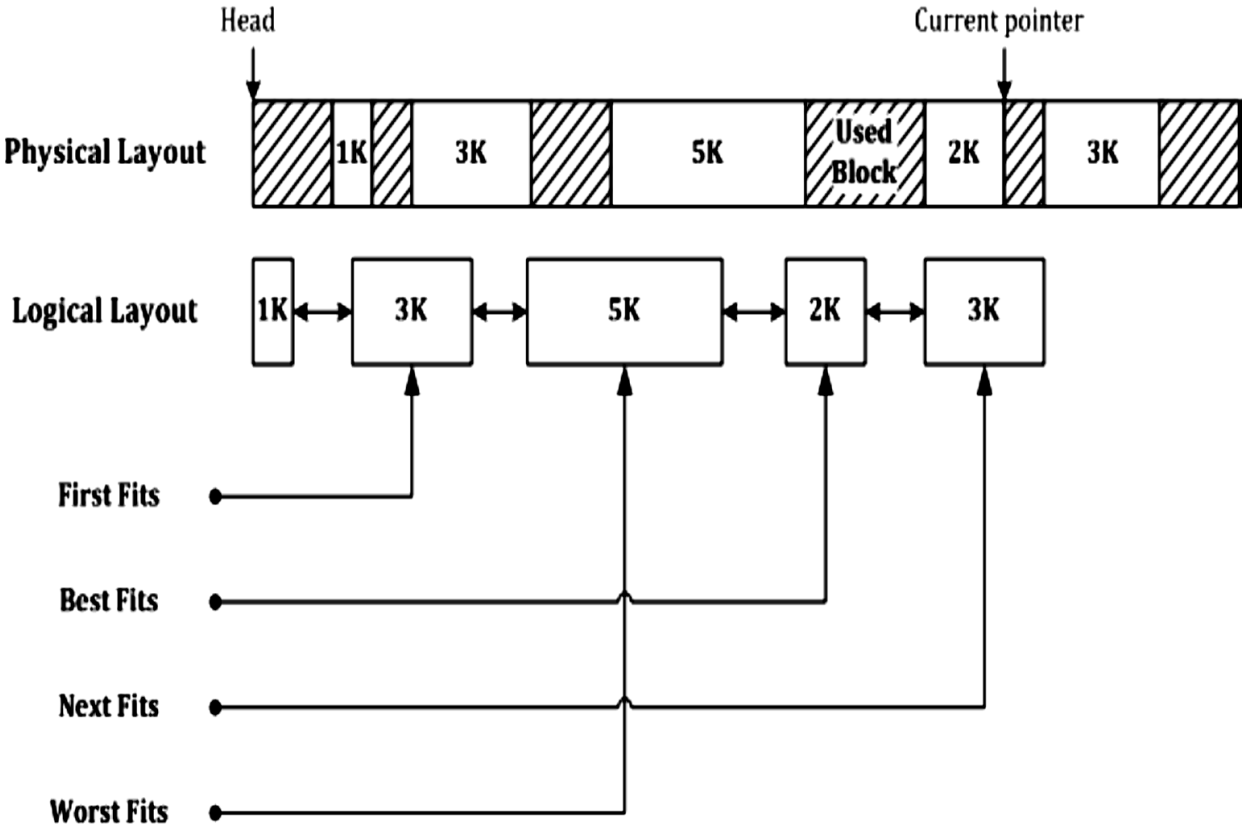

Figure 2.1: Structure for Sequential-Fit Allocators

2.3.1.1 Best-fit

The allocator is normally implemented using either a doubly or circular linked list. However, it is not only a single list of available unallocated memory blocks, but it has arrays of different category and sizes.

Generally, Implementation of best-fit policy can be done in various ways such as address-order, First-in-First-out and Last-in-First-out. These memory management algorithms are implemented based on in-depth searching algorithms. They also demonstrate proper and decent usage of memory but they also lead to some fragmentation [51]. Though, it is not an important issue [5] [117].

In best-fit allocator, the algorithm discovers for an unallocated memory block, which is big enough to fulfill the demand of the application, sequentially from the top of the unallocated block list till it finds an appropriate memory block. If the memory manager discovers an appropriate

block, the searching process ends, and it provides the memory block. If the memory manager discovers more than one appropriate memory blocks, the selection of memory block relies on the implementation of the allocator. If the size of the discovered memory block is bigger than the size demanded, the memory block will be split into two parts and the remaining part will be pushed into the unallocated block list, otherwise the memory manager will fail to allocate demanded block. In deallocation of the blocks, released blocks are combined with adjacent free memory blocks.

2.3.1.2 First-fit

The memory management algorithms which are employing First-fit policy are also most general sequential fit mechanisms [9] [51]. They try to discover the first unallocated memory block which is big enough to fulfill the demand of an application. Implementation of the First-fit policy can be done in various ways such as address-order, First-in First-out or Last-in First-out mechanisms. This one also, same as best-fit, can find a block in O(n) time complexity with the help of an array which is a doubly linked list [103] [104].

The First-fit allocator finds an unallocated block sequentially from the top of unallocated block lists until the demanded block is not found. If there is no suitable block found, it shows failure. If the size of the searched memory block is bigger than the demanded size, it is split into two parts and the remaining part will be pushed into the unallocated block list. In the deallocation of the blocks, released blocks are combined with adjacent free memory blocks.

The main issue with the first-fit policy is that repeated splitting happens on the top of the unallocated block list which creates so many small memory blocks adjacent on the top of the list. Due to these small memory blocks the searching time is increased as it requires going through entire list every time resulting into high fragmentation. Therefore, the traditional first-fit policy is not appropriate for an application which allocates and deallocates different sizes of memory blocks repeatedly. But, like best-fit, it may be more appropriate if implemented via additional classy data structures.

2.3.1.3 Next-fit

The allocation algorithms which employ next-fit policy are one of the variations of algorithms having a first-fit policy which has a moving pointer for memory blocks allocation [51]. The allocators preserve a pathway of the pointer which stores the position of the block where the previous search was successful. This pointer will be used as the starting point for the subsequent search. Implementation of the next-fit policy can be done in various ways such as address-order, first-in-first-out and last-in-first-out mechanisms.

Theoretically, the next-fit decreases the aggregate search time for a singular link list; but it tends to get inferior zone because it examines each unallocated memory block before searching the similar memory block due to the moving pointer.

As the pointer moves over the unallocated memory block lists frequently and is expected to store the objects in memory with different size and periods from different stages of the application's runtime. Therefore, it generates higher fragmentation than other sequential fit memory management algorithms. The allocators having a next-fit policy with last-in first-out mechanism have considerably lesser fragmentation than next-fit allocators implemented by address-order [117]. Next-fit is the finest solution to reduce the average response time in a mutually shared memory of SMP architecture [1] [3] [70].

Regarding allocation and deallocation of the memory blocks, the allocators having a next-fit policy are nearly identical to the first-fit allocators except the beginning point of finding.

In summary, sequential fit memory management algorithms are developed with the help of a linear list, which contains doubly or circular linked lists with various policies in like first-in-first-out or last-in-first-out. The best-fit as well as the first-fit are created using first-in-first-out or an address-ordered policy appears to work fine. Conversely, scalability is a significant issue with sequentially fit allocators because unallocated blocks increase the cost of searching.

Sequential fit algorithms can be collectively used, like half-fit, optimal-fit, or worst-fit. For example, in worst-fit policy, it searches for the biggest unallocated memory block which is big enough to fulfill the demand of an application as it tries to create the unallocated block as large as it can to avoid small fragmentations.

In the worst-case, the algorithms allocate memory blocks in O(n) time complexity, where n is the heap's size. These algorithms are not feasible and not adequate for any real-time systems.

2.3.2 Segregated unallocated block Lists

Majority of the latest memory management allocators, like tcmalloc, Tow Level Segregated Fit, DougeLea malloc, Hoard etc. implement segregated unallocated block list mechanisms which use an array of unallocated memory block lists. Due to this, the searching time for the demanded block is reduced but it also creates little internal fragmentations. The allocator uses the size of the memory block to maintain and separate out the different size block. Each block size is a power of two apart that's why each class of size keeps unallocated memory blocks of a specific size.

The allocators round up the demanded block size to the nearest class of size. The memory allocator searches for an unallocated block of the demanded size, which is big enough to fulfill the demand of an application, in a specific sized class or of marginally lesser size which is bigger than any smaller sized class.

While in deallocating of memory block, the memory management algorithm pushes an unallocated memory block into the unallocated block list for the given size on release of the allocated memory block. Wilson, in [117], specifies that the allocators having segregated unallocated block lists can be categorized into two classes: 1) simple segregated storage 2) segregated fit.

2.3.2.1 Simple Segregated Storage

It is one of the simple memory management algorithms which use an array of unallocated memory block lists. In this, there is no requirement of splitting and merging of unallocated blocks. These features differentiate between simple segregated storage from buddy systems. The main benefit of this method is that no captions are needed. As caption exhibits overheads, this algorithm

reduces memory usage – the captions typically raise memory usage by some percentage [120] – which is most significant when the average demanded size is very less.

As there is no need of splitting or merging of blocks and keeping of captions, these allocators are faster; particularly when the memory blocks of a provided size are demanded frequently in a short time span. It allocates the demanded block in O(1) time complexity.

The issue with this allocator is that it makes large external fragmentation, which is relative to the highest amount of memory used by the allocator (Mem_used) times the maximum block size (Mem_max) demanded by the application. It also generates internal fragmentation.

2.3.2.2 Segregated Fit

These allocators use arrays of unallocated block lists. Every array keeps unallocated blocks within a specific size class. This memory management algorithm is faster than a single unallocated block list for the majority of the cases, as it finds the unallocated block list for the suitable sized class when the application demands specific memory. It, then, selects an array within a specific size class and finds an unallocated memory block from the array with a sequential fit mechanism. If the unallocated block is not available, the allocators try to discover for a bigger block in the adjoining array continuously until it finds a bigger block. The bigger block will be split into two parts, the requested and the remainder. The remainder part will be pushed onto a specific array. This memory management algorithm is classified as 1) Exact Lists 2) Strict Size Classes with Rounding and 3) Size Classes with Range Lists.

1) **Exact Lists**: In this mechanism, the memory management algorithm requires having lists with an enormous number of unallocated block and each list has all possible block size. The memory managers use the exact lists inside the classes of small size to decrease a huge number of unallocated block lists.

2) **Strict Size Classes with Rounding**: According to this mechanism, the memory management algorithm rounds up the demanded size to the nearest sizes in the class of size, though it wastes

some of the memory space as internal fragmentation. The most significant benefit of this allocator is that it can hold all memory blocks of the same size on a single size list.

3) **Size Classes with Range Lists**: This mechanism allows unallocated block lists to maintain blocks of slightly different sizes. This is an extensively used mechanism for memory allocations.

Summarizing all approaches, the segregated unallocated block lists can be used with other mechanisms like first-fit or best-fit to find a specific unallocated block list. If the memory manager discovers a specific size list, it examines for an unallocated block in the list according to these mechanisms. In the worst-case, this memory management algorithm accomplishes the task in O(1) time complexity but due to large external fragmentation, these algorithms are not appropriate for the real-time system.

2.3.3 Buddy Allocators

Buddy allocators are specific derivations of segregated unallocated block lists with the help of size class which rounds up to nearest value. In this allocator, the entire area of the heap is hypothetically distributed into two regions. These regions are, in turn, divided into two smaller areas, and so on so forth. This scenario is known as simple buddy situation. In worst-case, the standard algorithm of buddy allocator has O($\log_2 n$) time complexity, where n is the highest heap size.

The significant benefit of this allocator is that it holds all available memory blocks on a size list that are of precisely of the same size. For example, if a size of an unallocated block list is 2Kb, then each block available in the list will be of the size 2Kb. The only difference between this allocator and the other segregated lists allocators is that there is a limited splitting and merging of unallocated blocks with the help of a specific equation. For example, a power of two equation is as under:

$$i = \lfloor \log_2 range \rfloor \qquad 2.2$$

Here, the parameter ***range*** is used to calculate a specific index which signifies a specific unallocated block list within the size class to be used for either pushing an unallocated block or finding for an unallocated block.

For allocating a memory block in a simple binary buddy, the algorithm searches an unallocated block inside a specific array acquired by equation 2.2. If the unallocated block is not available, then the allocators attempt to discover a bigger memory block in the nearby array sequentially till searching a bigger block is found. If an unallocated block has been discovered in some upper range array, the unallocated block will be removed from the unallocated block list and recursively divided into the size of power of two logarithmically to be of lesser size. However, the size of the allocated block would be still big enough to fulfill the demand. The residual blocks which are created due to splitting are pushed into the corresponding unallocated block lists.

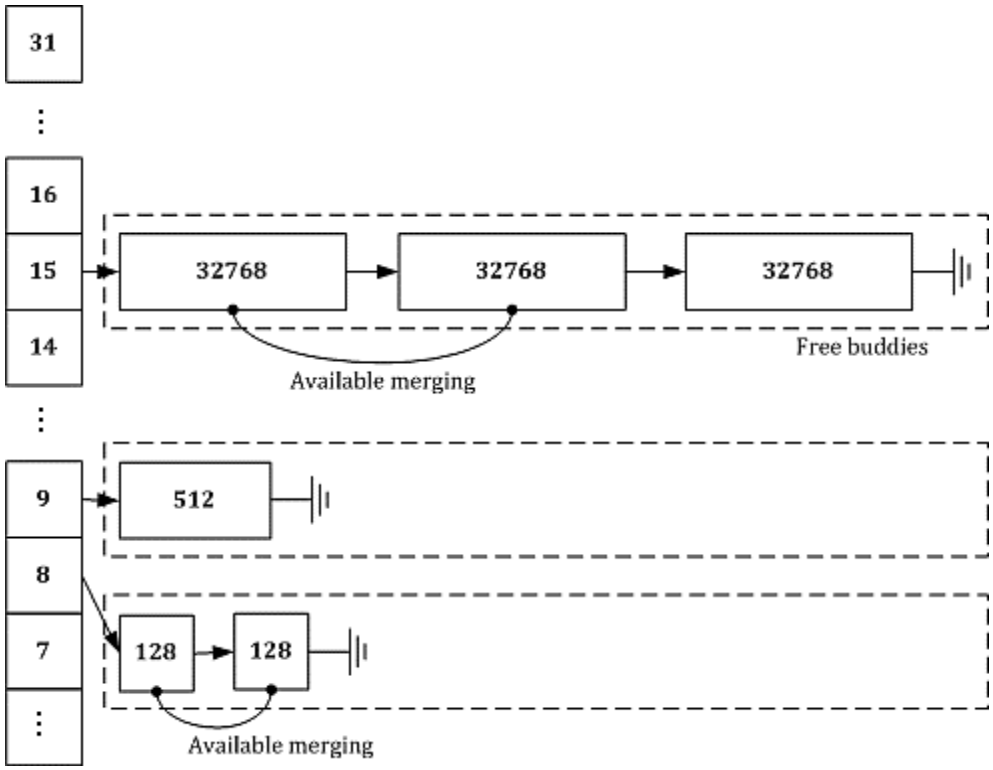

Figure 2.2: Structure of Buddy Allocator [100]

When an application releases an allocated memory block, it uses equation 2.2 to find the specific array inside size classes. The array found by the equation 2.2 is analyzed to check if it keeps neighboring memory blocks which are already unallocated and if it is so, then allocator

combines the released block with the neighboring memory blocks to make a new memory block of exactly dual the size. Subsequently, this procedure sequentially repeats until all unallocated block which encounters the criteria mentioned earlier. This simplest allocator is known as the binary buddy system. Wilson [117] categorized buddy systems into four categories as: 1) Binary Buddy 2) Fibonacci Buddy 3) Weighted Buddy and 4) Double Buddy.

1) **Binary buddy:** It is the simplest form of buddy systems. According to this allocator, all available memory blocks' size is a power of two, where every size is split into two identical fragments and combined into one dual size. Due to these features pointer manipulations become easy. For this, it has usually been considered as an allocator for a real-time system. In the worst case, this allocator performs the task in $O(\log_2\frac{m}{n})$ time complexity, where m is the highest heap size, and n is the highest allocated size of memory used by the application. The issue with these allocators is that internal fragmentation is comparatively very high around 25% [51].

2) **Fibonacci Buddy:** In 1997, Knuth [51] had proposed an allocator with the help of Fibonacci series numbers as the block sizes of buddy rather than a power of two, which lead to reduction in internal fragmentation related to binary buddies [39]. As all block sizes are a Fibonacci number, i.e., the addition of the two preceding numbers, one memory block can only be split if sizes are within the Fibonacci numbers as well. An issue with this allocator is that the residual block is probably unusable if the application allocates numerous unallocated blocks of the similar sizes [117].

3) **Weighted Buddy:** These memory management algorithms [83] [102] allow handling the classes of size in two ways. The size of all available memory blocks is 2^n and $3*2^n$ for all n. Hence, the size classes are comprised of the powers of two and three times a power of two among each pair of successive sizes, for example, 2, 3, 4, 6, 8, and so on. One significant benefit of this allocator is that, here, the mean internal fragmentation is lesser than the other existing buddy systems. On the contrary, a major issue with these allocators is that it creates large external fragmentation than other existing buddy systems [14].

4) **Double Buddy:** In 1986, Hagnis and Page [83] proposed allocator and named it double buddy systems. It is one of the implementations of weighted buddy allocator. These algorithms decrease the fragmentation related to weighted buddies. Here, the splitting criterion is the main difference between double buddy and weighted buddy. All unallocated memory blocks can be split in half, precisely creating all memory blocks in the size of power of two which is similar to the binary buddies.

In the worst-case, the time complexity of all algorithms discussed above are $O(\log_2 n)$ but because of large internal fragmentation [110], these algorithms are not appropriate for the real-time system.

2.3.4 Indexed Fit and Bitmapped Fit

In the sequential fit mechanism, for searching an unallocated block, linear search is required. It is also required in segregated fit for passing through an array inside a size class. An indexed fit mechanism is used to increase the speed of discovering an unallocated block of memory management algorithms. It can be used with other mechanisms to create a hybrid mechanism with different types of data structures. These mechanisms are used in detail indexing to maintain track of unallocated memory blocks inside size-based policies. Overall, all of these memory management allocators are the modifications of the indexed fit mechanism because the majority of the allocators maintain a record about which parts of the memory have been used or which are still unallocated [15] [24].

For example, the bitmapped fit mechanism is inherited from the indexed fit mechanism. This memory management allocator maintains a record about which portions of arrays are in use and which portion is free. Each bit is mapped to an unallocated memory block or array, based on their mechanisms. A bitmapped fit structure is used very less because, traditionally, it is very slow in finding an unallocated block. Now, with the latest processors, based on bit search instructions, it consumes limited clock cycles of the processor to find a block.

The most significant benefit of a bitmap scheme is that it can be developed with very less amount of memory. For instance, it can be one or two words in some of the application. If the development of a memory management algorithm needs a huge amount of memory, then there is a maximum possibility of interleaved data structure across the nodes in a Non-Uniform Memory Access architecture. To improve locality of searching on the node itself, a bitmap per node can be used.

Best examples of an indexed fit with bitmapped fit structures are Half-fit [80] and Two-Level Segregated Fit [59]. They use bitmaps to maintain to find which areas are unallocated. Fast fit [105] is also one of the best examples of an indexed fit mechanism which implements a Cartesian tree.

2.3.5 Hybrid Allocators

As stated earlier that a single memory management allocator has numerous drawbacks. Hence, the majority of the latest memory management allocators have combined various allocators to increase the speed of searching and inserting memory block [54] [55]. Majority of the algorithms used by latest allocators are a mixture of sequential fit, buddy system and segregated fit algorithms. In this section, some of the most extensively used hybrid memory management allocators in the general-purpose operating system as well as the real-time operating system.

2.3.5.1 Doug Lea(DLmalloc)

This algorithm was developed by Doug Lea in 1996,s extended by Gloger in 2006 [23] [34] and third time modified by the Free Software Foundation in 2012 [23] [28] [29]. In these all extended version of the memory management algorithms, the most important strategy and policy persist as it is. However, in terms of improvements in the latest extension, they implemented bitmaps to discover available unallocated blocks instead of the sequential searching which was being used in its earlier version. Presently, it implements a mixture of various mechanisms, based on the demanded size. It also employs the good-fit policy together with the segregated size-classes mechanism. DLmalloc allocator is designed and developed to use two different data structure,

based on the memory blocks' size. Figure 2.3 [52] [100] [112] demonstrates the structure of DLmalloc.

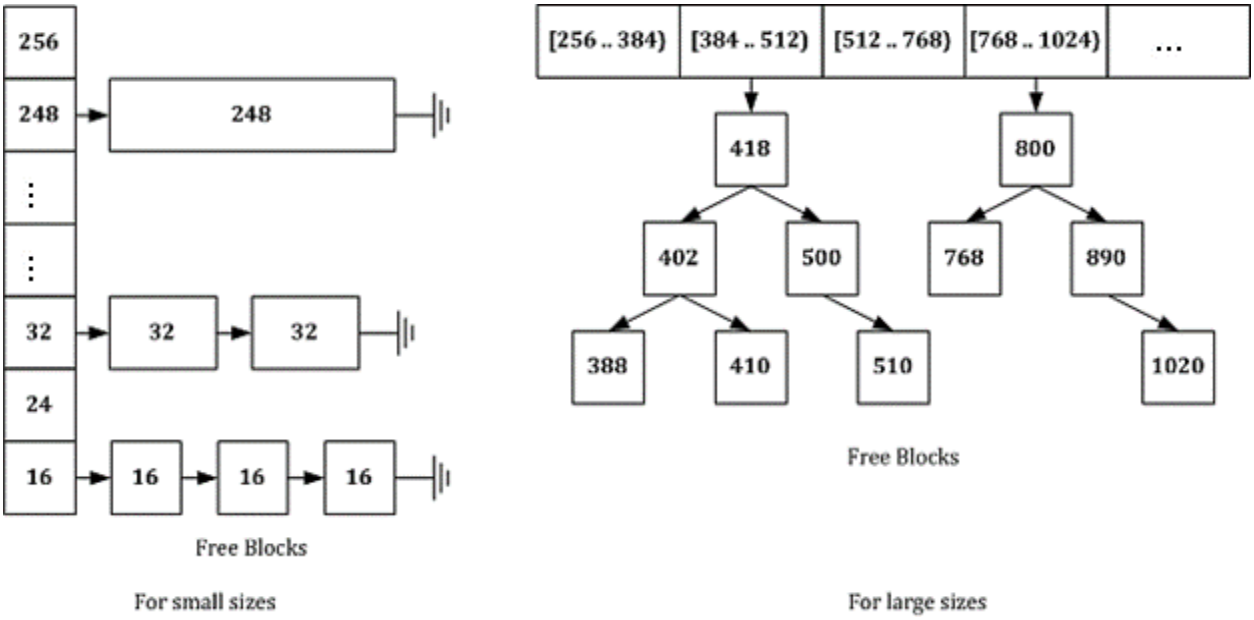

Figure 2.3: Structure of DLmalloc

To allocate a small memory block, this algorithm occupies a huge number of static size arrays known as small-bins. Bins occupy unallocated blocks which are having sizes not more than 256 bytes. Every bin comprised of unallocated blocks of equal size. If demanded memory block's size is not more than 256 bytes, the algorithm tries to find for existing blocks in the bins using the best-fit policy or finds blocks which are large enough to satisfy the request.

If the size of the demanded memory block is larger than 256 bytes and lesser than some fixed value (normally 256KB), the algorithm tries to search existing blocks in an array known as tree-bin, which is having a tree structure for the memory blocks. As shown in figure 2.3, tree-bins consist of a collection of the bin. Nodes in the tree structure act as small-bin, comprising of the blocks of equal size. For any kind of demands of block size beyond the fixed value, the algorithm transfers the demands to the operating system through some specific system call.

In the worst-case, to find a small block whose size less than 256 bytes, this memory management allocator takes O(1) time and for a larger block whose size is greater than 256 bytes, it takes O(m) times, where m is the depth of the tree.

2.3.5.2 Half-Fit

This algorithm, designed and implemented by Ogasawara 1995 [80], uses bitmapped fit strategy and achieves execution time in a constant manner. As it uses bitmap policy for allocating a released block, it is slow. Here, the bitmap is used, only, to maintain the status of unoccupied lists. The time complexity of half-fit is O(1).

This algorithm maintains a segregated list of a single level. In this list, unallocated blocks of different size are connected. It takes unallocated blocks of the required size from unallocated block list to satisfy the request. Figure 2.4 [80] [100] [112] shows the structure of Half-fit.

This algorithm has a specific allocation-deallocation methodology to avoid searching using bitmaps because of its constant execution time. If the requested size of the memory block is r, then the index i may be computed by this equation [79]:

$$i = \begin{cases} 0 & if\ r\ is\ 1 \\ \lfloor \log_2(r-1) \rfloor + 1 & otherwise \end{cases} \quad 2.3$$

Where i specifies the unallocated memory block lists whose width vary from 2^i to $2^{i+1} - 1$. After computing the value of i, an unallocated block is occupied from the block list indexed by i. If the there is no unallocated block in the list, the subsequent unallocated block list will be searched. If the size of an assigned memory block is more than the demanded memory block size, an unallocated block from the unallocated blocks list will be split into two distinct memory blocks of sizes r1 and r2 before allocation and the remaining memory block r2 will be put into the matching unallocated block list.

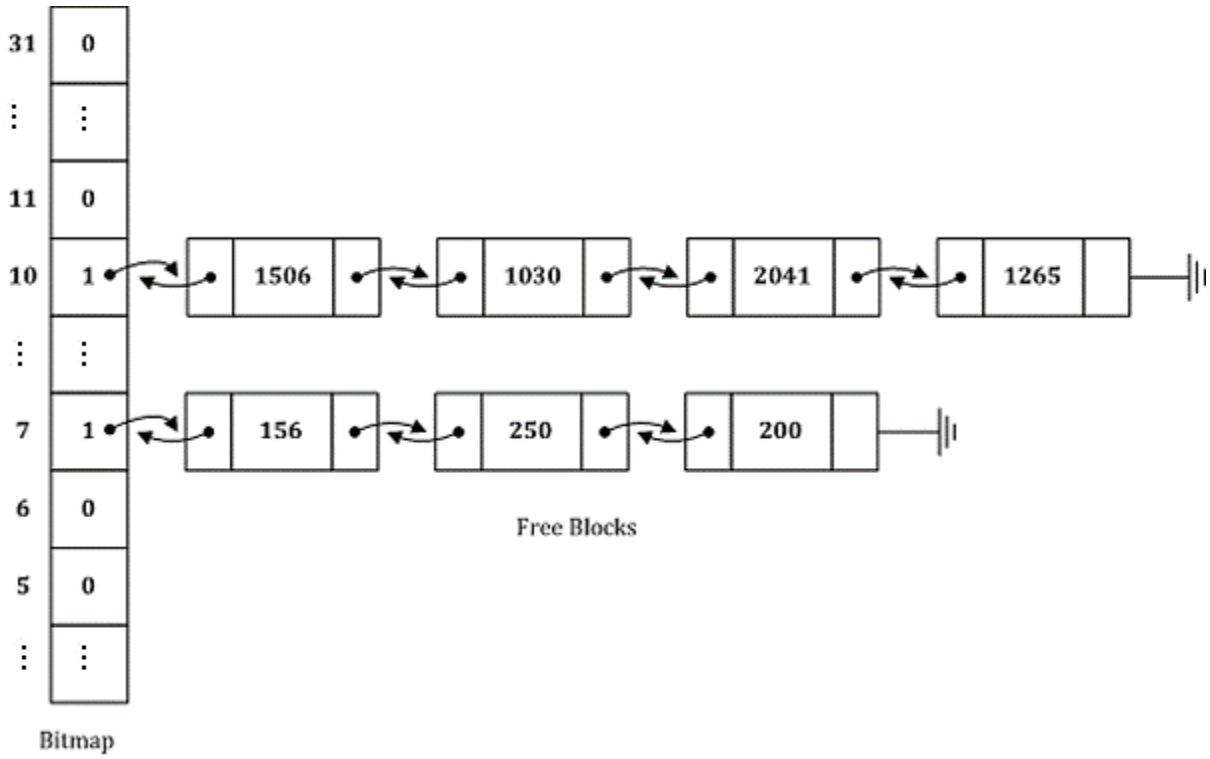

Figure 2.4: Structure for Half-Fit

For the deallocation process, the released memory block will be directly merged with neighboring memory block if it is unallocated. After an unallocated memory block is combined, the combined memory block's size is r, and this new memory block is pushed onto the top of the unallocated block list indexed by i. To find the value of i, the algorithm uses the following equation 2.4 [79]:

$$i = \lfloor \log_2 r \rfloor \qquad 2.4$$

As adjacent memory blocks are linked by doubly link list, merging with neighboring unallocated blocks can be done in O(1) time.

This algorithm can be used for real-time operating systems because of its constant time complexity. Furthermore, it demonstrates the best performance as compared to the binary buddy allocators. This allocator also provides the best response time in a worst-case scenario. But, it is not an ideal algorithm for the real-time operating system due to its internal fragmentation which is created due to its merging policy.

2.3.5.3 TLSF

TLSF (Two-Level Segregated Fit) [13] [58] [63] [67] is one of the best available dynamic memory allocation algorithms. This allocator is an enhancement of Half-fit allocator [60] [79]. This is the best allocator for the real-time operating system. Unlike a segregated list allocator, this algorithm has two levels of segregated lists of unallocated memory blocks; in which each list maintains the unallocated blocks of predefined size range to decrease internal fragmentation.

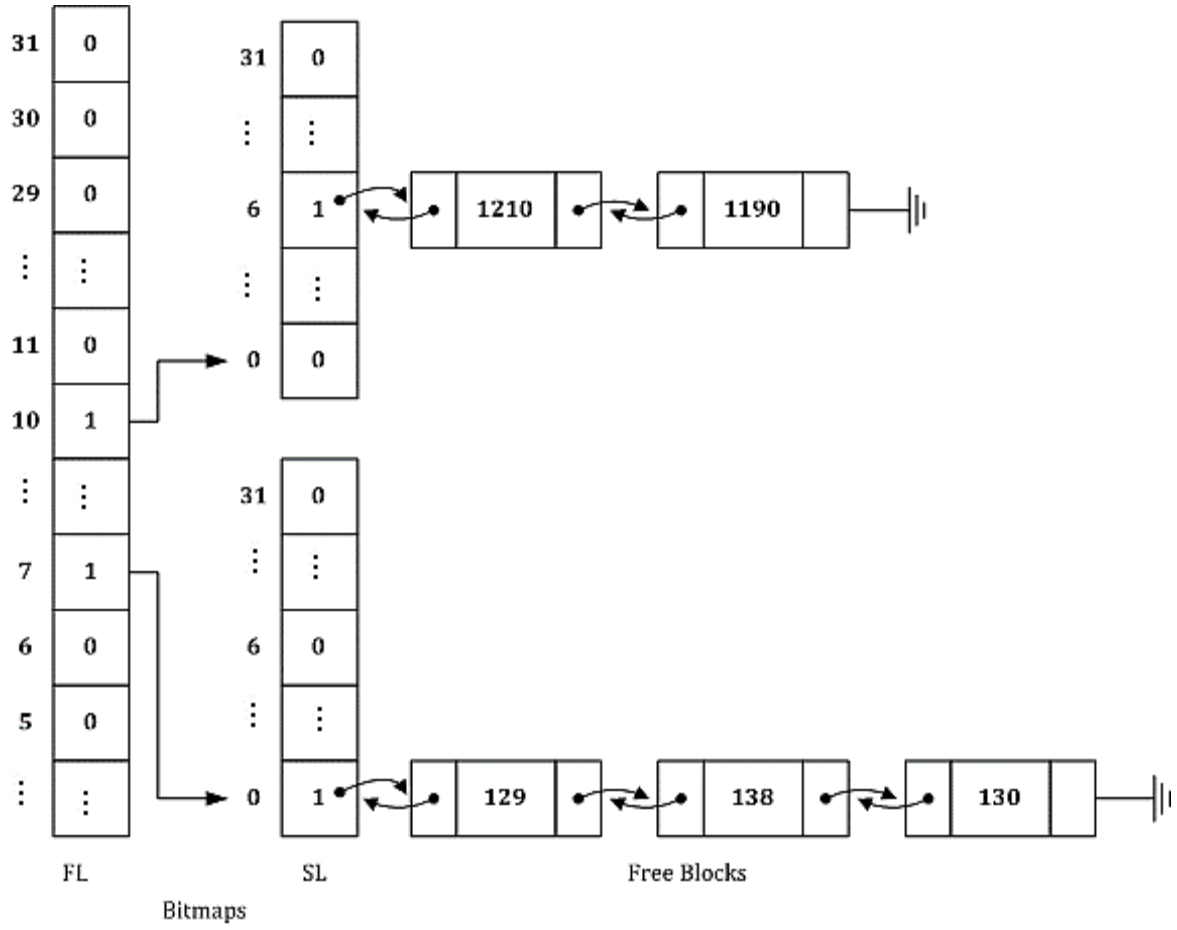

Figure 2.5: Structure of TLSF

The first-level of the list (FL) splits unallocated memory blocks into parts which are apart from each other by the power of two like 2, 4, 8, 16… and so on. The secondary array, known as second-level lists, splits each of the first level lists by a user-defined variable known as Second Level Index. TLSF structure is shown in Figure 2.5 [58] [100].

The primary goal of TLSF is to deliver restricted response time in both the procedures of memory allocation and deallocation irrespective of the size of memory. For allocating a memory block, TLSF uses equations 2.5 [68] [69] [118], where the specified size of a block computes the indexes of the two arrays pointing to the related segregated list.

$$\mathrm{i}(f,s) = \begin{cases} f = \lfloor \log_2(r + 2^{\lfloor \log_2(r) \rfloor - \mathrm{SLI}} - 1) \rfloor \\ s = \left\lfloor \dfrac{(r + 2^{\lfloor \log_2(r) \rfloor - \mathrm{SLI}} - 1 - 2^{i})}{2^{f-SLI}} \right\rfloor \end{cases} \quad 2.5$$

First-level of list (FL) f can be computed as the location of the previous bit set of the size, where the bit is set to 1. This index specifies blocks of memory according to their size. Each FL location indicates to a specific size class. For example, FL3 specifies size class width from 8 bytes to 16 bytes; FL4 specifies size class width from 16 bytes to 32 bytes. The second-level of the list can be calculated by the same equation. Each position indicates a memory block within similar sizes. These equations can be proficiently employed by the latest processors' instructions set only.

If the provided size of a memory demand is r, the index i (f, s), which is used to take the top of the unallocated block list keeping the nearest class list, is computed by the above-mentioned equation 2.5. If an unallocated block is discovered from the unallocated block list indexed by i, only that particular block will be fetched from the top of the unallocated block list and returned. If searching of the unallocated memory block at index i fail, then the subsequent unallocated block list whose index is nearby i will be inspected. Then, an unallocated memory block from the nearest unallocated block list of bigger sizes will be split into two memory blocks of sizes r and r′. The residual memory block of size r′ is pushed onto the related unallocated block list.

For the deallocation of a memory block, released memory blocks are directly merged with the neighboring unallocated memory blocks. If the neighboring blocks are unallocated, the neighboring blocks will be fetched from the segregated list and combined with the present memory block. The new block will be pushed onto the top of the unallocated block list indexed by j. For the calculation of j, TLSF uses the following equations 2.6 [68]:

$$\mathrm{j}\,(f,s) = \begin{cases} f = \lfloor \log_2 r \rfloor \\ s = \left\lfloor \dfrac{(r - 2^f)}{2^{f-SL}} \right\rfloor \end{cases} \qquad 2.6$$

As TLSF explores the demanded memory block's size to compute the suitable location indexed by the FL and SL depending on equations 2.5 and 2.6, it can be accomplished in O(1) time complexity.

Here, TLSF uses a shared heap which creates heap conflict in multi-threaded environments and it is not scalable also. Although, the improved version of TLSF does not have a constant execution time, its policy can still be accomplished in a constant time. In the modern type of TLSF [65], multiple memory pools are used along with an extra pool of memory which will be generated when available pools of memory are occupied by system calls.

The issue which may arise here is that if the last allocated memory block in each pool of memory requires to be free and the memory manager has already used the multiple pools of memory where the later releases the last allocated block, then the pool of memory that has the last allocated memory block will be put in the unallocated state with TLSF trying to combine neighboring memory pools while deallocating blocks of memory. Under this policy, the time of the merging process can rise linearly; therefore if there are N unallocated memory pools, it traverses N times to combine each other sequentially. Hence, under this policy, TLSF achieves O(n) time complexity [66].

2.3.5.4 tcmalloc

This algorithm is developed and implemented by Sanjay Ghemawat in 2010 [99]. It is an algorithm which associates both global heap structure and thread's private heap multiprocessor architecture. To allocate small memory block which ranges from 4 bytes to 32 Kb size, this allocator allocates private local heap to each thread. Hence, small size memory block allocation does not require synchronization mechanism for the thread.

To allocate large memory blocks which ranges from 32 Kb to 1 Mb, this allocator maintains a global heap structure which is collectively used by all available threads. As this global heap is shared by threads, some kind of synchronization mechanism is required to offer mutual exclusion. For this, it employs a spin-lock mechanism. If any of the applications demand a huge memory block whose size is beyond 1Mb, then allocator forwards the request to the existing operating system using a system call or APIs. Figure 2.6 [99] [100] [112] shows the structure of tcmalloc allocator. The time complexity of this allocator is O(1).

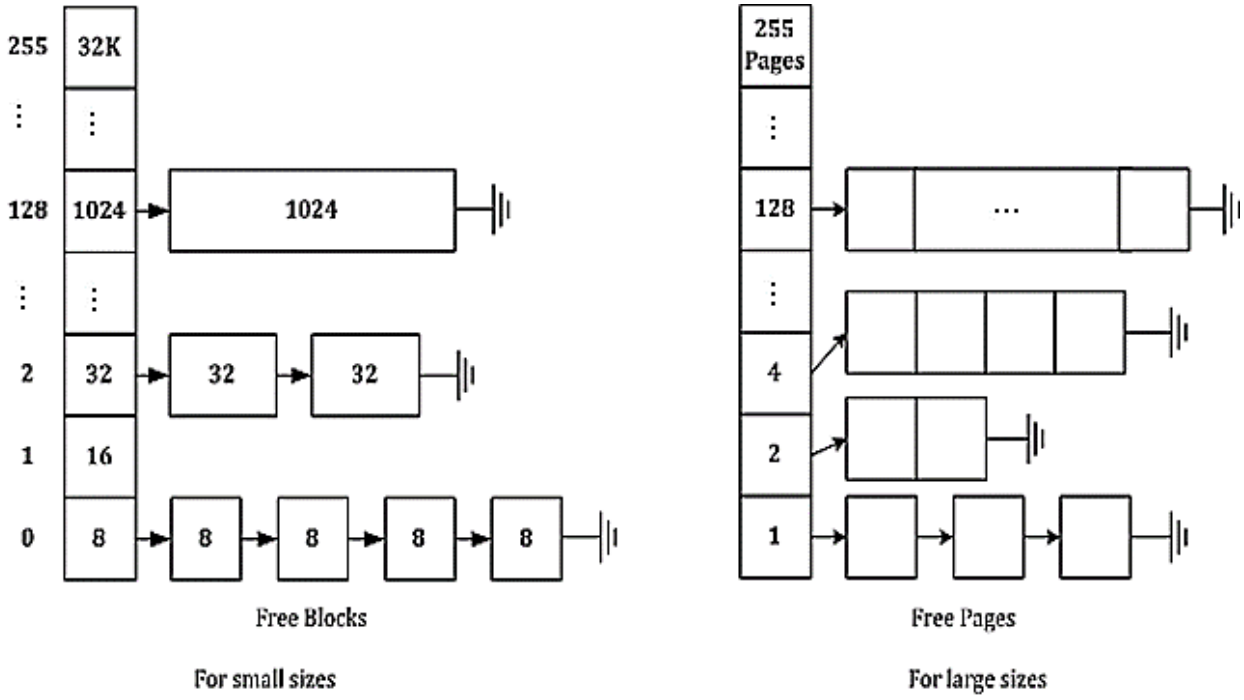

Figure 2.6: Structure of tcmalloc

If the available size of a demanded memory block is less than 32Kb, the memory manager examines whether it can fulfill the demand from the thread's private heap. If this fails, tcmalloc examines the global heap with the help of a lock to achieve synchronization. If the unallocated block list is empty, the subsequent unallocated block list will be inspected, and so on. If the global heap has an unallocated memory block of adequately big size, it will be split into two memory blocks, one block is remainder and another one is served. The remaining unallocated memory block will be pushed onto one of the unallocated block lists in the global heap.

In the deallocation of the memory blocks, if any thread releases a small memory block, then it will be pushed onto the list in the private thread heap of the related size. If the size of an unallocated block list goes beyond a specific threshold, for example, 2MB, then the memory management algorithm moves some of the unallocated blocks back to the global heap.

The tcmalloc uses the demanded size of the memory block to compute the suitable index in the global heap or private thread heap. So, this operation can be accomplished in O(1) time complexity. The main disadvantage of this allocator is that it has boundless memory consumption and it does not address to false sharing [113].

2.3.5.5 Hoard

This algorithm has been designed by Bergar in 2000 [6] and it is popular due to its speed and scalability in the multiprocessor environment.

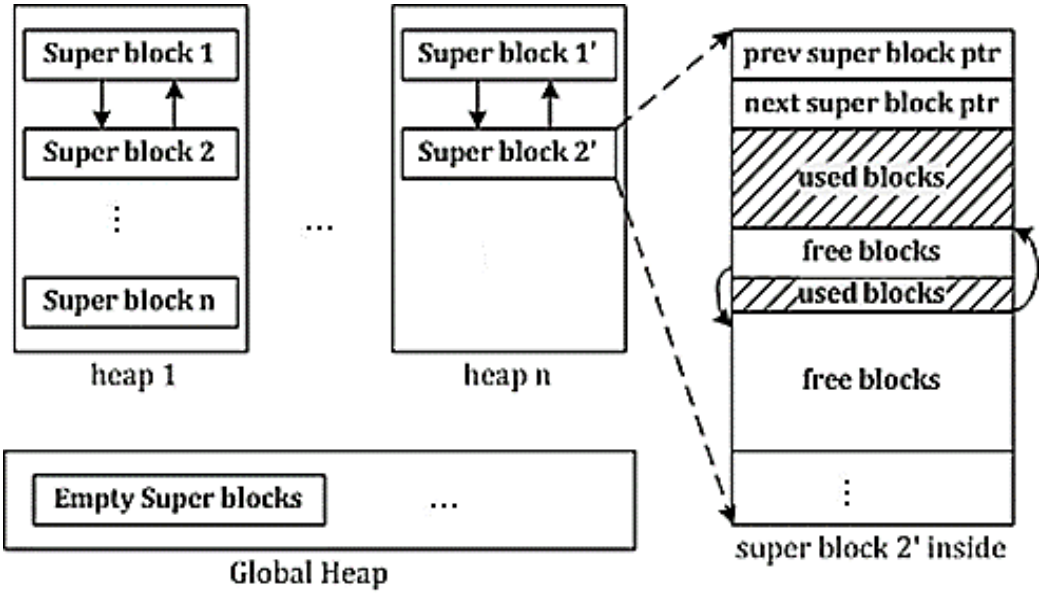

Figure 2.7: Structure of Hoard

This allocator also uses a segregated class mechanism, but unlike tcmalloc, it maintains a private heap for each processor and also maintains a global heap to avoid heap conflicts. Other than private processor heap and global heap, it also maintains one private heap per thread to allocate smaller size memory blocks whose size is less than 256 bytes. Threads which are executing

on the same processor can also share private processor heap. Figure 2.7 [6] [100] [112] shows the structure of this algorithm.

This algorithm, to allocate any blocks, whose size is less than 256 bytes, it first searches it into a heap of the thread and if it fails, then it searches into private processor heap. In a private processor heap, the memory manager allocates memory blocks from the system in a large piece, called superblock, which is an array of some sets of memory blocks comprised of unallocated block list. If private processor heap is completely searched, then it searches into a global heap. In such scenario, the thread locks the private processor heap which will be unlocked when the memory manager moves a new superblock to the private processor heap from the global heap.

This allocator has the time complexity of O(n) for allocation of a memory block as it requires to discover consecutively for the thread's superblock. Here n is superblock which is the number of chunks of the memory block. As mentioned earlier, there can be heap conflicts between different threads of the shared private processor heap; this allocator addresses it with double private processor heaps corresponding to a number of processors in the system. This allocator also uses a specific mapping function between processors and threads. However, the cache misses arises, here, due to the disruption of node oriented data locality.

2.3.5.6 Smart Memory Allocator Algorithm

This algorithm has been proposed by Ramakrishna M, Jisung Kim, Woohyong Lee and Youngki Chung in 2008 [26] [119]. This is a custom type of dynamic memory algorithm having the best response time and less memory fragmentation. This algorithm divides memory blocks into two categories: one is short-lived, and the other is long-lived memory blocks. The short-lived memory blocks are allocated in the direction of lowest level to highest level from heap whereas the long-lived memory blocks are allocated from highest level to lowest level [10] [12].

The used space of heap grows from highest level to lowest level as well as lowest to highest also. Initially, the entire heap memory is unallocated, and there will be one unallocated memory block for both the categories, short and long-lived memory [2] [4]. The heap space is divided

equally into two blocks. When the heap grows from both the sides, the virtual border between these two can easily be modified according to the dynamic memory request.

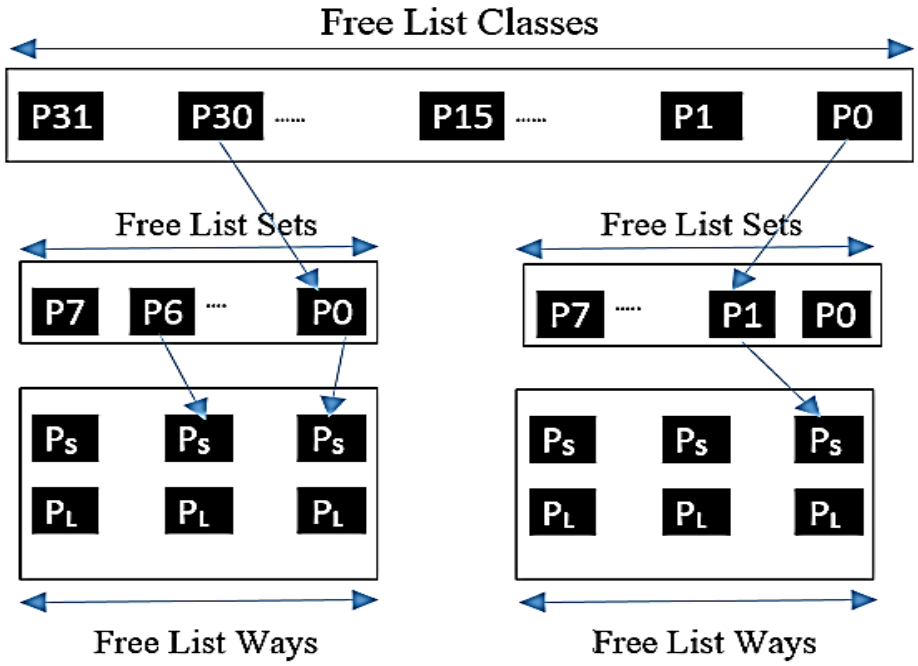

Figure 2.8: Structure of Smart Memory Allocator [114] [119]

This algorithm uses memory object life scope; it can easily allocate a memory block from either short-lived or long-lived memory pool [9]. Therefore, it can have best response time with lower fragmentation. It is implemented with a lookup table and hierarchical bitmaps which is an improved version of the multilevel segregated mechanism.

2.4 Summary

Dynamic memory allocation algorithms are important for the applications of this computing era. These, all allocators use all available memory resources more efficiently. The recent general-purpose operating systems offer the dynamic memory allocation, but since they are not enhanced, they may lead to issues like costly searching of memory block and fragmentation.

In last five decades, many memory management algorithms have been proposed. Each algorithm has its own pros and cons, but they try to decrease fragmentations or lower the execution time.

Table 2.1 shows the worst-case time complexity of the allocators as well as the fragmentation that occurs is acceptable or not along with the provision to support the NUMA architecture. The parameters in the table are: n is the heap size and m is the tree's depth. In context of real-time systems, the segregated Fit, tcmalloc and Half-fit are the only algorithms which satisfactorily meet the desired objectives and also provide a constant execution time.

Table 2.1: Summary of existing Memory Allocators of RTOS

Memory Management Algorithms	Parameters		
	Allocation	**Fragmentation**	**NUMA Support**
Sequential fit	O(n)	Acceptable	No
Buddy System	O($\log_n 2$)	Unacceptable	No
Doug Lea(DLmalloc)	O(m)	Acceptable	No
Tcmalloc	O(1)	Acceptable	Yes
Hoard	O(n)	Acceptable	No
Half-fit	O(1)	Unacceptable	No
TLSF	O(1)	Acceptable	No
Smart Memory Allocator	O($\log_n 2$)	Unacceptable	No

Furthermore, the buddy allocators having O($\log_2$ n) time complexity is affordable for real-time systems, but they cause large fragmentation. The other allocators are not acceptable for the real-time systems because of their boundless execution time. Moreover, only tcmalloc allocator provides support to NUMA architecture system.

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

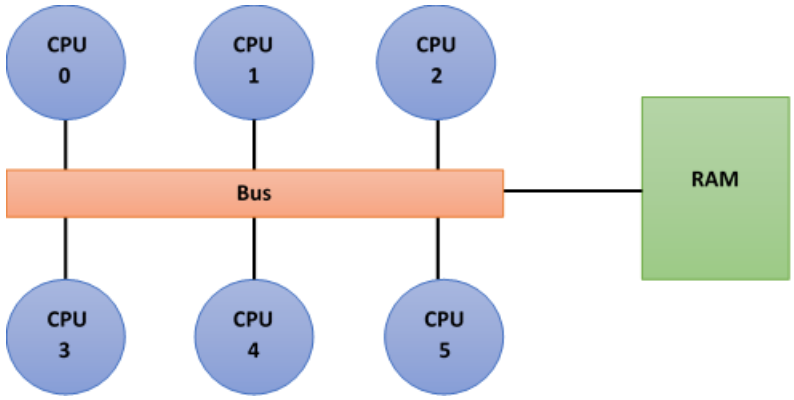

Figure 3.1: SMP Architecture

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.

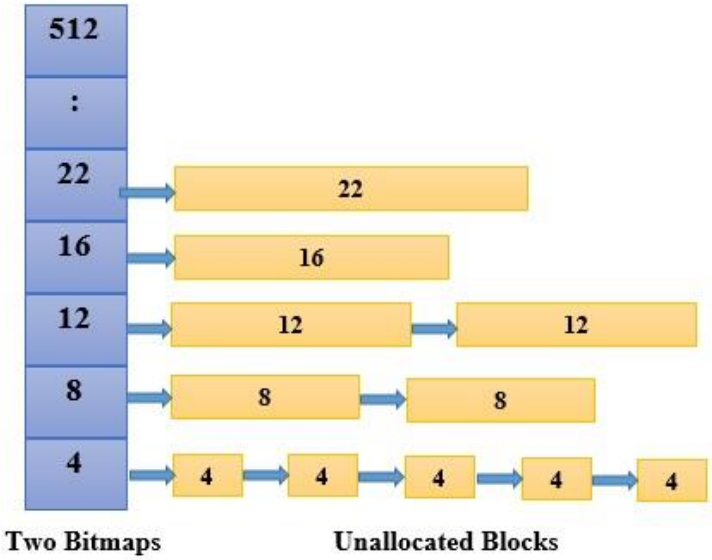

Figure 3.2: DmRT Structure for Small Block Allocation

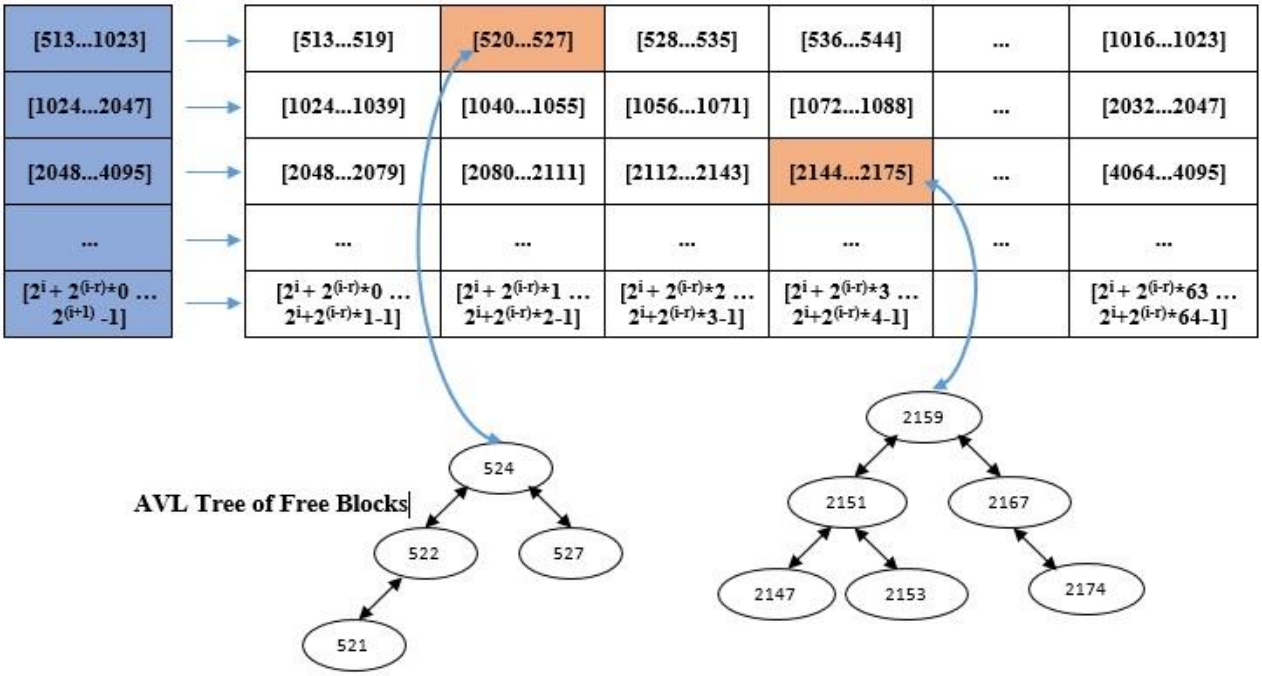

Figure 3.3: DmRT Structure for Normal Block Allocation

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

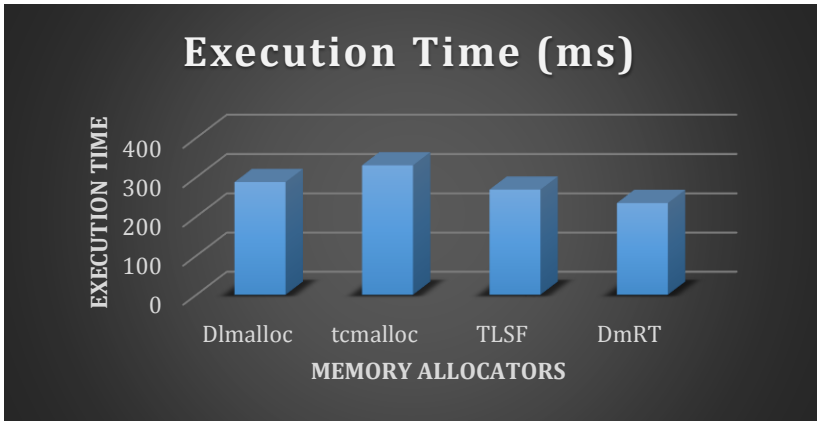

Figure 3.4: Execution time of Memory allocators in Best case

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.

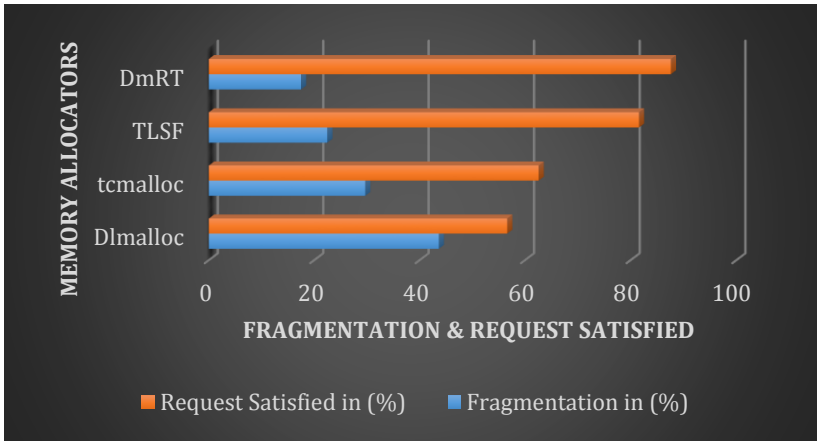

Figure 3.5: Fragmentation & Request Satisfied of Memory allocators in Best case

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

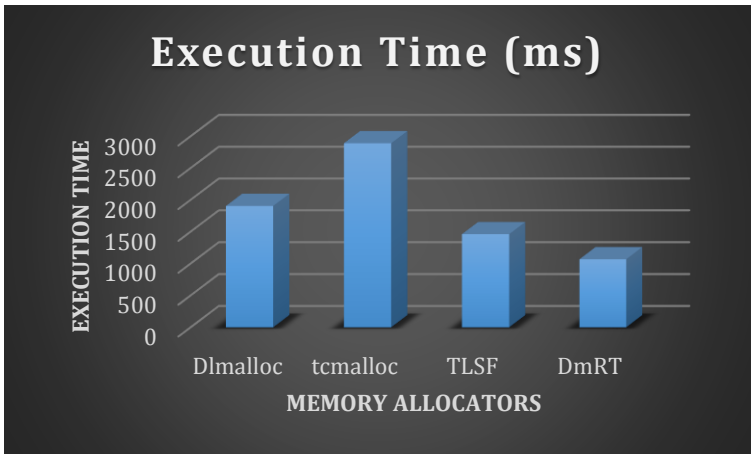

Figure 3.6: Execution time of Memory allocators in Average case

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.

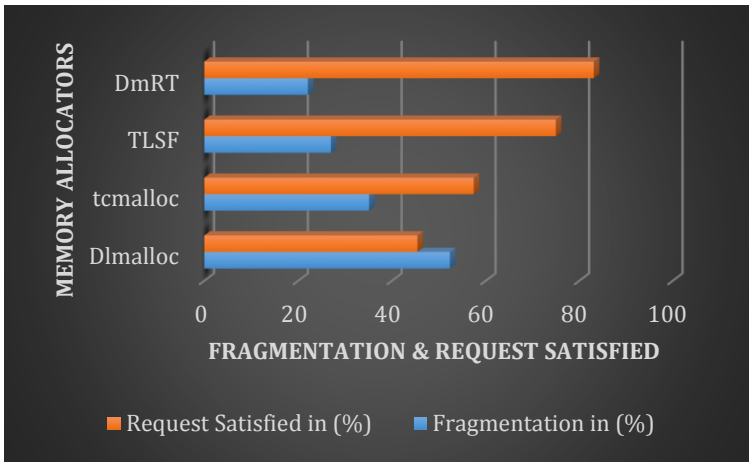

Figure 3.7: Fragmentation & Request Satisfied of Memory allocators in Average case

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

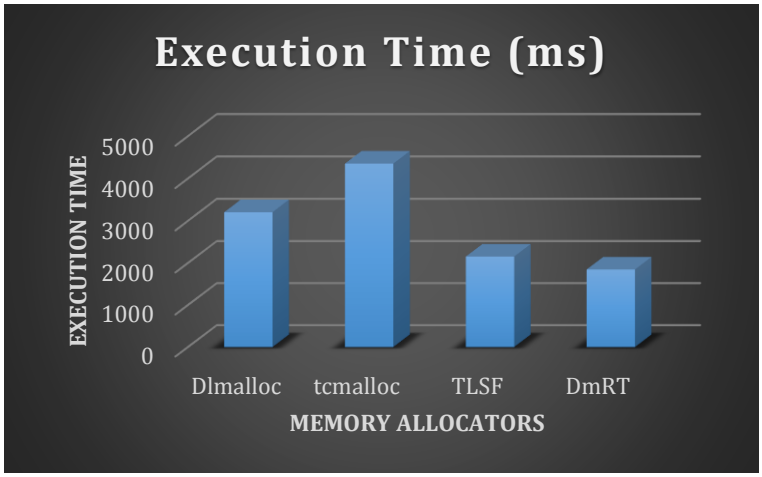

Figure 3.8: Execution time of Memory allocators in Worst case

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

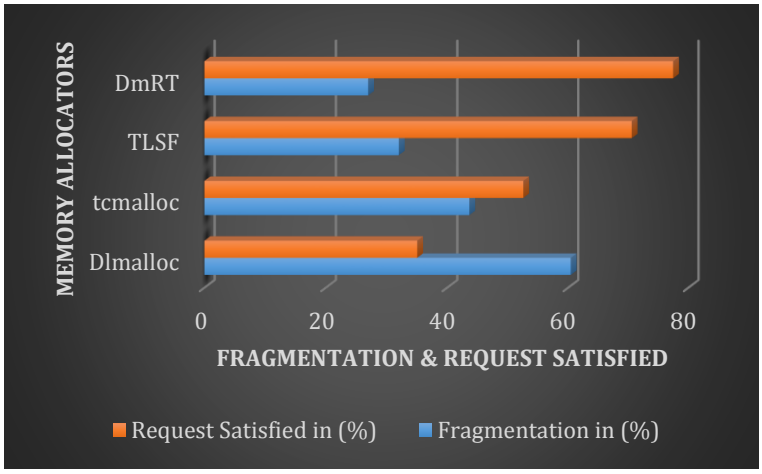

Figure 3.9: Fragmentation & Request Satisfied of Memory allocators in Worst case

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.

Chapter 4
DmRT for NUMA

In this chapter, the dynamic memory allocator called DmRT for Non-uniform memory access (NUMA) architecture will be discussed. As such, there is no change in strategies, policies, and mechanisms which have been used in DmRT for SMP but one more strategy has been introduced to find out the remote memory.

All the design principals such as strategies, policies, and mechanisms are briefly explained first, then, the method to find out remote memory for NUMA with its results with different test cases has been discussed.

4.1 Design Principals

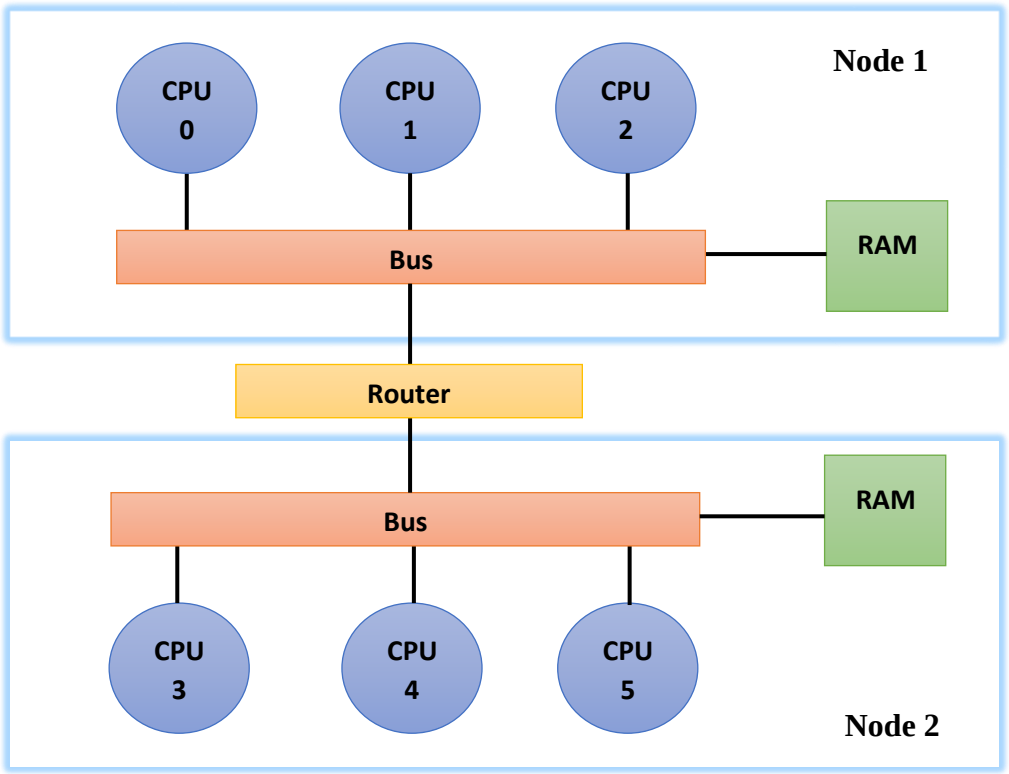

Figure 4.1: NUMA Architecture

NUMA (Non-uniform memory access) is one type of design for computer memory, used in multiprocessing, in which the access time of memory depends on the memory location relative to the processor. In NUMA architecture, a processor can read/write faster from its local memory than the non-local one like the local memory of other processor or shared memory between processors. The benefits of NUMA are limited to particular workloads, mainly on the servers where the data is often associated strongly with certain tasks or users.

In a high-performance computing generation, NUMA is the future of SMP, but its architecture is more complex than the Symmetric multiprocessors. Figure 4.1 shows simple NUMA architecture with two nodes each containing more than one processor and all are sharing a common memory. However, they may have their local memory as well. There are other complex architectures also available which can have 4 or 8 nodes. 4 nodes architecture has been considered for this proposed memory allocator, however, it is very easy to scale it up to 8 nodes.

In this chapter, a dynamic memory allocator DmRT for NUMA has been proposed and its design principals, pseudo code and results with different test cases have been discussed.

4.1.1 Strategy for selecting Remote Memory

The memory allocator which can work on NUMA based architecture for the real-time operating system has been displayed in figure 4.2. There are total four nodes where each node has two processors and each processor within a node are connected with a bus. These all nodes are connected with shared memory with their own local (private) memory.

When any processor requires a memory block, it, first, checks into its local memory; if the required memory block is available, then it will allocate the same block from the local memory. Now, if it fails in finding a local memory, then, it tries to access it from the shared memory. Here, if the memory block is available in shared memory, then it will be allocated. If the block is not found in shared memory also, then, it will ask another processor which is lightly loaded in terms of memory. So, the next step is to find the lightly loaded processor.

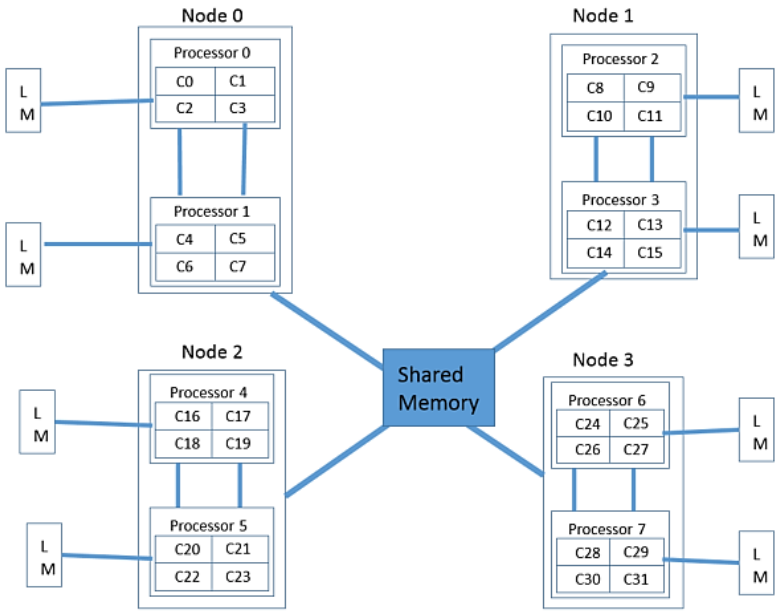

Figure 4.2: Complex NUMA Structure (4 Nodes)

According to memory utilization, each processor can be categorized into four categories as under:

1) Ideal

2) Heavily Loaded

3) Normal Loaded

4) Lightly Loaded

The first step is to calculate the average load for memory utilization of all processors using the following equation [85] [111].

$$Mem_{u\_avg} = \frac{Mem_{u1} + Mem_{u2} + Mem_{u3} + \cdots + Mem_{un}}{n} \quad 4.1$$

The second step is to find the upper and lower threshold value for memory utilization using the following equation [85] [111].

$$T_U = U \times Mem_{u\_avg}$$
$$T_L = L \times Mem_{u\_avg} \quad 4.2$$

Where, T_U = upper limit of threshold,

T_L = lower limit of threshold,

U and L are constants. (U >1 and L< 1)

In the proposed algorithm DmRT, U and L are set to 1.3 and 0.7 respectively, which is interpreted like this. If the memory utilization of a processor is 30% above the Mem_{u_avg}, the processor is said to be heavily loaded and if the memory utilization of a processor is 70% of the Mem_{u_avg}, it is said to be a lightly loaded processor; otherwise, it is considered as a normally loaded processor.

For example, Mem_{u_avg} is 50, then heavily loaded node can be considered as 30 % above Mem_{u_avg} (50 + 30 % of 50 = 65) i.e. its value is 65. And lightly loaded node can be considered as 70 % of Mem_{u_avg} (70% of 50 = 35) i.e. its value is 35.

Hence, Light weight Memory <= 35% Memory utilization

Heavy weight Memory >= 65% Memory utilization

Average (Normal) weight Memory > 35% to < 65% Memory utilization

Ideal Memory < 10% Memory utilization. Ideal Memory is one of the categories for DmRT memory structure in which there is no utilization or memory is only utilized for startup process of processor [85]. The next step is to select the appropriate processor's memory for allocating the memory. Here, which memory will be used to allocate the memory block is decided.

4.1.2 Multiple strategies for different sizes of blocks

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

i. A small block whose size of memory block is < 512 bytes
ii. A normal block whose size of memory block is < threshold (Some predefined size, here, 2Mb)
iii. A large block whose size for request exceeds the threshold or some predefined size

4.1.3 Search Policies and Mechanisms

After defining the strategies, the following policies and mechanisms have been considered to implement these strategies.

I. First, for Small blocks, the best-fit policy is used and implemented using exact-fit mechanisms to reduce the fragmentation in small sizes of blocks due to rounding up the request size of the memory block.

II. Second, for Normal blocks, the good-fit policy is used implemented using segregated lists, which in turn uses an array of unallocated block lists**.**

III. Finally, for Large blocks, the worst-fit policy is used.

4.1.4 Arrangement of blocks

The arrangement of the block is same as discussed in Symmetric Multiprocessing architecture (SMP) in section 3.1.3.

4.2 Pseudo code of Proposed Allocator for NUMA: DmRT

Pseudo code for the arrangement of the block and allocating a memory block is the same as what has been discussed in SMP in section 3.2. But pseudo code to find remote node is mentioned below:

4.2.1 Remote Node Search

$\boldsymbol{Mem_{ui}}$= Memory utilization of i^{th} node

$\boldsymbol{Mem_{us}}$= Memory utilization of self node

$\boldsymbol{T_U}$ = upper limit of threshold,

$\boldsymbol{T_L}$ = lower limit of threshold,

$\boldsymbol{U}$ and $\boldsymbol{L}$ are constants. ($\boldsymbol{U}$>1 and $\boldsymbol{L}$ < 1)

Here $\boldsymbol{U}$ = 1.3 and $\boldsymbol{L}$ = 0.7

$\boldsymbol{Mem_{u\_avg}}$= Average Memory utilization of all available nodes including shared memory

BEGIN

Calculate Memory Utilization of each node

Find Average Memory Utilization.

$$Mem_{u\_avg} = \frac{Mem_{u1} + Mem_{u2} + Mem_{u3} + \cdots + Mem_{un}}{n}$$

Find Upper and Lower Threshold Values

$T_U = H \times Mem_{u\_avg}$

$T_L = L \times Mem_{u\_avg}$

Sort node in ascending order of utilization

Categorize each node Ideal, lightly loaded, Average Loaded and Heavily Loaded

IF Self node is Ideal Node THEN

    Use local memory of self-node for utilization.

ELSE IF Ideal Node is available THEN

    Use local memory of ideal node for utilization

ELSE IF Lightly Loaded Node is available THEN

    IF Memory utilization of Lightly Loaded Node <= Mem_{us} THEN

        Use local memory of Lightly Loaded node for utilization.

    ELSE

        Use local memory of self-node for utilization.

    ENDIF

ELSE IF Average Loaded Node is available THEN

    IF Memory utilization of Average Loaded Node <= Mem_{us} THEN

        Use local memory of Average Loaded node for utilization.

    ELSE

        Use local memory of self-node for utilization.

    ENDIF

ELSE

Use local memory of self-node for utilization.

ENDIF

END

4.3 Results

Here, four different test cases have been considered for NUMA which are mentioned below:

Case 1: Existing allocators from Local and DmRT follows Local → Shared → Ideal

Existing allocators (Dlmalloc, tcmalloc and TLSF) allocate memory block from Local Memory while DmRT, first, tries to allocate a block from Local Memory. If it fails to do so, then, it attempts the same from Shared memory. If it fails, here too, it tries to find ideal memory and allocates a block from it. As DmRT tries to find a memory block from three different types of memory, its execution time will be more than the other allocators, but it provides a consistent execution time. Along with that, it satisfies the maximum number of the requests as well as creates less fragmentation due to proposed allocator (DmRT) structure.

Case 2: Existing allocators from Local and DmRT from Ideal

In this case, all existing allocators allocate memory block from Local memory only, while DmRT, first, finds the ideal memory and it will allocate a memory block from it. Here, existing allocators allocate the blocks from local memory only that is why they have less number of request satisfactions while DmRT has a maximum number of request satisfaction. The other parameters can also perform the best due to its structure.

Case 3: Existing allocators and DmRT both from Ideal

In this case, the existing allocators and DmRT, both first find ideal memory and allocate a block from it. As existing allocators and DmRT, both allocate memory from the ideal memory, the execution time will be almost same but the DmRT will have a maximum number of request satisfied and less fragmentation as an added advantage.

Case 4: Existing allocators and DmRT follow Local → Shared → Ideal

In this case, the existing allocators and DmRT, both first, try to allocate memory block from local and if they fail, they try to allocate the same block from shared memory. If they fail in shared memory too, then, they find ideal memory and allocate same memory block from it. Though both, the existing allocators and DmRT, follow the same path from allocating memory, the proposed allocator, DmRT, defeats all of them from all aspects.

In each case, there are three different test categories have been used.

a. Best case, i.e. the test has been carried out for 100 memory blocks request.
b. Average case, i.e. the test has been carried out for 1000 memory blocks request.
c. Worst case, i.e. the test has been carried out for 2000 memory blocks request.

There are three main parameters which have been 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: The number of Requests Satisfied should be as high as possible.

For comparisons, the following four memory management algorithms have been used.

a. Dlmalloc
b. tcmalloc
c. TLSF
d. DmRT

All the tests have been carried out on MemSimRT which will be discussed in detail in chapter 5. Each result mentioned here is the average of 100 attempts. The 100 attempts of each case have been mentioned in the Annexure I.

4.3.1 Existing from Local and DmRT follows Local → Shared → Ideal

1. Average of 100 attempts (Best Case, i.e., for 100 memory block requests)

Table 4.1: Existing from Local and DmRT follows Local → Shared → Ideal (Best Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	DmRT
Execution Time (ms)	326.2426	410.8068	290.2026	374.3901
Fragmentation in (%)	43.5037	36.6849	21.5874	10.5141
Request Satisfied in (%)	57.5068	62.3301	76.6088	94.6614

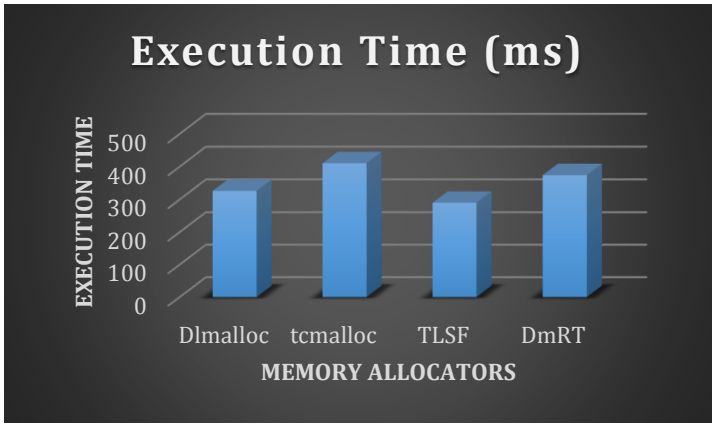

Figure 4.3: Execution time of Memory allocators in Best case

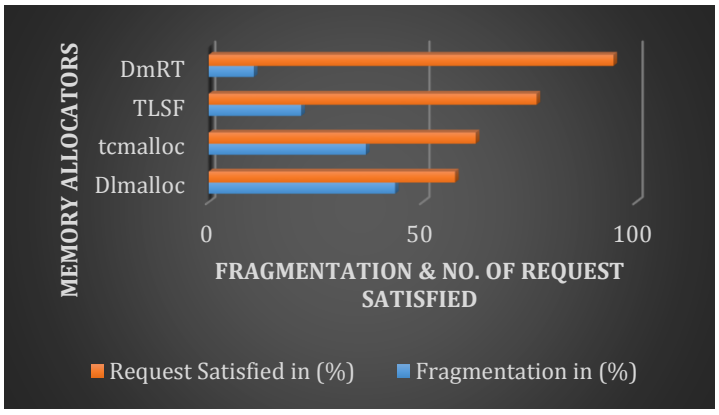

Figure 4.4: Fragmentation & Request Satisfied of Memory allocators in Best case

As shown in Figure 4.1, the DmRT takes **more execution time** than the DLmalloc and the TLSF because the DmRT follows the path of memory allocation from Local to Shared to Ideal, whereas and the existing algorithms allocate from local memory only. In this scenario also, tcmalloc takes maximum execution time.

As shown in figure 4.2, the DmRT **satisfies the maximum requests** and has the **lowest fragmentation** as compared to all other dynamic memory allocators, and Dlmalloc performs exactly opposite to it.

2. Average of 100 attempts (Average Case, i.e., for 1000 memory block requests)

Table 4.2: Existing from Local and DmRT follows Local → Shared → Ideal (Average Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	Proposed
Execution Time (ms)	2013.324	2988.474	1522.335	2303.212
Fragmentation in (%)	52.5491	44.4944	29.5867	15.6241
Request Satisfied in (%)	45.2403	54.2546	65.6095	87.4181

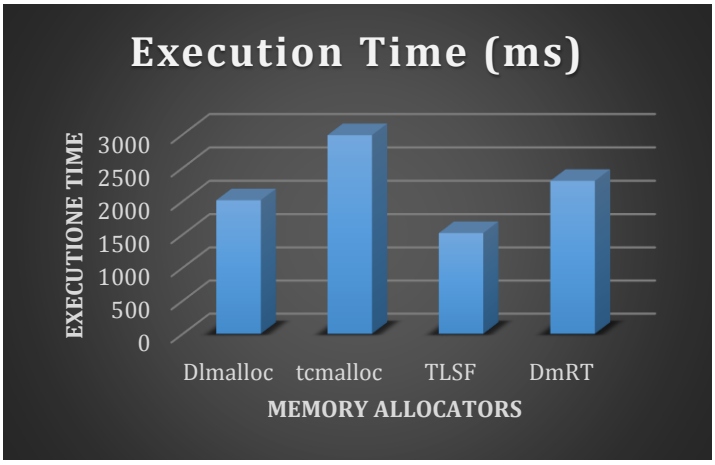

Figure 4.5: Execution time of Memory allocators in Average case

As shown in Figure 4.5, even for 1000 blocks requests, the DmRT takes **more execution time** than the DLmalloc and TLSF, as the DmRT follows the path of memory allocation from Local to shared to Ideal and the existing ones allocate from the local memory only. Here also, tcmalloc takes the maximum execution time.

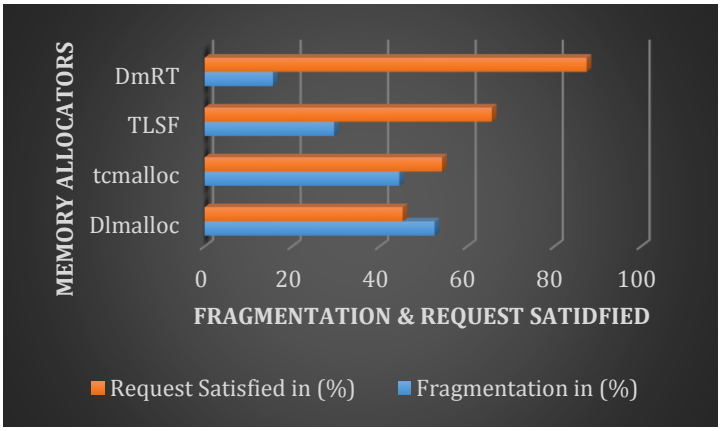

Figure 4.6: Fragmentation & Request Satisfied of Memory allocators in Average case

As shown in figure 4.6, the DmRT **satisfies the maximum requests** and has the **lowest fragmentation** with reference to all other dynamic memory allocators, and Dlmalloc performs exactly opposite to DmRT. Dlmalloc causes a **higher amount of Fragmentation than satisfying number of requests.**

3. Average of 100 attempts (Worst Case, i.e., for 2000 memory block requests)

Table 4.3: Existing from Local and DmRT follows Local → Shared → Ideal (Worst Case)

Algorithms / Parameters	Dlmalloc	Tcmalloc	TLSF	Proposed
Execution Time (ms)	3202.561	4348.651	2049.025	3361.854
Fragmentation in (%)	61.4645	43.3702	36.5828	20.5572
Request Satisfied in (%)	35.006	50.0315	60.055	82.3775

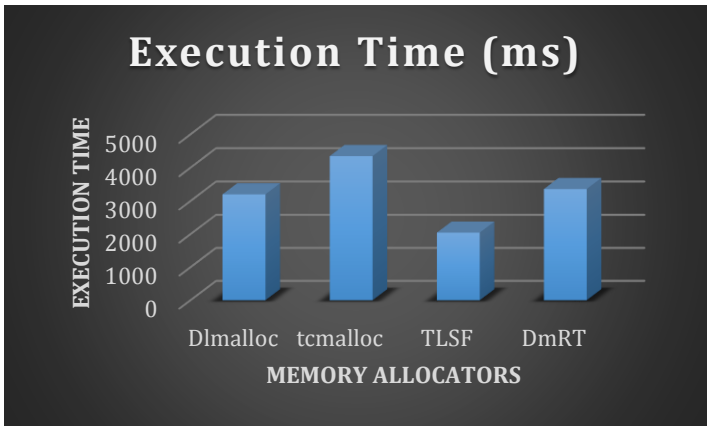

Figure 4.7: Execution time of Memory allocators in Worst case

As shown in Figure 4.7, for 2000 blocks requests also, the DmRT takes **more execution time** than DLmalloc and TLSF. Here too, it follows the same path of memory allocation from Local to shared to Ideal. The existing allocators allocate the same from local memory only yet the tcmalloc takes the maximum execution time.

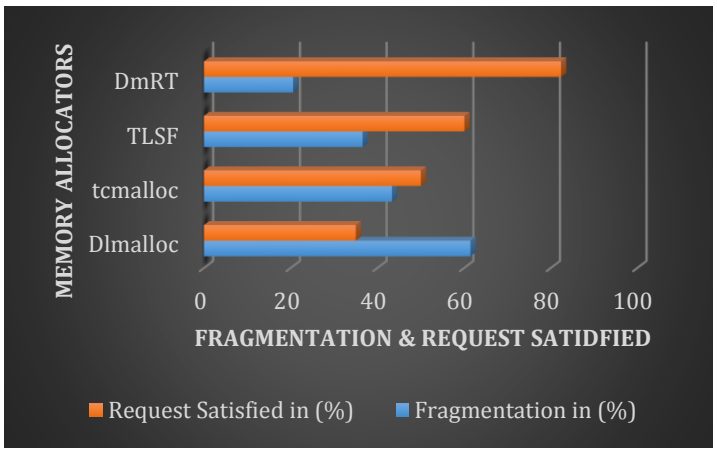

Figure 4.8: Fragmentation & Request Satisfied of Memory allocators in Worst case

As shown in figure 4.8, the DmRT **satisfies maximum requests** and shows the **lowest fragmentation** in comparison with all other dynamic memory allocators. The Dlmalloc is exactly opposite to it causing **higher Fragmentation than satisfying the number of requests.**

4.3.2 Existing From Local and DmRT from Ideal

1. Average of 100 attempts (Best Case, i.e., for 100 memory block requests)

Table 4.4: Existing From Local and DmRT from Ideal (Best Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	Proposed
Execution Time (ms)	338.0589	420.5202	296.4418	245.4583
Fragmentation in (%)	45.2763	33.6326	24.0237	15.4697
Request Satisfied in (%)	57.7885	64.1904	76.9458	89.9526

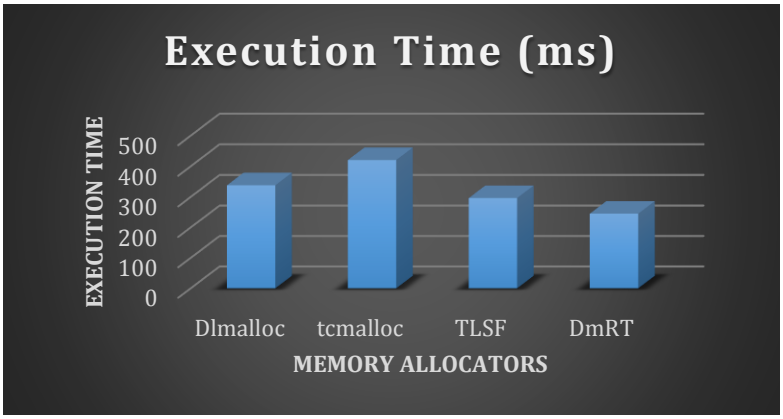

Figure 4.9: Execution time of Memory allocators in Best case

As shown in figure 4.9, the DmRT takes **minimum execution time as** compared to all other dynamic memory allocators, whereas tcmalloc takes the maximum execution time.

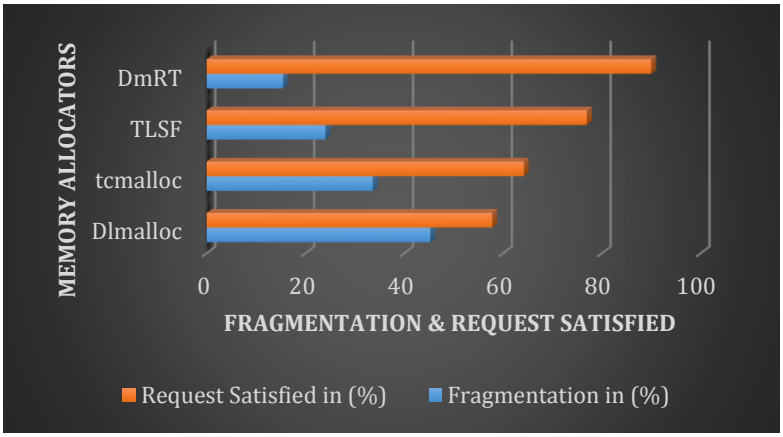

Figure 4.10: Fragmentation & Request Satisfied of Memory allocators in Best case

As shown in figure 4.10, the DmRT **satisfies the maximum requests** and has the **lowest fragmentation due to** allocation of memory from Ideal memory as compared to all other dynamic memory allocators. The Dlmalloc has the exactly opposite performance.

2. Average of 100 attempts (Average Case, i.e., for 1000 memory block requests)

Table 4.5: Existing From Local and DmRT from Ideal (Average Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	Proposed
Execution Time (ms)	2054.716	2995.241	1539.964	1115.835
Fragmentation in (%)	61.1009	53.0719	37.7599	26.0615
Request Satisfied in (%)	34.3767	50.9882	61.8621	73.6385

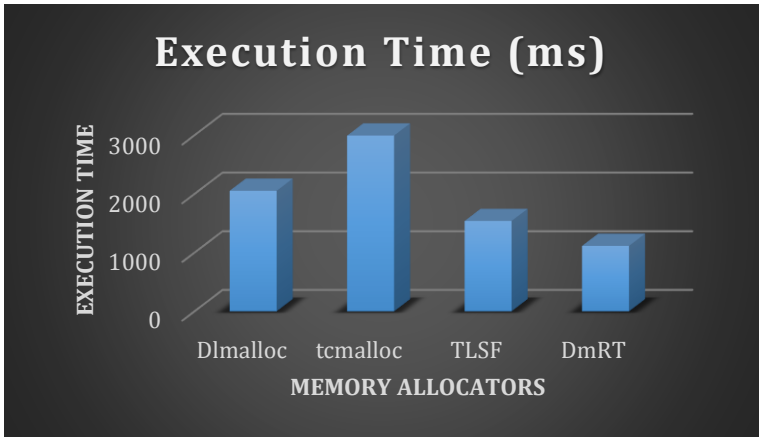

Figure 4.11: Execution time of Memory allocators in Average case

As shown in figure 4.11, the DmRT takes the **minimum execution time among all** dynamic memory allocators, and tcmalloc takes the maximum execution time.

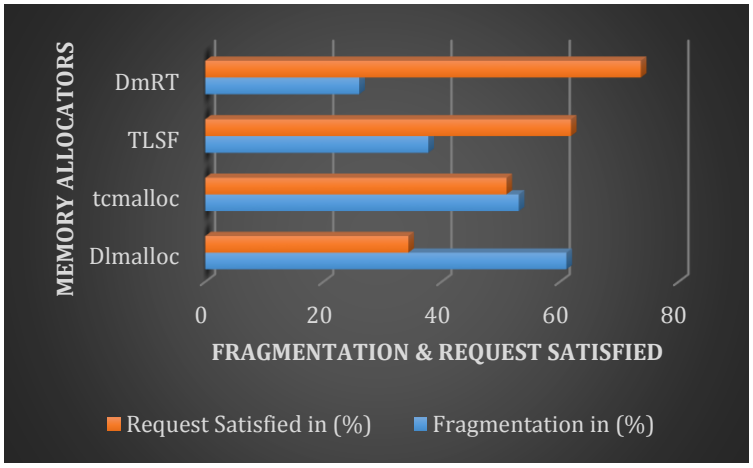

Figure 4.12: Fragmentation & Request Satisfied of Memory allocators in Average case

As shown in figure 4.12, the DmRT **satisfies the maximum requests** and shows the **lowest fragmentation** among all other dynamic memory allocators, and performance point of view, the Dlmalloc is exactly opposite. It causes a **higher amount of Fragmentation than satisfying the number of requests.**

3. Average of 100 attempts (Worst Case, i.e., for 2000 memory block requests)

Table 4.6: Existing From Local and DmRT from Ideal (Worst Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	Proposed
Execution Time (ms)	3283.047	4288.159	2064.704	1785.676
Fragmentation in (%)	60.4389	43.6719	32.0433	26.9948
Request Satisfied in (%)	34.9105	49.962	59.887	74.1195

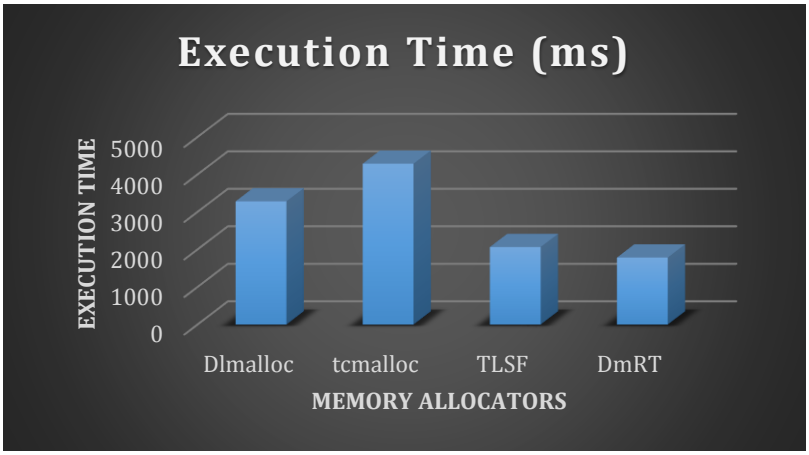

Figure 4.13: Execution time of Memory allocators in Worst case

As shown in figure 4.13, the DmRT takes **minimum execution time as** compared to all other dynamic memory allocators and the tcmalloc takes the maximum execution time.

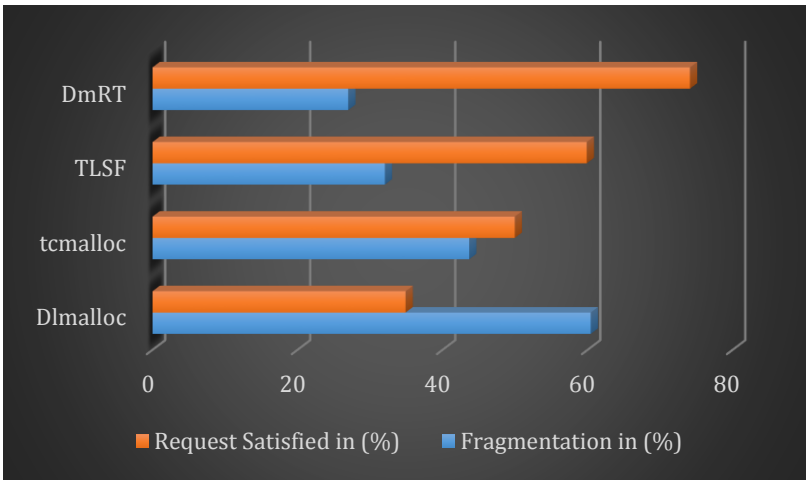

Figure 4.14: Fragmentation & Request Satisfied of Memory allocators in Worst case

As shown in figure 4.14, the DmRT **satisfies the maximum requests** and has the **lowest fragmentation among** all. The Dlmalloc performs exactly opposite to the DmRT with a **higher amount of Fragmentation than satisfying the number of requests.**

4.3.3 Existing and DmRT Both from Ideal

1. Average of 100 attempts (Best Case, i.e., for 100 memory block requests)

Table 4.7: Existing and DmRT Both from Ideal (Best Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	Proposed
Execution Time (ms)	374.8572	444.5905	319.6948	249.566
Fragmentation in (%)	35.2178	26.3902	19.2872	14.5781
Request Satisfied in (%)	67.5358	72.1999	83.1096	89.1851

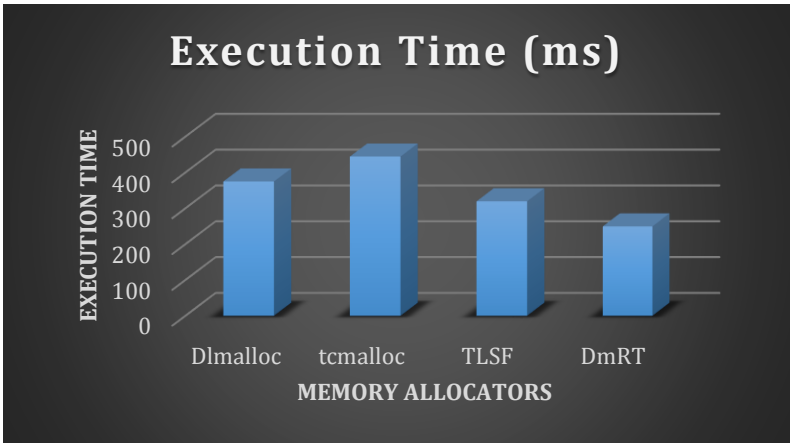

Figure 4.15: Execution time of Memory allocators in Best case

As shown in figure 4.15, the DmRT takes **minimum execution time** as compared to all other dynamic memory allocators, and tcmalloc takes maximum execution time even though all allocators allocate from Ideal memory.

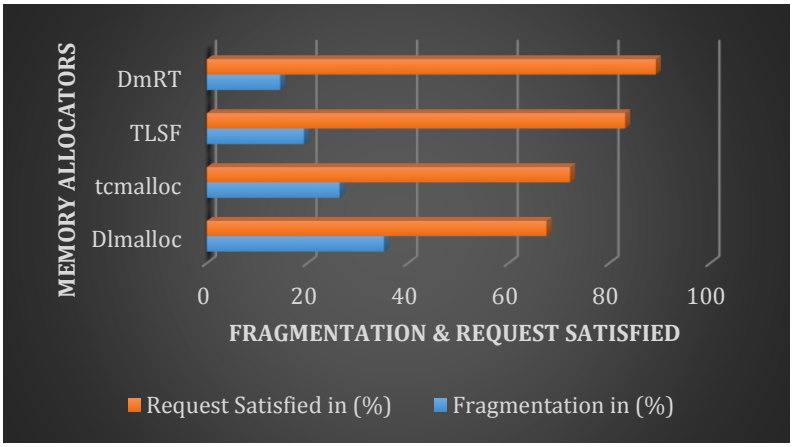

Figure 4.16: Fragmentation & Request Satisfied of Memory allocators in Best case

As shown in figure 4.16, Though all allocators are allocating memory from Ideal memory, the DmRT has the **maximum number of requests satisfied** and has the **lowest fragmentation**. The Dlmalloc is exactly opposite to it, as it has been.

2. Average of 100 attempts (Average Case, i.e., for 1000 memory block requests)

Table 4.8: Existing and DmRT Both from Ideal (Average Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	Proposed
Execution Time (ms)	2110.58	3240.527	1530.277	1149.484
Fragmentation in (%)	43.0248	35.5497	26.3408	19.7854
Request Satisfied in (%)	55.5939	64.722	73.3219	81.1776

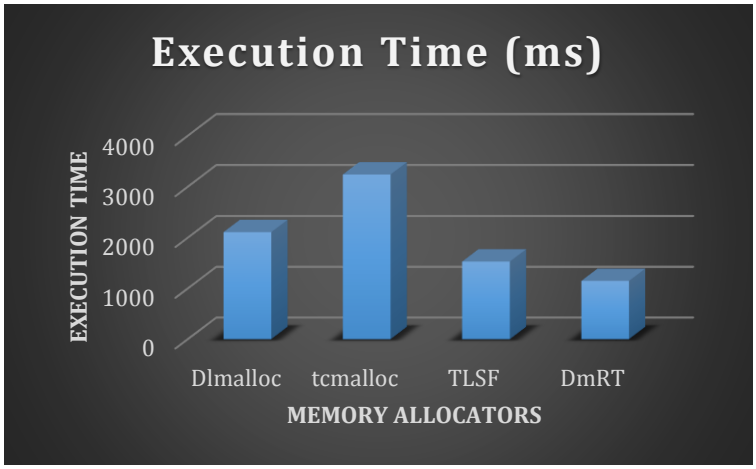

Figure 4.17: Execution time of Memory allocators in Average case

As shown in figure 4.17,

For 1000 blocks request, though all allocators are allocating memory from the Ideal memory, the DmRT takes the **minimum execution time as** compared to all other dynamic memory allocators, and tcmalloc takes the maximum.

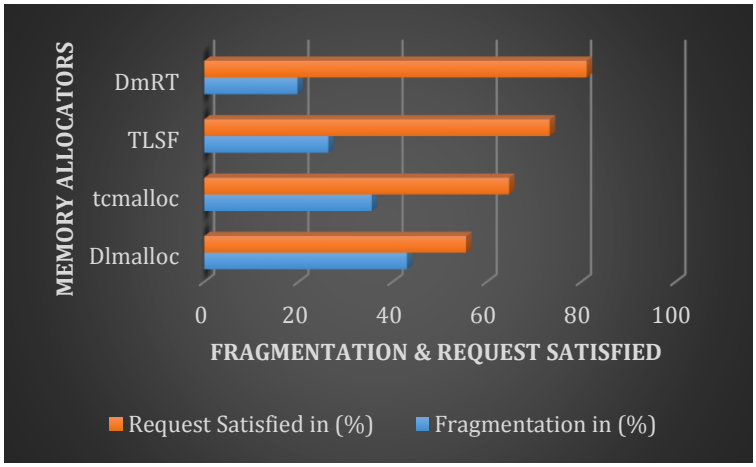

Figure 4.18: Fragmentation & Request Satisfied of Memory allocators in Average case

As shown in figure 4.18,

For 1000 blocks request, though all allocators are allocating memory from the Ideal memory, DmRT, as usual, **satisfies the maximum requests** with the lowest **fragmentation among** all. The Dlmalloc does exactly opposite to it.

3. Average of 100 attempts (Worst Case, i.e., for 2000 memory block requests)

Table 4.9: Existing and DmRT Both from Ideal (Worst Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	Proposed
Execution Time (ms)	3277.428	4467.102	2172.658	1860.321
Fragmentation in (%)	52.6015	43.6602	35.9747	25.1212
Request Satisfied in (%)	44.834	52.813	64.469	74.053

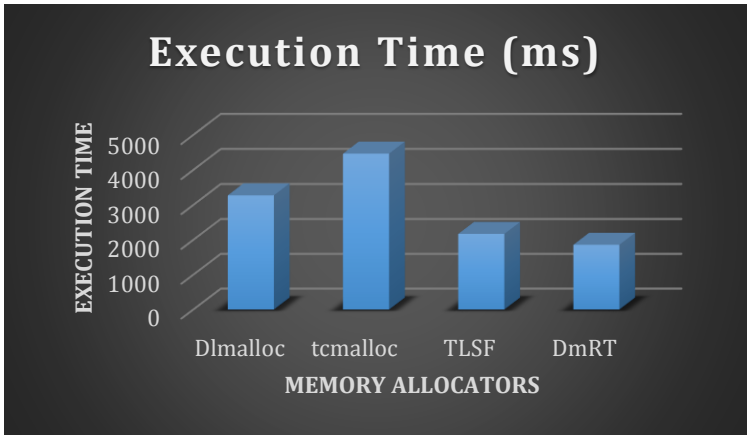

Figure 4.19: Execution time of Memory allocators in Worst case

As shown in figure 4.19, Though all allocators are allocating memory from the Ideal memory, the DmRT takes **minimum execution time among** all and the tcmalloc takes the maximum execution time.

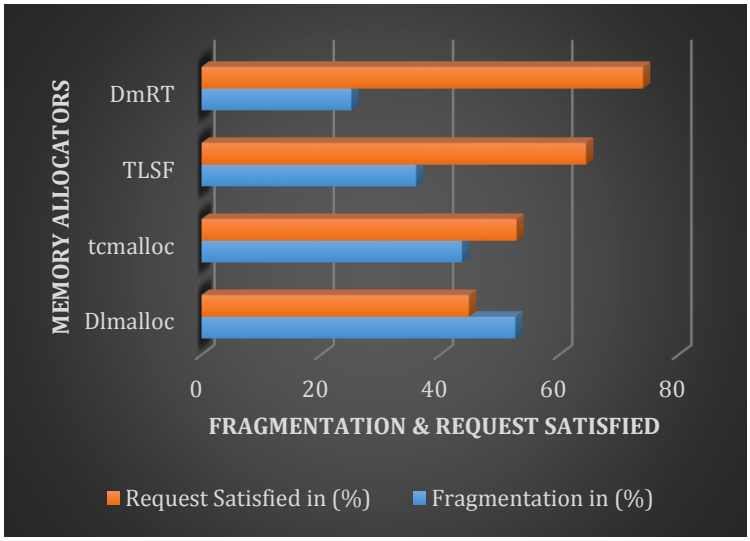

Figure 4.20: Fragmentation & Request Satisfied of Memory allocators in Worst case

As shown in figure 4.20, though all allocators are allocating memory from the Ideal memory, the DmRT has the highest **satisfying requests** and has the **lowest fragmentation**. The Dlmalloc causes a **higher amount of Fragmentation than satisfying the number of requests.**

4.3.4 Existing and DmRT follow Local → Shared → Ideal

1. Average of 100 attempts (Best Case, i.e., for 100 memory block requests)

Table 4.10: Existing and DmRT follow Local → Shared → Ideal (Best Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	Proposed
Execution Time (ms)	528.5204	636.5573	480.8449	385.8492
Fragmentation in (%)	31.4948	23.8764	17.5356	10.4119
Request Satisfied in (%)	71.7431	78.1612	86.8777	93.7361

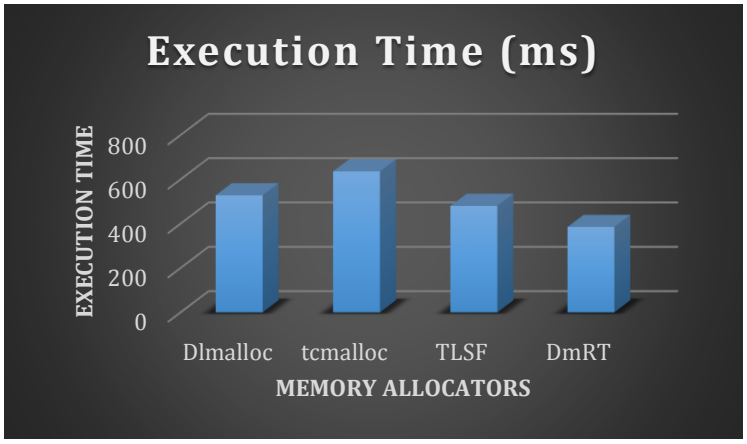

Figure 4.21: Execution time of Memory allocators in Best case

As shown in figure 4.21, Though all allocators following the path of Local, Shared and Ideal memory for allocating memory, DmRT takes **minimum execution time** compared to all other dynamic memory allocators, and tcmalloc takes maximum execution time.

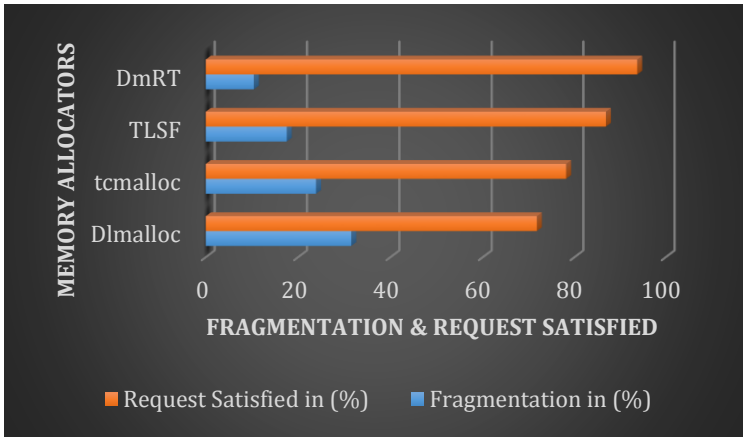

Figure 4.22: Fragmentation & Request Satisfied of Memory allocators in Best case

As shown in figure 4.22, Though all allocators following the path of Local, Shared and Ideal memory for allocating memory, the DmRT **satisfies the maximum requests** with the **lowest fragmentation** and Dlmalloc does exactly opposite to it.

2. Average of 100 attempts (Average Case, i.e., for 1000 memory block requests)

Table 4.11: Existing and DmRT follow Local → Shared → Ideal (Average Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	Proposed
Execution Time (ms)	2928.034	4166.439	2541.517	2252.019
Fragmentation in (%)	38.895	31.6255	22.913	15.0111
Request Satisfied in (%)	62.0389	70.6678	79.9134	87.5092

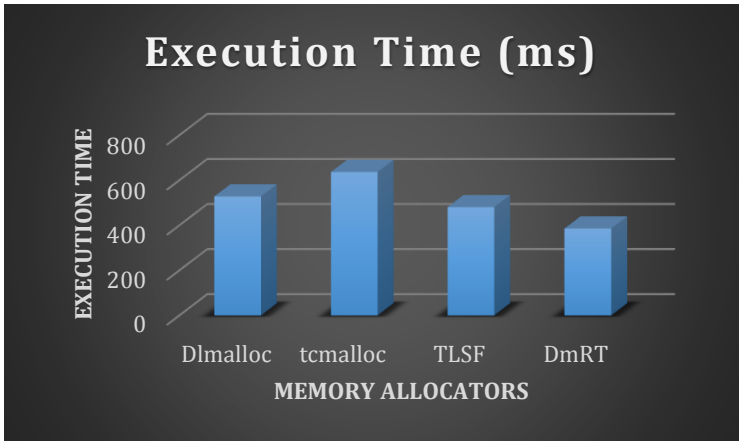

Figure 4.23: Execution time of Memory allocators in Average case

As shown in figure 4.23, for 1000 blocks allocation though all allocators following the path of Local, Shared and Ideal memory, DmRT takes **minimum execution time** as compared to all other dynamic memory allocators, and tcmalloc takes maximum execution time.

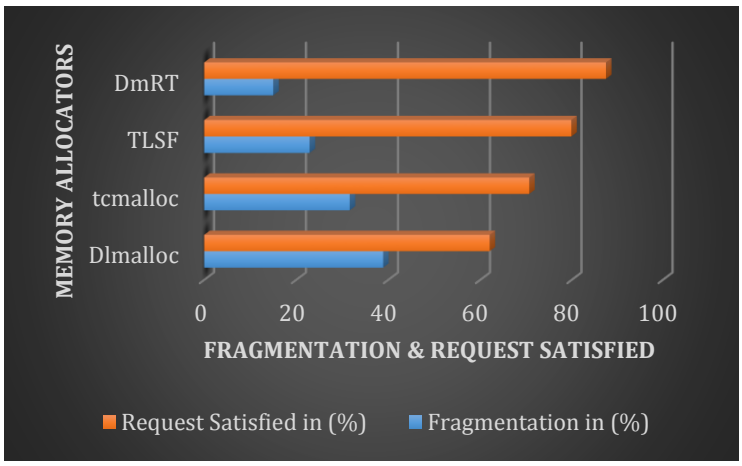

Figure 4.24: Fragmentation & Request Satisfied of Memory allocators in Average case

As shown in figure 4.24, for 1000 blocks allocation though all allocators following the path of Local, Shared and Ideal memory, DmRT **satisfies maximum requests** and has **lowest fragmentation** as compared to all other dynamic memory allocators, and Dlmalloc is exactly opposite to it.

3. Average of 100 attempts (Worst Case, i.e., for 2000 memory block requests)

Table 4.12: Existing and DmRT follow Local → Shared → Ideal (Worst Case)

Algorithms / Parameters	Dlmalloc	tcmalloc	TLSF	Proposed
Execution Time (ms)	4231.555	5107.635	3684.495	3367.702
Fragmentation in (%)	47.3835	38.2598	30.8472	19.1314
Request Satisfied in (%)	50.8095	59.9945	73.883	82.662

Figure 4.25: Execution time of Memory allocators in Worst case

As shown in figure 4.25, for 2000 blocks allocation though all allocators following the path of Local, Shared and Ideal memory, DmRT takes **minimum execution time** as compared to all other dynamic memory allocators, and tcmalloc takes maximum execution time.

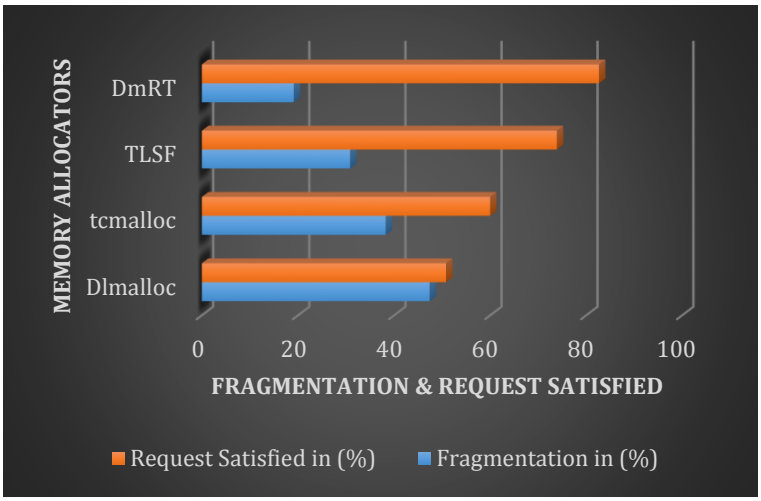

Figure 4.26: Fragmentation & Request Satisfied of Memory allocators in Worst case

As shown in figure 4.26, for 2000 blocks allocation though all allocators following the path of Local, Shared and Ideal memory, DmRT **satisfies maximum request** and has **lowest fragmentation** compare to all other dynamic memory allocators, and Dlmalloc is exactly opposite to it.

Chapter 5
Memory Management Simulator: MemSimRT

There are so many simulators available to simulate different test cases for scheduling in a real-time operating system [16] [20] [56] [73-76] like Litmus-RT, Mark3, rtsim, etc., but till date, no such simulator is available for simulating memory management algorithm for RTOS. Hence, MemSimRT has been designed to simulate various memory allocators for both SMP as well as NUMA architecture based RTOS. Its front end created in C# while back-end developed using python.

Download MemSimRT using this QRcode:

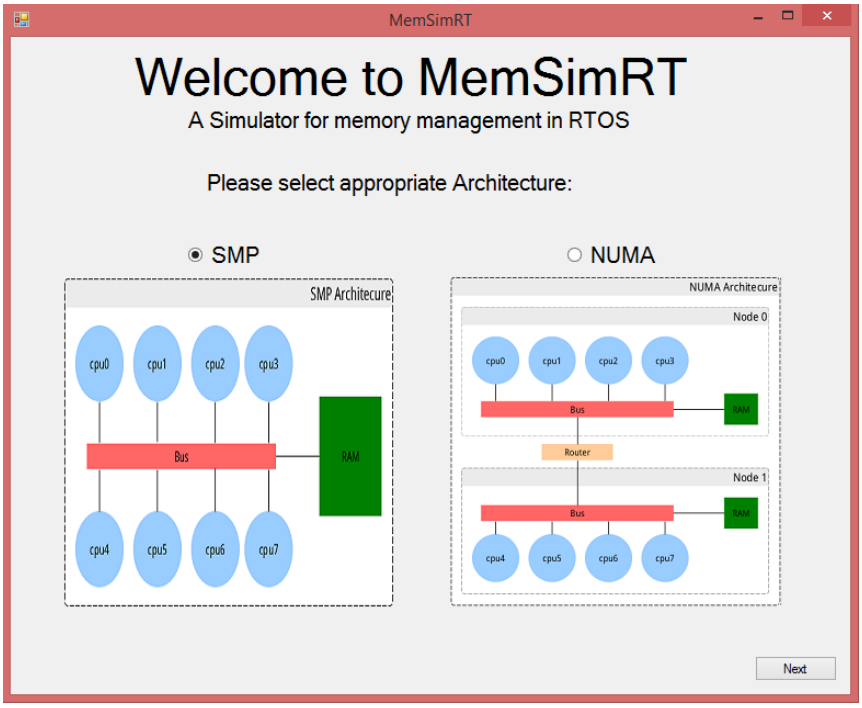

Figure 5.1: The Welcome screen of MemSimRT

Figure 5.1 shows the welcome screen of MemSimRT. So it is the Home screen of the simulator. As per our dynamic memory allocator, it has two alternatives. One is SMP, i.e., Symmetric multiprocessor and second is NUMA, i.e. Non-uniform memory access based architecture. Basically, in this simulator, NUMA is designed for eight processors, but it can be modified as per requirement by slightly changing the script.

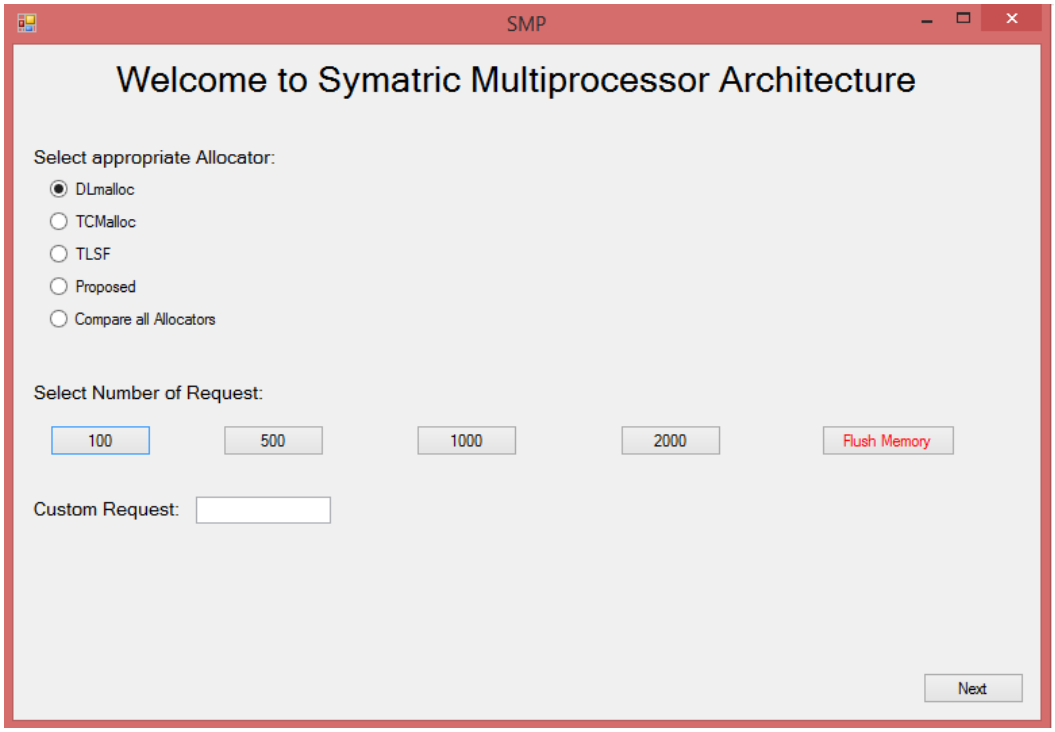

Figure 5.2: The Welcome screen of SMP

Figure 5.2 shows that, when SMP is selected, this screen will appear. One can select appropriate memory allocator for RTOS and select available number of the request. Also, memory block request other than 100, 500, 1000 and 2000 can be provided by defining the specific number in text box of the custom request and then press next button.

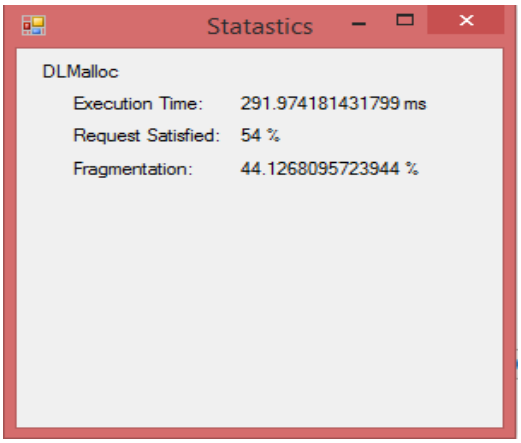

Figure 5.3 shows the result of the individual memory allocator. In result, it shows the execution time in ms (millisecond) taken by specific allocator, how many requests have been satisfied in % as well as Fragmentation generated by allocator in %.

Figure 5.3: Statistics of individual Memory Allocator

Figure 5.4: Memory block allocation as per request by DmRT

Figure 5.4 shows the exact memory allocation scenario by proposed memory allocator which is actually divided into three different categories as per algorithm for small blocks whose size less than 512 bytes, normal or medium blocks whose size is between 512 bytes to 2Mb and Large blocks whose size is beyond 2 Mb. The request denoted by asterisk (*) defines that block allocation is failed. At the end of each memory area, it shows a number of requests satisfied in %.

Also, any number of memory block request can be provided. Direct buttons are available for 100, 500, 1000 and 2000 memory block request. To provide memory block request other than 100, 500, 1000 and 2000, then define it in text box of the custom request and press next button.

Also, to allocate memory blocks in a specific area only like in small memory, medium memory or large memory can be done by selecting the specific checkbox. "*Flush memory" button* is used to flush the entire memory.

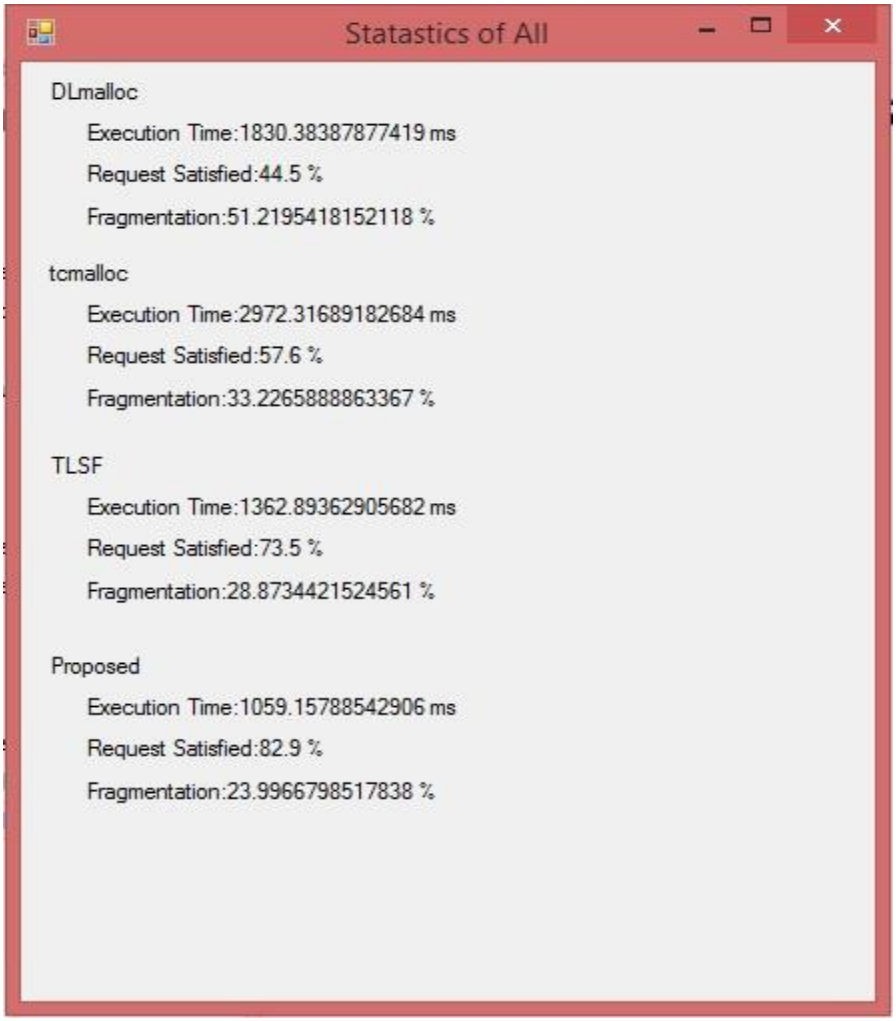

Figure 5.5: Statistics of all memory allocators

Figure 5.5 shows the statistics of all allocators in the same window. If "*compare all allocators"* is selected then result analysis for same block requests of all allocators will be shown in this window.

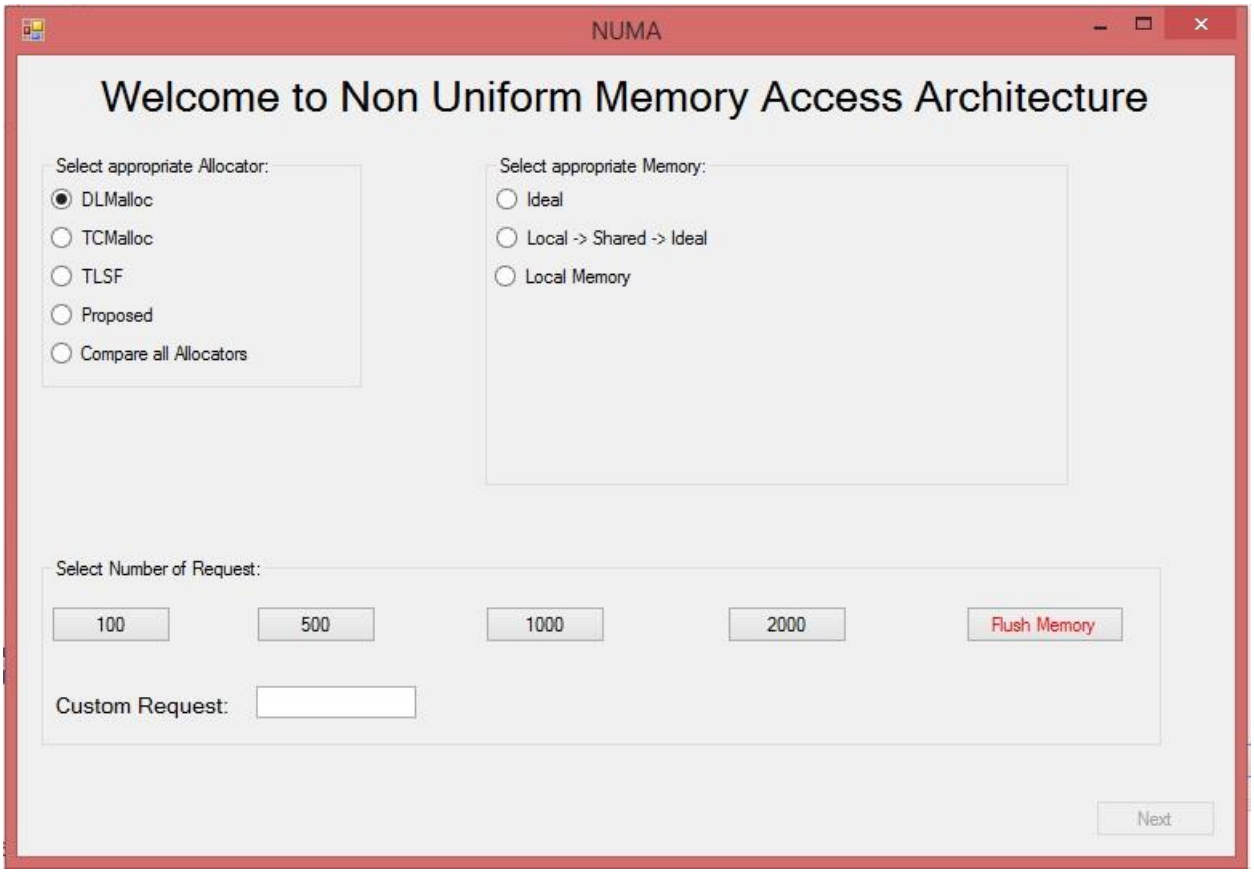

Figure 5.6: The Welcome screen of NUMA

As shown in Figure 5.6, it is the first screen for NUMA, and one can select a specific allocator, appropriate memory, and number of request from it. Memory can be *Local*, *Ideal or Local -> Shared -> Ideal*, i.e. it first tries to allocate a memory block from Local Memory if it fails then Shared memory and still if it fails then it will search from the Ideal memory. If Ideal memory is selected then first thing is to find processor which can have ideal memory that is shown in figure 5.7.

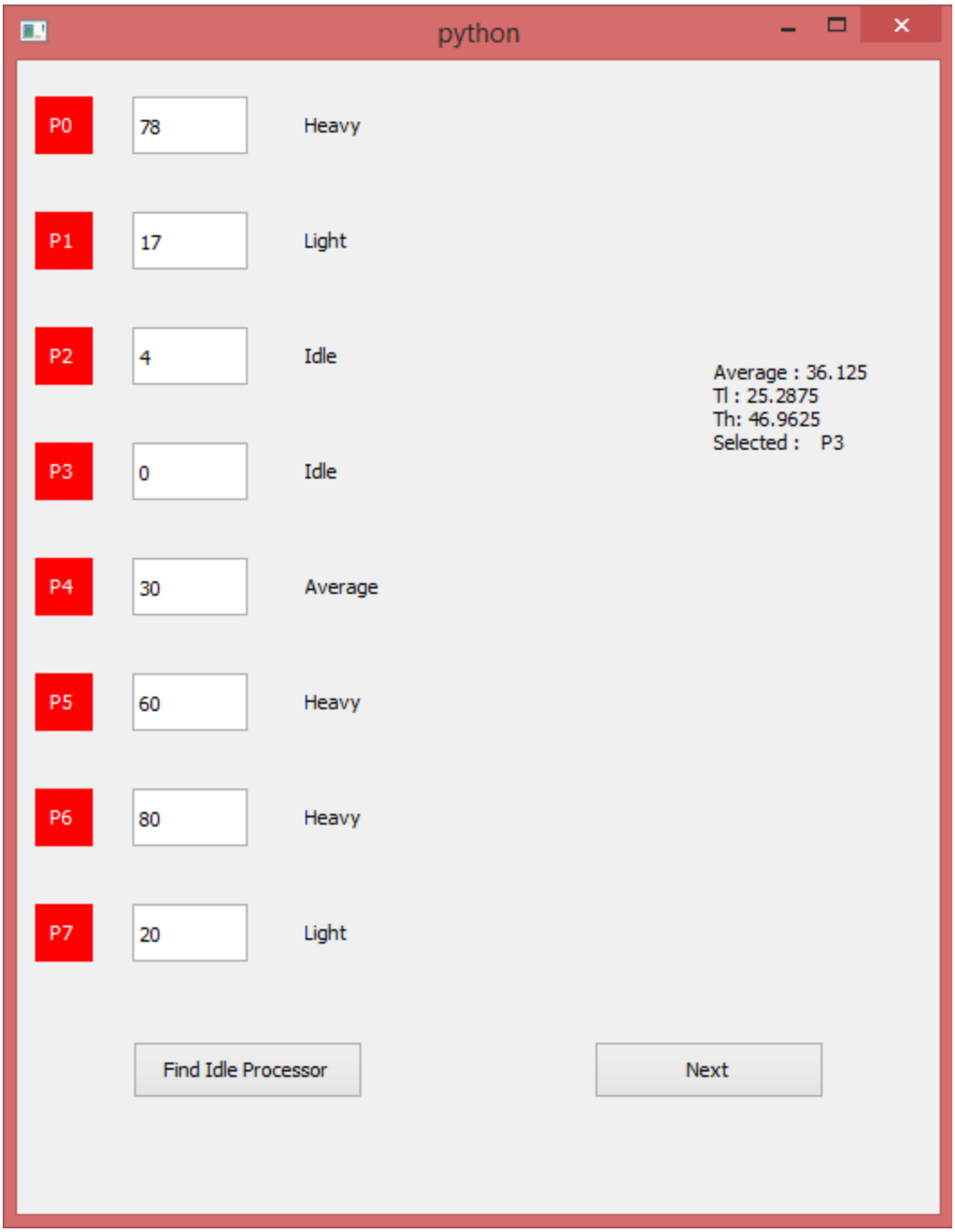

Figure 5.7: All Processors with its memory utilization in (%)

As shown in Figure 5.7, it is a NUMA architecture of eight processors (P0 to P7) and its local memory utilization is defined with it. Also, it has found the category of memory according to the threshold policy. Explicit memory utilization can also be defined by mentioning in the textfield.

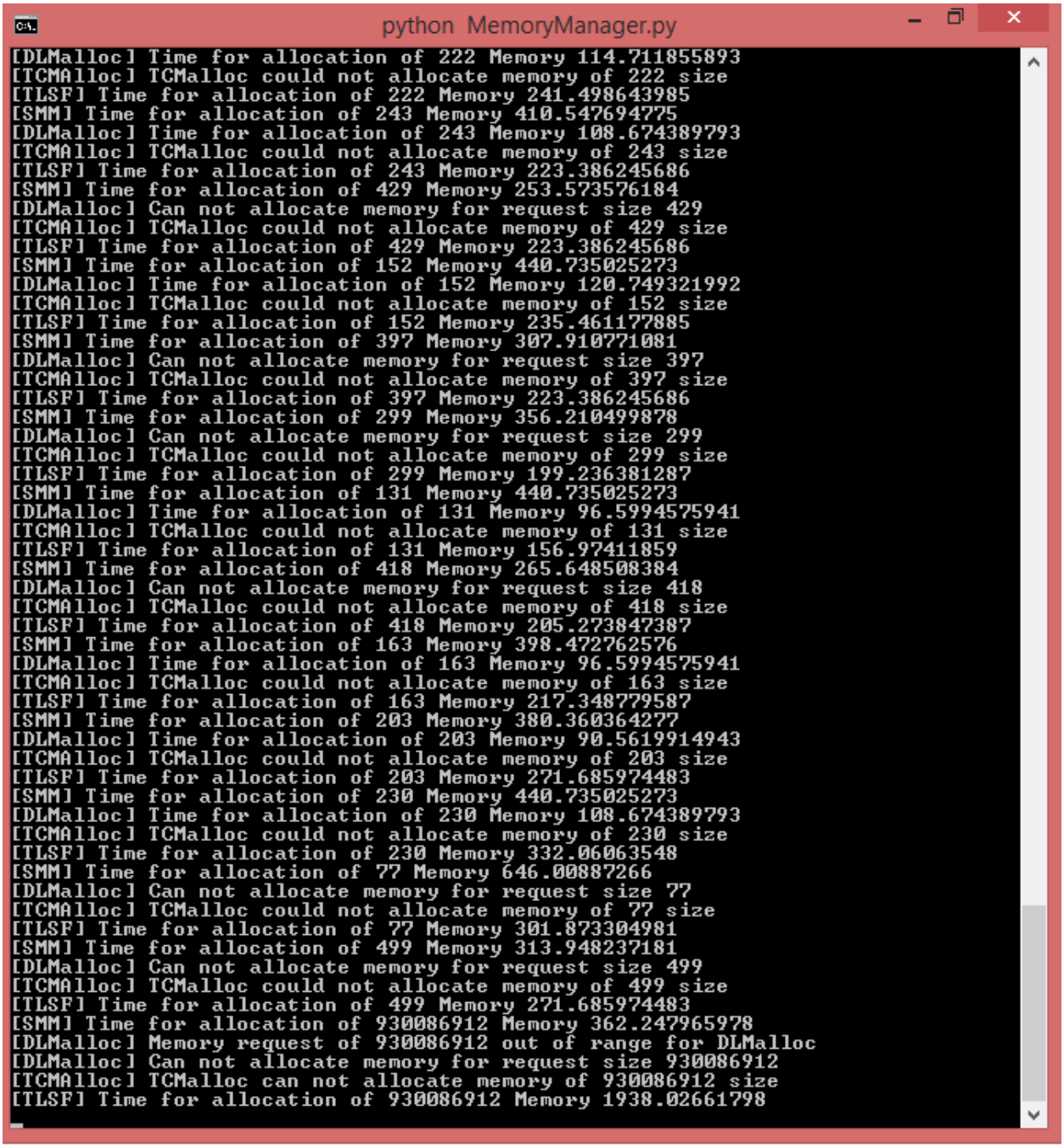

Figure 5.8: Memory block allocation Log

As shown in Figure 5.8, its memory allocation log. It is the status of requested memory block of the specific allocator. Also, it generates the CSV file for the same.

Chapter 6
Conclusion & Future Scope

6.1 Conclusion

1. Dynamic memory allocator has been proposed for symmetric multiprocessing system which provides consistent and optimum execution time, less memory fragmentation as well as satisfies maximum number of memory request as compared to other existing dynamic memory allocators.
2. As per the need of high-performance computing, a dynamic memory allocator for NUMA architecture based real-time operating system has been proposed which also provides consistent and optimum execution time, less memory fragmentation as well as satisfies maximum number of memory request.
3. MemSimRT has been designed to simulate various memory allocators for both SMP as well as NUMA architecture based RTOS.

6.2 Future Work

The analysis presented here points to several areas for future work.

1. The implementation presented here is a user-level memory allocation allocator; so one can reform the kernel of Linux to minimize the distance when remote access is required.
2. Synchronization for sharing the resource is a significant thought in the proposal of a memory allocation algorithm which supports concurrency and scalability. But, for real-time systems, it is essential to consider schedulability with the synchronization. So one can design a scheduler which can integrate with DmRT.
3. The garbage collector is still a challenge in the real-time system because of its random delays. Still, there are real-time garbage collectors available which provide satisfactory performance on soft real-time systems. The incorporation of these garbage collector with DmRT can be another area of future work.

References

[1]. Andrew Borg. (2006). Coarse grain memory management in real-time systems. University of York, UK

[2]. Apache Software Foundation (2013). Memory management with pools. "http://www.fmc-modeling.org/category/projects/apache/amp/3 3Extending Apache.html".

[3]. Ban´us, J. M., Arenas, A., and Labarta, J. (2002). An efficient scheme to allocate soft-aperiodic tasks in multiprocessor hard real-time systems. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 2, PDPTA '02, (pp. 809–815). CSREA Press.

[4]. Barrett, D. A. and Zorn, B. G. (1993). Using lifetime predictors to improve memory allocation performance. SIGPLAN Not., 28(6): (pp. 187–196).

[5]. Bays, C. (1977). A comparison of next-fit, first-fit, and best-fit. Commun. ACM, 20(3): (pp. 191–192).

[6]. Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. (2000). Hoard: a scalable memory allocator for multithreaded applications. SIGPLAN Not., 35(11): (pp. 117–128).

[7]. Bohra, A. and Gabber, E. (2001). Are mallocs free of fragmentation? In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, (pp. 105–117), Berkeley, CA, USA. USENIX Association.

[8]. Bolosky, W. J. and Scott, M. L. (1993). False sharing and its effect on shared memory performance. In In Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), (pp. 57–71).

[9]. Brent, R. P. (1989). Efficient implementation of the first-fit strategy for dynamic storage allocation. ACM Trans. Program. Lang. Syst., 11(3): (pp. 388–403).

[10]. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc: A generic framework for managing hardware affinities in hpc applications. In Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on, (pp. 180 –186).

[11]. Burns, A. and Wellings, A. J. (2001). Real-Time Systems and Programming Languages. Addison Wesley, 3rd edition.

[12]. Changwoo Min, Inhyeok Kim, Taehyoung Kim, Young Ik Eom. (2012). VMMB: Virtual Machine Memory Balancing for Unmodified Operating Systems. Published in Journal of Grid Computing. 10(1): (pp. 69-84)

[13]. Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., and Iyer, R. (2005). Defeating memory corruption attacks via pointer taintedness detection. In Proceedings of the 2005 International Conference on Dependable Systems and Networks, DSN '05, (pp. 378 – 387), Washington, DC, USA. IEEE Computer Society.

[14]. Chowdhury, S. K. and Srimani, P. K. (1987). Worst case performance of weighted buddy systems. Acta Informatica, 24:555–564. 10.1007/BF00263294.

[15]. Christian Del Rosso. (2005). Dynamic Memory Management for Software Product Family Architectures in Embedded Real-Time Systems. Fifth Working {IEEE} / {IFIP} Conference on Software Architecture (pp. 211-212)

[16]. Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, Emmett Witchel. (2007). TxLinux: using and managing hardware transactional memory in an operating system. Published in Proceedings of the 21st {ACM} Symposium on Operating Systems Principles SOSP (pp. 87-102)

[17]. Confessore, G., Dell'Olmo, P., and Giordani, S. (2001). An approximation result for a periodic allocation problem. Discrete Applied Mathematics, 112(1–3): (pp. 53–72).

[18]. Cowan, C., Wagle, F., Pu, C., Beattie, S., and Walpole, J. (2000). Buffer overflows: attacks and defenses for the vulnerability of the decade. In DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings, volume 2, (pp. 119–129).

[19]. Crespo, A., Ripoll, I., and Masmano, M. (2006). Dynamic memory management for embedded real-time systems. 225: (pp. 195–204).

[20]. Daniel P. Bovet, M. C. (2005). Understanding the Linux Kernel, chapter Chapter 2. O'Reilly, 3rd edition.

[21]. Dipti Diwase, Shraddha Shah, Tushar Diwase and Priya Rathod. (2012). Survey Report on Memory Allocation Strategies for Real-time Operating System in Context with Embedded Devices. International Journal of Engineering Research and Applications, Vol. 2, Issue 3, (pp.1151-1156).

[22]. Dirk Vogt, Cristiano Giuffrida, Herbert Bos, Andrew S. Tanenbaum. (2014) Techniques for efficient in-memory checkpointing. Published in Operating Systems Review 48(1) (pp. 21-25)

[23]. las, N. (2011). nedmalloc. http://www.nedprod.com/programs/portable/nedmalloc.

[24]. Edge, J. (2009). Perfcounters added to the mainline. http://lwn.net/Articles/336542/.

[25]. Felicia Ionescu. (2000). Application-Level Virtual Memory Management in Real-Time Multiprocessor Systems. Proceedings of the {ACM} Symposium on Applied Computing, Villa Olmo, Via Cantoni 1, 22100 Como, Italy. (pp. 610-614)

[26]. Ferreira, T., Matias, R., Macedo, A., and Araujo, L. (2011). An experimental study on memory allocators in multicore and multithreaded applications. In Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on, (pp. 92 –98).

[27]. Fredkin, E. (1960). Trie memory. Commun. ACM, 3(9): (pp. 490–499).

[28]. FSF, F. s. f. (2012a). Glibc, the gnu c library. http://www.gnu.org/software/libc/libc.html.

[29]. FSF, F. s. f. (2012b). The gnu c++ library manual. http://gcc.gnu.org/onlinedocs/libstdc++.

[30]. Garey, M. R. and Johnson, D. S. (1979). A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., San Francisco, CA.

[31]. Gergov, J. (1996). Approximation algorithms for dynamic storage allocation. In Algorithms — ESA '96, volume 1136, pages 52–61. Springer Berlin / Heidelberg.

[32]. Gergov, J. (1999). Algorithms for compile-time memory optimization. In Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms, SODA '99, (pp. 907–908), Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[33]. Gloger, W. (2001). Dynamic memory allocator implementations in linux system libraries.

[34]. Gloger, W. (2006). ptmalloc2. "http://www.malloc.de/en/".

[35]. Grunwald, D., Zorn, B., and Henderson, R. (1993). Improving the cache locality of memory allocation. SIGPLAN Not., 28(6): (pp. 177–186).

[36]. Hans-Georg Eßer. (2011) Combining memory management and filesystems in an operating systems course. Proceedings of the 16th Annual {SIGCSE} Conference on Innovation and Technology in Computer Science Education, Darmstadt, Germany.

[37]. Hasan, Y. and Chang, M. (2005). A study of best-fit memory allocators. Computer Languages, Systems & Structures, 31(1): (pp. 35 – 48).

[38]. Hasan, Y., Chen, W.-M., Chang, J. M., and Gharaibeh, B. M. (2010). Upper bounds for dynamic memory allocation. IEEE Trans. Comput., 59(4): (pp. 468–477).

[39]. Hirschberg, D. S. (1973). A class of dynamic memory allocation algorithms. Commun. ACM, 16(10): (pp. 615–618).

[40]. Hyde, R. L. and Fleisch, B. D. (1996). An analysis of degenerate sharing and false coherence. J. Parallel Distrib. Comput., 34(2): (pp. 183–195).

[41]. Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, Lawrence Chiu. (2014). Phase change memory in enterprise storage systems: silver bullet or snake oil? Published in Operating Systems Review 48(1): (pp. 82-89)

[42]. Ian McDonald. (1999). Distributed, Configurable Memory Management in an Operating System Supporting Quality of Service. FTDCS 1999: (pp. 191-196).

[43]. Ian McDonald. (2001). Memory management in a distributed system of single address space operating systems supporting quality of service. University of Glasgow, UK

[44]. Jane W. S. Liu. (2000). "Real-time System", 1st Edition published by Person Education.

[45]. Jeremiassen, T. E. and Eggers, S. J. (1995). Reducing false sharing on shared memory multiprocessors through compile time data transformations. SIGPLAN Not., 30(8): (pp. 179–188).

[46]. Johnstone, M. S. and Wilson, P. R. (1998). The memory fragmentation problem: solved? SIGPLAN Not., 34(3): (pp. 26–36).

[47]. Kaminski, P. (2009). Numa aware heap memory manager. "http://amddevcentral.com/Assets/NUMA aware heap memory manager article final.pdf".

[48]. Kingsley, C. (1982). Description of a very fast storage allocator.

[49]. Kiyohito Miyazaki, Yoshinari Nomura, Hideo Taniguchi. (2013). Memory Segmentation and Transfer in Mint Operating System. Published in 16th International Conference on Network-Based Information Systems NBiS 2013 (pp. 366-371)

[50]. Kleen, A. (2005). A NUMA API for Linux. Novel Inc. accessed on September 2011.

[51]. Knuth, D. (1997). The art of computer programming: Fundamental Algorithms, volume 1. addison-Wesley, 2 edition.

[52]. Larson, P.-k. and Krishnan, M. (1998). Memory allocation for long running server applications. SIGPLAN Not., 34(3): (pp. 176–185).

[53]. Lea, D. (1996). A memory allocator. ”http://g.oswego.edu/dl/html/malloc.html”. Unix/Mail December, 1996.

[54]. Lei Liu, Mengyao Xie, Hao Yang. (2017). Memos: Revisiting Hybrid Memory Management in Modern Operating System. CoRR abs/1703.07725

[55]. Lei Liu, Yong Li, Chen Ding, Hao Yang, Chengyong Wu. (2016). Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random? IEEE Trans. Computers 65(6): (pp. 1921-1935)

[56]. Linus Torvalds, e. (2011). Source codes of linux kernel v3.0.4. ”http://lxr. linux.no/linux+v3.0.4/”.

[57]. Liu, T. and Berger, E. D. (2011). Sheriff: precise detection and automatic mitigation of false sharing. SIGPLAN Not., 46(10): (pp. 3–18).

[58]. Luby, M. G., Naor, J. S., and Orda, A. (1994). Tight bounds for dynamic storage allocation. In Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms, SODA ’94, (pp. 724–732), Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[59]. M. Masmano, I. Ripoll and A. Crespo. (2004). TLSF: A new dynamic memory allocator for real-time systems. Proceedings of 16th Euromicro Conference on Real-Time Systems, Catania, Italy, July 2004, (pp. 79-88).

[60]. Majo, Z. and Gross, T. R. (2011). Memory system performance in a numa multicore multiprocessor. In Proceedings of the 4th Annual International Conference on Systems and Storage, SYSTOR ’11, (pp. 1–10), New York, NY, USA. ACM.

[61]. Manoj Kumar Putchala, Adam R. Bryant. (2016). Synchron-ITS: An Interactive Tutoring System to Teach Process Synchronization and Shared Memory Concepts in an Operating Systems Course. International Conference on Collaboration Technologies and Systems CTS 2016: (pp. 180-187)

[62]. Marathe, J. and Mueller, F. (2006). Hardware profile-guided automatic page placement for ccnuma systems. In Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP ’06, (pp. 90–99), New York, NY, USA. ACM.

[63]. Marchand, A., Balbastre, P., Ripoll, I., Masmano, M., and Crespo, A. (2007). Memory resource management for real-time systems. In Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference, (pp. 201 –210).

[64]. María-del-Mar Gallardo, Pedro Merino, David Sanán. (2009). Model Checking Dynamic Memory Allocation in Operating Systems. Published in Journal of Automation Reasoning 42(2-4): (pp. 229-264)

[65]. Mark Oskin, Diana Keen, Justin Hensley, Lucian Vlad Lita, Frederic T. Chong. (2002). Operating Systems Techniques for Parallel Computation in Intelligent Memory. Parallel Processing Letters 12(3-4): (pp. 311-326)

[66]. Masmano (2012). The lastest version of TLSF source. http://wks.gii.upv.es/tlsf/files/src/TLSF-2.4.6.tbz2.

[67]. Masmano, M., Ripoll, I., and Crespo, A. (2003). Dynamic storage allocation for real-time embedded systems. Proc. of Real-Time System Simposium WIP.

[68]. Masmano, M., Ripoll, I., Balbastre, P., and Crespo, A. (2008a). A constant-time dynamic storage allocator for real-time systems. Real-Time Systems, 40(2): (pp. 149–179).

[69]. Masmano, M., Ripoll, I., Real, J., Crespo, A., and Wellings, A. (2008b). Implementation of a constant-time dynamic storage allocator. Software: Practice and Experience, 38(10): (pp. 995–1026).

[70]. McCamant, S. and Ernst, M. D. (2007). A simulation-based proof technique for dynamic information flow. In Proceedings of the 2007 workshop on Programming languages and analysis for security, PLAS '07, (pp. 41–46), New York, NY, USA. ACM.

[71]. McCurdy, C. and Vetter, J. (2010). Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on, (pp. 87–96).

[72]. Meenakshi Sundaram Bhaskaran, Jian Xu, Steven Swanson. (2014). Bankshot: caching slow storage in fast non-volatile memory. Operating Systems Review 48(1): (pp. 73-81)

[73]. Mehta, H., Owens, R., Irwin, M., Chen, R., and Ghosh, D. (1997). Techniques for low energy software. In Low Power Electronics and Design, 1997. International Symposium, (pp. 72–75).

[74]. MicroQuill (2012). shbench benchmark tool. http://www.microquill.com.

[75]. Mochel, P. (2005). The sysfs filesystem. In Linux Symposium, (pp. 313–326), Ottawa, Ontario, Canada.

[76]. Molnar, I. (2009). Performance counters for linux, v8. http://lwn.net/Articles/336542/.

[77]. MS Johnstone. (1998). Fragmentation Problem: Solved? Proceeding of the Int. Symposium on Memory Management (ISMM'98), Vancouver, Canada. ACM Press

[78]. Nethercote, N. and Mycroft, A. (2002). The cache behaviour of large lazy functional programs on stock hardware. SIGPLAN Not., 38(2 supplement): (pp. 44–55).

[79]. Nilsen, K. and Gao, H. (1995). The real-time behavior of dynamic memory management in c++. In Real-Time Technology and Applications Symposium, 1995. Proceedings, (pp. 142–153).

[80]. Ogasawara, T. (1995). An algorithm with constant execution time for dynamic storage allocation. In RTCSA '95: Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications, pages 21–25, Washington, DC, USA. IEEE Computer Society.

[81]. Ogasawara, T. (2009). Numa-aware memory manager with dominant-thread based copying gc. SIGPLAN Not., 44(10): (pp. 377–390).

[82]. Oracle Inc. (2013). Memory architecture. "http://docs.oracle.com/cd/E14072 01/server.112/e10713/memory.htm".

[83]. Page, I. and Hagins, J. (1986). Improving the performance of buddy systems. Computers, IEEE Transactions on, C-35(5): (pp. 441 –447).

[84]. Pawel Pisarczyk. (2007). Experimental Dependability Evaluation of Memory Manager in the Real-time Operating System. IJCSA 4(1): (pp. 33-38)

[85]. Paul Werstein, Hailing Situ, Zhiyi Huang. (2006). "Load Balancing in a Cluster Computer", Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'06).

[86]. Peng Wei, Lihua Yue, Zhanzhan Liu, Xiaoyan Xiang. (2008). Flash memory management based on predicted data expiry-time in embedded real-time systems. Proceedings of the 2008 {ACM} Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008 (pp. 1477-1481)

[87]. Polishchuk, M., Liblit, B., and Schulze, C. W. (2007). Dynamic heap type inference for program understanding and debugging. SIGPLAN Not., 42(1): (pp. 39–46).

[88]. Puaut, I. (2002). Real-Time Performance of Dynamic Memory Allocation Algorithms. In ECRTS ’02: Proceedings of the 14th Euromicro Conference on Real-Time Systems, (pp. 41–49), Washington, DC, USA. IEEE Computer Society.

[89]. Puaut, I. and Hardy, D. (2007). Predictable paging in real-time systems: A compiler approach. In Real-Time Systems, 2007. ECRTS ’07. 19th Euromicro Conference on, (pp. 169 –178).

[90]. Randell, B. (1969). A note on storage fragmentation and program segmentation. Commun. ACM, 12(7).

[91]. Reza Salkhordeh, Hossein Asadi. (2016) An Operating System level data migration scheme in hybrid DRAM-NVM memory architecture. Automation & Test in Europe Conference & Exhibition. (pp. 936-941)

[92]. Robart L. Budzinski, Edward S. Davidson. (1981). A Comparison of Dynamic and Static Virtual Memory Allocation Algorithms” IEEE Transactions on software Engineering, Vol. SE-7, NO. 1, January 1981.

[93]. Robertson, W., Kruegel, C., Mutz, D., and Valeur, F. (2003). Run-time detection of heap-based overflows. In Proceedings of the 17th USENIX conference on System administration, LISA ’03, (pp. 51–60), Berkeley, CA, USA. USENIX Association.

[94]. Robson, J. (1980). Storage allocation is np-hard. Information Processing Letters, 11(3): (pp. 119–125).

[95]. Robson, J. M. (1971). An estimate of the store size necessary for dynamic storage allocation. J. ACM, 18(3): (pp. 416–423).

[96]. Robson, J. M. (1977). Worst case fragmentation of first fit and best fit storage allocation strategies. The Computer Journal, 20(3): (pp. 242–244).

[97]. S. Baskiyar, Ph.D. and N. Meghanathan. (2005). A Survey of Contemporary Real-time Operating Systems Informatica 29, (pp. 233–240)

[98]. Sangsoo Park, Yonghee Lee, Heonshik Shin. (2004). An experimental analysis of the effect of the operating system on memory performance in embedded multimedia computing. EMSOFT 2004. (pp. 26-33)

[99]. Sanjay Ghemawat, P. M. (2010). Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[100]. Seyeon Kim. (2013). Node-oriented dynamic memory management for real-time systems on ccNUMA architecture systems. University of York, UK.

[101]. SGI (2004). Standard template library programmer's guide: Allocators. "http://www.sgi.com/tech/stl/Allocators.html".

[102]. Shen, K. K. and Peterson, J. L. (1974). A weighted buddy method for dynamic storage allocation. Commun. ACM, 17(10) (pp. 558–562).

[103]. Shore, J. E. (1975). On the external storage fragmentation produced by first-fit and best-fit allocation strategies. Commun. ACM, 18(8): (pp. 433–440).

[104]. Standish, T. A. (1980). Data Structure Techniques. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[105]. Stephenson, C. J. (1983). New methods for dynamic storage allocation (fast fits). SIGOPS Oper. Syst. Rev., 17(5): (pp. 30–32).

[106]. Sybase Inc. (2013). Configuration parameters that affect memory allocation.

[107]. Tao, J., Kunze, M., and Karl, W. (2008). Evaluating the cache architecture of multicore processors. In Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference, (pp. 12 –19).

[108]. Tiancheng Liu, Ying Li, Andrew Schofield, Matt Hogstrom, Kewei Sun, Ying Chen. (2008). Partition-based heap memory management in an application server. Operating Systems Review 42(1): (pp. 98-103)

[109]. Tikir, M. and Hollingsworth, J. (2008). Hardware monitors for dynamic page migration. Journal of Parallel and Distributed Computing, 68(9): (pp. 1186–1200).

[110]. Timothy Wood, Gabriel Tarasuk-Levin, Prashant J. Shenoy, Peter Desnoyers, Emmanuel Cecchet, Mark D. Corner. (2009). Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. Published in Operating Systems Review 43(3): (pp. 27-36)

[111]. Vatsal Shah, Kanu Patel. (2012). Load Balancing algorithm by Process Migration in Distributed Operating System. International Journal of Computer Science and Information Technology & Security (IJCSITS), ISSN: 2249-9555, Vol. 2, No.6.

[112]. V Shah, A Shah. (2017). Critical Analysis for Memory Management Algorithm for NUMA based Real-time Operating System. IEEE Xplore.

[113]. V Shah, A Shah. (2018). Proposed Memory Allocation Algorithm for NUMA based Soft Real-time Operating System. International Conference On Emerging Technologies In Data Mining And Information Security (IEMIS 2018)

[114]. Vatsal Shah, Apurva Shah. (2016). An Analysis and Review on Memory Management Algorithms for Real-time Operating System. International Journal of Computer Science and Information Security (IJCSIS), Vol. 14, No. 5.

[115]. Vee, V.-Y. and Hsu, W.-J. (1999). A scalable and efficient storage allocator on shared memory multiprocessors. In Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks, ISPAN '99, Washington, DC, USA. IEEE Computer Society.

[116]. Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. (1995b). Dynamic Storage Allocation: A Survey and Critical Review. In IWMM '95: Proceedings of the International Workshop on Memory Management, (pp. 1–116), London, UK. Springer-Verlag.

[117]. Wilson, P., Johnstone, M., Neely, M., and Boles, D. (1995a). Memory allocation policies reconsidered. Technical report, Technical report, University of Texas at Austin Department of Computer Sciences.

[118]. XiaoHui Sun, JinLin Wang, xiao chan. (2007). "An Improvement of TLSF Algorithm".

[119]. Youngki Chung, Ramakrishna M, Jisung Kim and Woohyong Lee. (2008). Smart Dynamic Memory Allocator for embedded systems. Proceedings of 23rd International Symposium on Computer and Information Sciences, ISCIS '08.

[120]. Zhou, X. and Petrov, P. (2011). Towards virtual memory support in real-time and memory-constrained embedded applications: the interval page table. Computers Digital Techniques, IET, 5(4): (pp. 287 –295).

Publications

1. Vatsalkumar H. Shah, Dr. Apurva Shah, (May, 2016). ***"An Analysis and Review on Memory Management Algorithms for Real-time Operating System"*** *published in International Journal of Computer Science and Information Security, Volume 14, Issue 5, (pp. 236-240)* (Web of Science Thomson Reuters, Scopus, DOAJ)

2. Vatsalkumar H. Shah, Dr. Apurva Shah, (December, 2017). ***"Critical Analysis for Memory Management Algorithm for NUMA based Real-time Operating System"*** *published in Proceedings of IEEE Conference, 2017***.** (pp. 323-327). (INSPEC, Scopus Indexed)

3. Vatsalkumar H. Shah, Dr. Apurva Shah, (February, 2018). ***"Proposed Memory Allocation Algorithm for NUMA based Soft Real-time Operating System"*** *published in Advances in Intelligent Systems and Computing (AISC), Springer Series*. (ISI, DBLP, EI-Compendex, SCOPUS)

4. Vatsalkumar H. Shah, Dr. Apurva Shah, (June, 2018). ***"Memory Allocator for SMP & NUMA based Soft Real-time Operating System"*** *published in Advances in Intelligent Systems and Computing (AISC), Springer Series*. (ISI, DBLP, EI-Compendex, SCOPUS)

Annexure 1

Test Case: Existing and DmRT allocators allocate from Available Local Memory (SMP)

1. Execution time for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	288.34	327.13	275.49	231.27		**51**	289.77	328.06	270.68	233.84
2	289.16	336.33	265.7	238.33		**52**	285.85	335.79	261.53	233.5
3	292.92	328.88	264.89	239.29		**53**	291.96	326.5	270.75	234.93
4	285.29	334.04	261.84	236.12		**54**	291.94	327.84	265.29	239.12
5	291.48	320.52	261.24	239.85		**55**	287.37	327.7	272.98	230.96
6	285.73	335.15	267.12	236.18		**56**	282.93	333.73	267.21	235.21
7	287.63	338.47	272.78	231.99		**57**	288.97	325.8	268.35	233.59
8	285.57	320.71	271.64	234.12		**58**	282.58	320.62	263.46	234.85
9	286.49	333.16	264.94	239.77		**59**	283.79	337.87	273.47	238.13
10	286.43	329.69	266.34	230.24		**60**	286.38	331.07	269.38	239.96
11	291.08	337.47	273.02	230.46		**61**	293.61	321.46	267.83	236.49
12	291.01	324.02	267.05	233.09		**62**	287.2	326.52	267.48	233.67
13	281.44	333.49	274.97	232.36		**63**	285.5	323.78	266	230.95
14	295.63	330.52	273.73	237.13		**64**	286.23	336.86	269.75	232.1
15	287.06	326.78	268.04	239.8		**65**	285.02	320.96	267.59	231.99
16	290.61	337.58	274.18	234.11		**66**	292.54	321.07	263.14	231.39
17	281.8	329.6	268.5	237.29		**67**	292.64	333.84	274.95	230.03
18	282.87	331.05	270.25	237.53		**68**	284.27	338.76	270.38	230.69
19	291.95	327.18	272.76	239.59		**69**	284.69	328.31	270.8	232.81
20	281.59	335.69	261.9	232.91		**70**	281.48	334.41	275.86	234.16
21	290.12	330.43	264.25	230.07		**71**	294.31	338.43	264.16	234.49
22	282.69	338.22	271.37	237.42		**72**	295.56	330.57	272.03	237.3
23	289.04	324.71	269.34	236.77		**73**	288.71	333.98	273.09	232.77
24	292.46	326.83	272.38	238.59		**74**	288.99	330.2	274.09	232.82
25	293.08	322.17	264.39	233.3		**75**	283.37	337.07	267.4	232.04
26	293.08	339.94	274.92	234.35		**76**	286.13	339.43	270.46	234.69
27	291.16	327.97	267.82	233.7		**77**	289.02	331.14	266.73	233.45
28	281.04	321.36	273.77	231.89		**78**	282.06	327.4	263.22	233.08
29	291.63	327.77	267.6	232.21		**79**	287.85	328.09	274	236.8
30	281.13	325.23	265.26	230.51		**80**	282.75	322.18	272.95	230.1
31	283.89	339.45	262.56	239.03		**81**	294.2	333.64	261.57	232.91
32	287.46	329.42	269.51	232.19		**82**	295.5	335.93	263.92	232.33
33	286.55	336.05	264.04	234.11		**83**	287.34	338.39	265.11	231.55
34	291.21	335.4	263.75	236.46		**84**	288.27	323.44	272.95	234.15
35	287.82	339.22	271.3	236.34		**85**	285.54	337.76	274.82	238.8
36	281.34	324.31	270.33	233.65		**86**	295.64	327.96	275.84	234.57
37	288.73	325.84	272.03	239.23		**87**	295.79	337.34	267.75	231.32
38	282.3	322.63	268.98	237.75		**88**	283.2	337.3	265.52	235.06
39	284.46	329.03	270.31	235.98		**89**	285.1	323.73	271.11	230.33
40	294.86	336.09	272.66	235.88		**90**	290.62	331.34	263.94	236.78
41	289.09	336.21	261.76	231.3		**91**	289.47	332.27	261.69	231.68
42	282.52	327	275.62	231.46		**92**	282.41	326.5	268.4	236.64
43	286.26	324.56	266.74	236.26		**93**	294.65	337.95	262.97	232.21
44	286.96	326.64	271.73	236.15		**94**	287.78	333.28	270.06	233.64
45	282.08	325.29	266.24	239.03		**95**	286.32	336.87	264.48	235.75
46	286.43	337.94	268.35	230.66		**96**	288.23	323.57	267.28	233.82
47	285.4	330.81	267.84	236.58		**97**	284.98	328.17	268.52	235.31
48	290.98	322.01	264.14	239.51		**98**	292.21	332.23	273.12	232.36
49	287.91	325.74	275.92	237.77		**99**	293.84	323.48	269.54	235.29
50	284.57	333.34	262.49	231.44		**100**	290.92	320.37	262.42	239.85
Avg of 100 Attempts	**287.8581**	**330.3003**	**268.598**	**234.6128**						

Test Case: Existing and DmRT allocators allocate from Available Local Memory (SMP)

2. Fragmentation for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	41.77	30.84	24.68	15.36		**51**	44.28	27.69	21.51	16.51
2	44.22	30.98	24.67	17.26		**52**	45.37	28.91	22.57	18.92
3	45.53	29.38	22.96	16.4		**53**	45.37	28.76	23.24	16.84
4	45.81	28.85	24.26	18.18		**54**	44.31	31.41	20.26	18.8
5	44.69	30.43	22.58	15.25		**55**	41.11	30.06	23.52	16.51
6	43.99	31.81	21.1	17.4		**56**	41.45	27.49	20.77	16.95
7	41.13	27.42	24.48	15.74		**57**	42.38	30.15	24.56	19.1
8	42.35	28.91	23.56	19.7		**58**	45.95	30.78	20.63	19.69
9	43.26	31.68	23.15	19.21		**59**	42.57	30.85	21.1	19.35
10	44.29	28.4	24.11	18.08		**60**	42.81	29.35	23.71	19.65
11	43.68	29.12	24.59	15.03		**61**	44.42	30.75	21.04	16.95
12	41.21	29.72	21.86	18.84		**62**	45.57	30.6	23.32	18.88
13	42.37	31.04	20.47	16.14		**63**	45.87	30.19	22.68	16.95
14	44.76	27.74	20.87	17.14		**64**	41.11	28	21.94	18.45
15	43	30.53	20.53	18.62		**65**	45.93	28.58	24.12	17.89
16	44.79	28.03	23.13	19.87		**66**	45.71	31.31	20.9	15.04
17	45.3	29.78	24.56	19.17		**67**	44.65	29.73	22.3	15.64
18	41.65	28.46	21.61	15.11		**68**	41.32	30.65	22.47	15.83
19	41.55	27.92	22.31	16.96		**69**	43.55	31.93	21.7	19.35
20	44.3	28.91	23.34	16.92		**70**	41.66	28.71	22.92	15.78
21	44.37	30.26	22.54	18.18		**71**	45.15	27.41	20.67	19.24
22	41.19	27.7	20.66	15.24		**72**	43.06	31.38	23.81	18.42
23	43.4	30.59	24.89	17.19		**73**	44.88	31.54	24.22	16.11
24	41.73	30.97	21.64	16.56		**74**	44.44	27.41	21.43	18.6
25	42.99	28.55	21.9	18.16		**75**	43.67	30.13	21.62	15.34
26	45.67	28.12	22.33	19.85		**76**	45.29	30.94	21.6	17.21
27	44.07	31.14	22.29	17.44		**77**	41.58	27.13	24.44	15.71
28	45.84	27.82	21.1	16.02		**78**	41.93	30.34	21.52	19.81
29	41.98	31.34	20.72	15.43		**79**	41.32	27.19	20.37	17.42
30	42.19	31.85	23.19	15.06		**80**	45.58	31.05	20.06	17.89
31	42.19	30.92	20.12	15.46		**81**	45.05	31.59	24.01	17.87
32	42.24	27.08	24.19	17.84		**82**	45.38	27.6	23.77	18.27
33	44.19	30.81	23.95	19.22		**83**	45.19	27.01	22.48	18.94
34	42.39	31.85	20.26	16.38		**84**	44.96	30.56	24.31	18.99
35	43.26	29.85	20.79	18.03		**85**	42.5	27.36	21.31	17.19
36	43.96	31.81	21.63	18.34		**86**	44.84	28.98	22.83	16.1
37	45.53	29.38	24.42	16.54		**87**	43.64	31.21	21.89	19.22
38	43.42	29.78	22.81	16.61		**88**	44.31	29.2	22.09	16.94
39	43.33	29.73	23.65	17.84		**89**	41.56	31.7	23.68	18.14
40	45.24	27.53	22.28	16.33		**90**	44.44	30.48	20.91	18.34
41	42.16	30.64	23.04	15.23		**91**	44.59	28.15	21.99	19.52
42	45.6	27.09	20.86	19.13		**92**	42.96	29.2	21.83	17.58
43	43.07	27.59	20.06	19.56		**93**	45.5	31.29	24.87	18.74
44	41.02	30.2	24.09	15.32		**94**	44.13	28.73	21.04	18.01
45	45.22	31.13	23.07	17.55		**95**	44.38	30.1	22.27	15.19
46	44.88	31.97	23.82	19.47		**96**	43.96	31.56	22.88	15.01
47	42.85	29.5	21.15	17.64		**97**	43.42	29.59	23.03	19.17
48	42.69	31.81	21.84	16.91		**98**	43.32	29.7	24.96	17.92
49	43.71	28.52	21.92	18.48		**99**	42.35	28.47	23.21	19.71
50	43.07	31.69	22.65	16.38		**100**	42.85	28.33	22.87	16.86
Avg of 100 Attempts	**43.65**	**29.68**	**22.48**	**17.50**						

Test Case: Existing and DmRT allocators allocate from Available Local Memory (SMP)

3. No. of Request Satisfied for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	57	62	83	88		**51**	56	62	81	89
2	59	63	84	88		**52**	56	63	84	88
3	56	61	84	89		**53**	59	61	83	88
4	56	61	81	86		**54**	59	63	84	86
5	55	65	83	87		**55**	57	61	84	88
6	59	62	83	87		**56**	57	64	79	89
7	55	60	83	87		**57**	57	60	84	86
8	57	62	83	86		**58**	56	61	81	85
9	57	65	83	86		**59**	55	64	82	86
10	54	62	80	88		**60**	59	65	84	86
11	57	62	81	89		**61**	56	64	81	87
12	54	61	83	90		**62**	56	65	81	90
13	56	64	79	89		**63**	58	64	81	88
14	55	65	82	90		**64**	54	61	83	86
15	55	63	83	86		**65**	54	60	79	90
16	58	62	80	88		**66**	55	61	81	89
17	54	63	83	88		**67**	57	65	81	88
18	57	60	82	89		**68**	55	61	81	87
19	56	61	82	86		**69**	58	64	81	86
20	58	64	82	86		**70**	55	64	80	87
21	58	64	80	86		**71**	57	64	81	89
22	58	64	82	87		**72**	56	64	79	87
23	59	62	82	89		**73**	57	61	82	89
24	57	63	82	87		**74**	56	62	82	86
25	55	61	80	87		**75**	56	63	79	90
26	54	65	80	87		**76**	56	65	80	85
27	57	65	82	90		**77**	57	64	84	86
28	54	62	80	89		**78**	55	62	81	88
29	58	64	84	87		**79**	57	65	81	88
30	56	61	80	90		**80**	58	65	80	87
31	56	60	80	87		**81**	58	61	80	88
32	59	61	79	85		**82**	58	60	83	90
33	55	62	82	85		**83**	54	65	82	85
34	58	64	83	89		**84**	58	61	81	90
35	58	60	80	88		**85**	55	63	82	87
36	55	65	82	89		**86**	55	64	82	86
37	57	60	83	90		**87**	57	63	82	85
38	57	60	83	87		**88**	57	64	81	85
39	55	62	82	86		**89**	59	61	82	86
40	55	65	83	89		**90**	56	65	84	89
41	58	65	83	89		**91**	58	62	79	85
42	57	64	80	87		**92**	56	61	81	90
43	55	61	81	88		**93**	57	61	80	90
44	58	64	83	87		**94**	58	61	82	87
45	55	62	79	87		**95**	58	62	79	88
46	58	62	81	90		**96**	59	63	82	86
47	56	61	82	90		**97**	56	65	79	85
48	58	65	84	89		**98**	55	61	79	87
49	56	65	81	90		**99**	58	63	84	88
50	59	65	81	89		**100**	56	60	82	89
Avg of 100 Attempts	**56.62**	**62.59**	**81.57**	**87.62**						

Test Case: Existing and DmRT allocators allocate from Available Local Memory (SMP)

1. Execution Time for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	1995.11	2896.17	1381.19	1069.76		**51**	1811.2	2727.38	1303.27	1054.52
2	1953.73	3039.92	1528.14	1053.01		**52**	1946.9	2760.15	1392.28	1017.18
3	1888.73	3080.42	1442.76	1019.13		**53**	1816.12	2796.57	1405.25	1031.91
4	1958.15	2824.03	1355	1044.85		**54**	1934.69	3070.81	1529.98	1087.95
5	1956.2	2906.1	1480.57	1026.51		**55**	1881.33	2966.48	1315	1115.07
6	1959.53	2941.39	1596.21	1000.51		**56**	1930.68	3022.12	1363.43	1075.95
7	1988.52	2712.13	1595.66	1034.22		**57**	1870.43	2962.25	1585.3	1139.64
8	1829.79	3089.7	1551.09	1135.16		**58**	1845.94	2702.25	1395.17	1113.11
9	1805.38	3021.91	1461.34	1049.19		**59**	1840.65	3074.06	1466.05	1140.82
10	1851.57	2860.88	1569.23	1065.34		**60**	1802.44	2968.37	1598.24	1088.32
11	1930.61	2805.39	1599.12	1071.7		**61**	1932.84	2931.62	1466.73	1053.75
12	1876.48	3072.65	1420.21	1001.34		**62**	1911.69	3062.3	1300.16	1052.28
13	1974.12	2957.93	1411.22	1086.63		**63**	1914.45	3013.03	1563.26	1148.46
14	1942.79	2985.92	1529.92	1114.61		**64**	1857.34	2965.84	1524.52	1077.6
15	1817.58	3013.36	1542.16	1075.65		**65**	1927.82	2727.97	1383.42	1071.21
16	1900.92	2948.91	1463.55	1036.02		**66**	1888.69	2813.86	1370.16	1065.83
17	1878.87	2709.91	1513.39	1056.26		**67**	1911.72	2801.1	1554.77	1021.55
18	1891.06	2817.56	1568.9	1138.93		**68**	1855.03	2707.91	1341.36	1091.91
19	1904.82	3004.66	1306.4	1049.21		**69**	1813.26	3058.05	1441.57	1101.43
20	1969.05	3007.7	1319.69	1146.31		**70**	1894.75	2898.84	1411.8	1087.9
21	1961.37	3018.67	1470.13	1128.88		**71**	1883.13	2703.03	1560.01	1103.19
22	1830.67	2731.7	1305.94	1113.93		**72**	1875.85	2944.25	1469.16	1116.01
23	1820.36	2731.15	1495.77	1067.74		**73**	1993.35	2813.39	1505.27	1018.02
24	1870.7	2720.55	1523.33	1032.62		**74**	1957.18	2720.82	1465.04	1080.94
25	1810.63	2711.55	1361.48	1101.47		**75**	1984.11	3065.1	1357.22	1111.88
26	1999.34	2766.91	1334.5	1088.51		**76**	1976.01	3065.24	1403.71	1013.95
27	1973.38	3049.81	1491.79	1034.76		**77**	1954.28	2742.91	1496.27	1029.44
28	1906.19	3069.65	1397.96	1074.99		**78**	1909.58	2919.43	1520.11	1085.76
29	1938.14	3096.51	1494.62	1050.06		**79**	1849.94	2917.62	1558.36	1049.66
30	1999.41	2854.22	1555.52	1071.96		**80**	1866.62	2704.7	1528.08	1082.35
31	1859.86	2843	1481.35	1138.41		**81**	1929.85	2701.15	1435.14	1087.3
32	1873.52	3048.96	1460.99	1037.26		**82**	1867.71	2857.67	1509.87	1119.72
33	1842.11	2776.78	1578.97	1044.5		**83**	1804.62	3024.48	1312.1	1086.87
34	1861.19	2724.43	1559.84	1018.87		**84**	1866.55	3058.9	1544.04	1047.95
35	1894.88	2863.37	1485.12	1058.71		**85**	1936.87	3022.52	1562.53	1058.19
36	1991.32	2923.02	1387.88	1071.88		**86**	1936.04	3085.46	1546.61	1146.92
37	1919.07	2893.19	1426.67	1121.58		**87**	1851.38	2838.26	1535.43	1028.85
38	1931.5	2859.25	1588.76	1043.86		**88**	1929.94	3040.72	1421.71	1000.18
39	1970.13	2808.71	1414.1	1069.13		**89**	1889.24	2936.51	1300.53	1073.32
40	1999.5	3036.29	1327.59	1093.14		**90**	1971.91	2848.7	1434.42	1102.31
41	1830.07	2759.66	1419.04	1098.82		**91**	1933.4	2781.37	1468.95	1023.11
42	1946.29	2888.36	1596.32	1118.55		**92**	1803.17	3020.77	1581.79	1047.62
43	1950.17	3098.37	1540.46	1037.5		**93**	1984.22	2854.46	1437.52	1083.32
44	1840.35	2721.34	1594.53	1001.99		**94**	1832.09	2955.1	1402	1019
45	1948.38	2819.17	1462.7	1016.31		**95**	1820.91	2843.31	1398.96	1028.49
46	1927.67	2789.01	1308.31	1025.1		**96**	1990.4	2720.52	1332.16	1063.3
47	1883.99	2701.59	1515.8	1003.25		**97**	1883.18	3000.24	1396.45	1020.85
48	1960.71	2885.41	1545.37	1057.69		**98**	1979.95	2990.72	1398.64	1136.14
49	1909.35	2834.43	1479.62	1009.33		**99**	1881.92	2714.9	1507.85	1006.1
50	1950.94	2703.06	1399.54	1107.63		**100**	1877.03	2702.35	1385.83	1079.62
Avg of 100 Attempts	**1904.826**	**2890.503**	**1461.272**	**1067.995**						

Test Case: Existing and DmRT allocators allocate from Available Local Memory (SMP)

2. Fragmentation for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	51.24	36.11	27.46	20.9		**51**	53.31	36.95	26.38	23.43
2	54.84	36.21	26.67	22.72		**52**	51.11	33.82	26.89	22.62
3	51.44	34.45	27.8	22.85		**53**	53.75	36.85	28.86	21.36
4	53.81	35.26	26.45	21.11		**54**	50.62	34.74	27.18	22.64
5	53.39	35.41	27.98	20.67		**55**	52.54	34.07	28.53	22.15
6	50.02	33.35	27.22	22.73		**56**	50.31	33.9	28.75	22.55
7	53.12	35.77	25.56	23.43		**57**	53.41	33.22	28.53	22.7
8	52.67	36.49	25.35	21.1		**58**	52.28	33.07	26.9	21.34
9	52.58	34.75	28.87	20.07		**59**	51.42	35.51	27.76	22.22
10	52.14	36.08	25.52	23.39		**60**	50.66	36.09	25.29	21.39
11	54.24	36.41	27.71	22.71		**61**	50.06	35.94	28.36	20.6
12	51.13	35.4	28.26	21.11		**62**	52.94	33.2	25.36	23.88
13	51.86	35.38	28.06	21.17		**63**	52.66	35.85	25.29	22.18
14	52.19	35.38	25.26	22.52		**64**	51.41	36.48	25.44	23.49
15	54.07	36.64	25.28	21.38		**65**	54.2	33.73	27.11	21.27
16	54.62	35.94	25.06	21.41		**66**	51.3	36.52	27.36	23.84
17	54.59	36.45	27.29	23.69		**67**	50.72	35.26	28.34	20.78
18	51.11	36.08	27	20.8		**68**	51.74	33.51	27.08	22.79
19	50.21	36.76	25.4	20.86		**69**	52.45	35.48	27.9	22.45
20	53.73	35.82	26.11	21.54		**70**	51.25	36.3	26	22.45
21	52.11	35.66	28.59	21.74		**71**	54.15	35.84	25.66	23.06
22	50.92	33.4	28	22.96		**72**	51.25	33.72	28.17	21.03
23	51.23	33.81	26.48	23.03		**73**	54	34.62	27.63	22.44
24	51.77	36.2	28.84	23.6		**74**	51.3	35.93	28	22.53
25	52.39	36.27	27.56	20.95		**75**	53.63	34.88	27.02	23.42
26	54.85	35.67	27.41	20.87		**76**	53.06	34.98	27.05	21.83
27	50.13	34.2	28.98	23.07		**77**	54.4	35.12	26.27	21.4
28	50.39	33.13	27.3	21.27		**78**	53.54	33.2	27.92	23.51
29	51.25	33.74	27.1	22.31		**79**	52.81	34.55	25.57	21.08
30	50.4	33.63	28.17	23.18		**80**	53.79	33.31	26.3	21.92
31	53.15	36.79	28.91	23.93		**81**	52.3	35.02	25.73	21.07
32	51.87	34.61	27.38	23.31		**82**	53.46	35.53	26.11	21.82
33	53.56	34.84	28.47	23.04		**83**	52.47	35.6	25.92	21.88
34	53.27	33.21	27.46	20.7		**84**	53.87	35.78	27.91	22.57
35	54.21	34.65	28.46	23.82		**85**	54.98	36.55	25.14	21.17
36	50.99	36.97	25.45	21.56		**86**	53.85	34.76	28.4	20.46
37	50.14	35.37	27.99	23.35		**87**	51.98	35.29	26.89	20.78
38	51.58	36.67	25.42	20.86		**88**	52.41	36.72	28.95	23.67
39	51.32	34.61	27.3	22.07		**89**	54.24	34.19	26.2	20.71
40	50.8	34.89	25.12	22.35		**90**	54.25	36.55	27.68	20.53
41	53.52	36.83	25.55	20.38		**91**	53.71	33.7	25.16	20.43
42	51.57	33.67	25.27	23.09		**92**	53.68	35.12	26.04	21.43
43	50.44	36.35	26.99	21.57		**93**	50.11	35.28	27.88	23.64
44	54.02	35.51	26.71	20.69		**94**	51.1	35.01	27.84	23.88
45	50.77	34.86	26.78	23.37		**95**	50.2	34.83	26.2	23.3
46	54.34	34.79	28.25	20.4		**96**	50.24	35.53	28.46	22.07
47	52.74	33.12	27.57	20.15		**97**	54.39	35.04	26.37	21.5
48	51.91	36.82	26.97	22.3		**98**	50.82	34.13	26.09	23.51
49	53.31	35.26	27.4	20.91		**99**	53.23	35.35	26.51	23.79
50	52.44	33.78	25.51	22.5		**100**	53.51	35.63	27.97	22.97
Avg of 100 Attempts	**52.3926**	**35.157**	**27.0205**	**22.0902**						

Test Case: Existing and DmRT allocators allocate from Available Local Memory (SMP)

3. No. of request Satisfied for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	44.1	57.7	74.6	81.1		**51**	44.1	58.8	76.2	84.1
2	43.7	57.3	74.3	83.6		**52**	47.7	58.4	76.7	82.9
3	47.0	58.4	74.5	82.5		**53**	43.4	58.0	76.9	84.2
4	46.7	56.2	75.7	84.6		**54**	45.1	58.0	76.4	84.3
5	47.1	58.2	76.5	82.1		**55**	46.3	58.6	73.8	84.7
6	43.6	56.7	76.7	83.1		**56**	47.5	57.8	74.6	83.3
7	44.1	59.0	76.3	82.3		**57**	47.5	58.0	73.6	83.7
8	46.9	58.0	73.1	82.0		**58**	43.9	57.1	74.5	83.5
9	43.9	57.4	73.0	84.2		**59**	43.9	58.9	76.3	82.3
10	46.7	56.2	74.4	84.4		**60**	45.6	57.4	75.5	83.5
11	47.4	57.3	74.8	84.4		**61**	47.9	56.1	74.1	84.6
12	44.3	56.1	76.9	84.9		**62**	45.3	58.7	76.3	83.2
13	46.5	58.2	76.7	82.3		**63**	45.1	57.8	73.8	83.0
14	46.8	56.1	74.7	83.1		**64**	46.1	57.3	76.8	82.4
15	43.4	57.2	76.3	81.9		**65**	43.8	57.1	73.7	84.9
16	46.4	58.2	76.0	81.4		**66**	43.1	58.1	75.7	82.7
17	46.4	57.7	73.6	82.4		**67**	46.0	59.0	73.4	84.2
18	46.2	57.7	73.8	84.8		**68**	46.4	56.6	73.6	84.1
19	47.5	56.6	76.5	83.9		**69**	46.0	58.0	73.7	82.8
20	44.1	56.8	75.1	81.8		**70**	47.4	57.0	74.8	84.8
21	46.8	58.8	76.1	82.3		**71**	45.2	58.6	73.9	82.5
22	47.4	56.6	75.7	83.9		**72**	44.1	56.2	75.3	81.7
23	45.7	58.4	75.2	84.0		**73**	43.6	56.2	73.5	83.7
24	43.3	56.9	73.7	82.2		**74**	44.4	57.0	76.4	81.1
25	46.5	56.2	74.5	84.4		**75**	44.6	56.4	76.9	82.7
26	43.1	56.0	76.8	82.2		**76**	45.8	58.3	76.9	82.8
27	45.1	58.7	75.1	82.4		**77**	43.8	59.0	73.1	83.3
28	46.9	56.1	73.9	82.7		**78**	46.3	56.7	73.1	82.8
29	45.8	58.3	75.4	82.7		**79**	44.5	58.3	75.3	84.5
30	47.5	56.9	74.2	81.7		**80**	45.7	57.3	74.3	81.1
31	43.3	58.8	73.4	84.3		**81**	47.3	56.0	76.5	84.9
32	43.9	56.4	73.3	83.7		**82**	43.3	57.1	76.6	83.4
33	44.8	58.8	73.2	82.6		**83**	44.4	58.8	74.3	83.9
34	44.0	58.0	75.6	84.5		**84**	47.0	57.2	74.7	82.1
35	46.9	57.6	76.4	81.0		**85**	45.8	58.7	73.5	83.7
36	47.7	56.7	76.2	83.9		**86**	43.2	57.3	75.1	83.5
37	47.0	56.4	74.5	81.9		**87**	45.6	58.2	75.9	83.6
38	47.4	57.9	75.2	84.4		**88**	44.3	56.5	75.3	83.0
39	46.3	57.0	73.1	82.8		**89**	47.9	57.5	74.4	81.1
40	46.5	58.4	73.9	84.7		**90**	44.7	56.3	75.9	82.7
41	45.2	58.5	77.0	83.0		**91**	44.3	58.0	74.0	81.7
42	44.0	58.1	73.2	84.5		**92**	47.6	58.0	75.8	82.7
43	45.4	56.8	76.0	82.2		**93**	45.7	57.7	76.0	81.3
44	43.4	58.1	75.1	83.8		**94**	47.6	56.1	73.7	81.5
45	44.0	56.3	74.1	84.3		**95**	46.4	56.2	73.2	81.1
46	47.2	56.1	73.0	84.1		**96**	46.3	56.2	76.9	84.6
47	44.2	57.4	73.4	82.2		**97**	46.8	58.2	73.7	84.2
48	45.0	56.0	76.2	81.6		**98**	43.2	56.7	76.0	82.0
49	44.1	57.5	75.8	83.1		**99**	44.3	58.9	76.3	84.0
50	43.3	57.7	74.0	82.6		**100**	46.4	57.7	75.6	82.2
Avg of 100 Attempts	**45.458**	**57.4617**	**74.9894**	**83.109**						

Test Case: Existing and DmRT allocators allocate from Available Local Memory (SMP)

1. Execution Time for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	3276.71	4315.49	2109.34	1890.63		**51**	3177.93	4368.19	2159.45	1899.71
2	3275.88	4353.08	2184.18	1898.78		**52**	3254.06	4350.76	2125.61	1830.11
3	3268.1	4389.03	2169.11	1834.24		**53**	3185.35	4318.69	2155.41	1844.93
4	3289.33	4344.61	2113.54	1818.87		**54**	3128.39	4398.84	2100.14	1818.87
5	3284.92	4379.25	2129.49	1816.27		**55**	3280.51	4361.88	2146.9	1841.38
6	3118.69	4396.33	2167.11	1893.64		**56**	3145.16	4354.37	2125.73	1853.83
7	3103.2	4378.69	2184.27	1896.43		**57**	3205.75	4333.98	2177.52	1810.07
8	3283.6	4332.45	2164.84	1836.82		**58**	3209.58	4354.4	2188.47	1868.83
9	3235.02	4331.49	2180.49	1823.41		**59**	3263.87	4371.79	2199.44	1828.36
10	3263.98	4396.53	2192.42	1899.32		**60**	3139.43	4342.74	2199.97	1868.61
11	3172.78	4310.35	2103.63	1850.54		**61**	3267.34	4351.84	2121.1	1883.72
12	3210.57	4301.67	2138.62	1863.42		**62**	3175.02	4356.97	2175.39	1872.98
13	3155.86	4334.07	2126.83	1808.34		**63**	3230.35	4393.29	2167.46	1898.93
14	3177.62	4367.34	2157.43	1861.08		**64**	3192.04	4358.69	2103.97	1880.26
15	3269.22	4391.81	2166.53	1807.01		**65**	3269.51	4302.87	2170.92	1878.34
16	3123.68	4361.65	2103.4	1869.43		**66**	3125.16	4300.09	2188.76	1857.67
17	3223.57	4394.17	2196.39	1852.73		**67**	3163.22	4338.51	2188.12	1821.96
18	3250.55	4303.31	2101.5	1817.74		**68**	3119.7	4373.88	2176.86	1847.27
19	3253.95	4330.31	2156.98	1844.41		**69**	3224.42	4338.46	2158.14	1893.14
20	3242.7	4321.22	2133.61	1875.82		**70**	3220.12	4325.65	2177.19	1891.69
21	3151.55	4365.14	2182.62	1818.2		**71**	3208.83	4379.87	2190.73	1862.22
22	3175.06	4367.06	2179.03	1809.47		**72**	3109.78	4351.16	2147.15	1822.29
23	3260.01	4369.44	2192.8	1863.69		**73**	3126.2	4367.34	2129.86	1878.66
24	3115.65	4373.07	2140.52	1814.06		**74**	3257.94	4342.19	2102.44	1898.1
25	3123.27	4392.62	2160.08	1819.96		**75**	3171.47	4316.58	2185.56	1881.33
26	3254.54	4385.27	2152.64	1823.8		**76**	3192.33	4379.47	2115.06	1829.28
27	3133.24	4335.38	2186.27	1867.02		**77**	3193.14	4300.69	2177.25	1809.62
28	3150.5	4323.37	2112.87	1824.31		**78**	3243.28	4311.72	2113.2	1820.64
29	3115.91	4362.32	2177.89	1862.85		**79**	3297.93	4308.18	2150.03	1832.43
30	3125.42	4374.97	2106.43	1844.88		**80**	3174.82	4381.34	2130.03	1838.44
31	3109.49	4373.28	2110.19	1842.88		**81**	3242.84	4364.16	2142.46	1875.43
32	3263.59	4343.79	2195.51	1855.29		**82**	3242.41	4339.35	2199.28	1825.93
33	3178.17	4308.26	2115.41	1812.37		**83**	3240.43	4372.19	2190.1	1841.81
34	3230.74	4390.03	2159.11	1816.02		**84**	3218.25	4338.03	2155.81	1839.66
35	3207.8	4365.64	2173.93	1893.4		**85**	3180.45	4363.49	2139.32	1828.59
36	3135.44	4392.74	2125.05	1867.53		**86**	3253.84	4358.85	2131.4	1842.14
37	3291.36	4329.02	2102.53	1817.37		**87**	3209.46	4335.69	2191.96	1895.5
38	3139.45	4384.42	2150.46	1864.29		**88**	3266.06	4394.99	2196.38	1860.24
39	3207.81	4380.71	2169.81	1815.51		**89**	3199.3	4345.75	2121.19	1869.7
40	3226.01	4338.18	2102.97	1811.3		**90**	3126.82	4339.71	2193.63	1877.93
41	3162.87	4378.76	2156.76	1814.23		**91**	3281.97	4314.74	2161.36	1831.09
42	3147.32	4357.11	2171.32	1838.24		**92**	3174.66	4368.71	2131.3	1812.6
43	3294.04	4361.68	2160.75	1849.51		**93**	3259.55	4365.71	2182.49	1887.11
44	3176.73	4386	2154.23	1804.72		**94**	3209	4327.26	2193.32	1831.89
45	3253.6	4392.84	2157.08	1818.34		**95**	3285.1	4303.76	2156.01	1810.33
46	3289.49	4308.49	2126.72	1847.9		**96**	3168.81	4334.19	2134.14	1832.92
47	3193.5	4310.83	2110.08	1821.42		**97**	3162.97	4334.73	2147.6	1821.62
48	3116.6	4343.97	2147.96	1816.07		**98**	3221.52	4322.51	2174.58	1823.21
49	3282.19	4390.25	2185.66	1868.9		**99**	3189.18	4386.38	2172.3	1882.13
50	3244.71	4315.76	2170.72	1847.07		**100**	3136.42	4335.37	2111.58	1832.2
Avg of 100 Attempts	**3204.577**	**4352.133**	**2153.912**	**1847.152**						

Test Case: Existing and DmRT allocators allocate from Available Local Memory (SMP)

2. Fragmentation for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	58.43	44.2	33.41	28.03		**51**	59.35	41.44	30.62	26.82
2	59.85	42.7	31.61	27.04		**52**	59.73	43.73	33.98	25.65
3	60.44	45.83	30.81	25.71		**53**	58.47	42.3	32.58	25.38
4	59.91	45.83	33.78	27.76		**54**	61.31	44.82	33.64	27.34
5	59.62	41.2	30.65	25.51		**55**	61.05	43.08	30.6	25.4
6	60.02	45.57	32.78	28.89		**56**	61.18	41.95	31.12	28.36
7	60.17	41.77	33.64	28.2		**57**	62.29	45.88	33.9	27.59
8	61.65	44.97	31.7	27.64		**58**	58.2	45.51	31.95	25.18
9	62.64	43.44	33.83	25.96		**59**	58.54	46	31.57	25.3
10	61.13	43.09	30.05	27.1		**60**	58.75	45.17	30.12	28.14
11	60.73	43.57	33.51	28.86		**61**	58.91	42.19	31.6	27.74
12	61.39	45.48	33.86	26.16		**62**	60.23	45.97	33.52	25.38
13	59.43	43.9	32.47	26.56		**63**	62.21	45.52	32.29	25.02
14	60.52	41.42	32.46	27.99		**64**	58.53	42.74	30.38	27.82
15	60.13	42.54	32.51	28.43		**65**	61.95	44.14	33.31	28.85
16	61	44.06	31.06	25.11		**66**	61.38	44.39	30.94	28.03
17	62.51	44.28	31.19	27.44		**67**	61.72	42.93	31.65	26.27
18	59.51	43.07	33.03	27.34		**68**	61.62	43.98	32.35	28.95
19	58.96	43.65	31.11	25.3		**69**	60.33	41.59	31.19	26.5
20	62.55	41.47	30.85	28.75		**70**	60.41	43.04	30.82	27.71
21	62.24	43.68	33.71	28.57		**71**	60.4	45.12	31.26	26.14
22	58.25	44.1	30.9	25.45		**72**	61.63	43.07	30.64	27.68
23	59.19	45.81	31.89	26.51		**73**	60.75	42.48	33.88	27.66
24	59.42	41.22	33.16	27.77		**74**	59.87	41.03	32.29	26.88
25	60.17	44.77	30.1	28.36		**75**	61.47	42.55	30.48	27.95
26	58.17	42.07	33.01	27.94		**76**	60.55	43.38	33.07	28.32
27	62.03	42.62	30.19	27.97		**77**	59.69	45.91	33.88	26.88
28	59.14	43.22	32.23	26.48		**78**	58.38	45.1	31.43	28.99
29	61.8	44.35	30.61	27.52		**79**	61.12	42.84	33.55	28.6
30	58.67	42.86	33.95	27.38		**80**	62.14	43.66	30.64	28.84
31	61.41	41.63	33.52	25.94		**81**	62.9	44.65	30.24	25.57
32	58.87	44.94	30.16	27.77		**82**	59.61	43.12	32.71	26.88
33	62.55	43.7	30.46	26		**83**	61.84	45.67	30.52	27.12
34	62.54	44.02	32.1	27.38		**84**	58.51	42.89	32.77	25.23
35	60.48	43.25	33.15	27.31		**85**	59.38	45.56	33.59	26.76
36	59.03	42.39	32.41	25.84		**86**	59.5	45.38	33.13	28.45
37	61.69	42.38	32.18	27.48		**87**	58.36	44.91	33.95	26.42
38	59.75	44.03	32.55	25.68		**88**	62.92	44.44	31.81	27.45
39	62.15	44.33	31.53	25.99		**89**	61.84	42.06	30.72	25.14
40	62.99	43.38	32.98	28.88		**90**	62.23	43.36	31.58	26.06
41	59.44	43.94	32.7	26.75		**91**	59.54	43.17	32.94	25.44
42	58.51	41.12	31.35	27.93		**92**	60.99	45.64	33.61	28.62
43	60.46	41.88	32.15	26.51		**93**	60.19	44.55	31.87	27.48
44	62.8	45.43	32.82	25.33		**94**	59.46	44.03	31.13	25.76
45	62.17	42.64	31.7	26.28		**95**	60.92	43.23	33.54	28
46	60.69	41.77	31.83	25.42		**96**	61.7	45.26	32.82	27.08
47	58.41	44.78	33.53	28.13		**97**	59.16	44.86	30.41	26.2
48	60.98	43.91	32.19	25.17		**98**	58.37	42.13	30.58	25.55
49	59.17	42.98	30.92	25.18		**99**	58.44	44.71	30.61	28.46
50	61.39	42.52	30.3	28.26		**100**	60.72	44.3	31.96	25.48
Avg of 100 Attempts	**60.4389**	**43.6719**	**32.0433**	**26.9948**						

Test Case: Existing and DmRT allocators allocate from Available Local Memory (SMP)

3. No. of Request Satisfied for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	33.80	50.35	70.00	76.95		**51**	34.55	51.40	71.70	77.10
2	36.35	51.35	69.50	78.05		**52**	33.50	50.55	70.00	76.35
3	34.10	54.95	71.85	78.85		**53**	35.35	54.70	71.45	78.30
4	35.85	50.60	71.20	78.80		**54**	34.75	50.30	71.30	77.35
5	33.95	52.75	69.10	76.35		**55**	33.40	53.00	72.00	75.10
6	35.55	52.85	71.05	78.45		**56**	36.25	50.25	69.90	78.15
7	36.75	54.70	70.45	78.15		**57**	35.25	53.65	71.05	76.00
8	35.95	52.20	69.30	78.00		**58**	37.00	53.20	69.10	79.55
9	36.00	50.25	71.90	75.50		**59**	36.65	53.70	71.20	76.15
10	35.20	54.60	70.85	77.15		**60**	36.15	54.40	69.05	78.00
11	33.00	50.15	70.30	75.00		**61**	36.30	54.20	70.45	75.45
12	36.75	53.45	71.05	75.85		**62**	33.65	53.35	71.15	78.10
13	35.55	53.55	71.85	76.15		**63**	34.30	53.75	71.00	76.35
14	36.90	52.15	69.65	79.75		**64**	34.35	55.00	69.90	77.55
15	36.35	53.45	69.75	77.15		**65**	33.90	50.45	71.80	78.20
16	36.45	54.55	69.35	77.30		**66**	35.25	53.90	70.40	75.65
17	33.75	50.85	71.90	79.90		**67**	33.30	53.25	69.45	79.55
18	36.60	52.60	71.85	76.15		**68**	35.35	51.10	71.05	77.00
19	33.10	53.40	71.90	77.90		**69**	36.90	53.45	71.05	79.90
20	34.05	50.20	69.35	78.55		**70**	33.65	52.75	69.60	75.30
21	33.65	51.75	71.85	77.35		**71**	33.00	54.75	71.65	77.30
22	34.35	51.75	71.75	78.55		**72**	35.10	50.10	71.80	77.95
23	34.05	53.65	71.20	78.35		**73**	34.15	52.20	69.00	76.05
24	36.45	50.05	70.05	77.25		**74**	34.80	54.05	69.20	78.20
25	33.95	52.95	71.90	77.00		**75**	35.10	52.35	70.85	78.55
26	36.60	52.00	69.20	76.65		**76**	34.90	50.95	69.25	76.00
27	36.05	51.00	69.55	78.65		**77**	35.95	52.00	70.00	77.45
28	33.10	54.55	71.85	78.85		**78**	34.60	54.20	70.35	78.35
29	34.35	51.00	69.00	78.95		**79**	36.90	53.35	70.75	78.25
30	36.75	53.45	70.45	79.70		**80**	35.85	52.70	70.45	77.00
31	36.50	51.70	69.40	79.05		**81**	35.15	53.40	70.65	75.30
32	33.95	53.60	71.90	79.80		**82**	34.90	50.50	70.05	76.75
33	35.55	50.95	71.85	76.70		**83**	36.30	54.05	71.25	76.70
34	33.10	54.45	71.15	78.15		**84**	36.75	50.70	69.45	76.65
35	35.80	54.30	70.85	77.75		**85**	34.85	54.55	69.30	77.65
36	35.65	53.00	70.40	76.50		**86**	34.65	50.70	69.45	77.05
37	33.45	52.05	70.20	76.65		**87**	35.00	50.85	69.60	79.45
38	35.70	50.20	70.90	77.30		**88**	34.20	53.00	69.85	76.20
39	35.30	52.25	71.50	78.75		**89**	34.15	53.60	69.85	78.20
40	36.15	54.90	71.40	75.10		**90**	33.95	53.20	69.50	76.65
41	35.65	53.60	70.10	77.10		**91**	35.60	51.45	69.05	78.20
42	34.25	51.50	70.00	78.30		**92**	35.40	51.55	70.70	78.95
43	34.40	54.90	71.40	75.20		**93**	36.70	53.90	70.70	75.40
44	36.80	51.05	71.10	76.25		**94**	37.00	54.65	71.00	77.50
45	34.15	53.50	70.30	76.00		**95**	33.70	50.35	71.95	75.70
46	35.40	54.25	70.35	79.40		**96**	34.55	50.50	69.65	78.75
47	33.25	51.15	69.80	79.75		**97**	33.60	54.00	71.95	79.60
48	34.30	50.80	71.65	75.55		**98**	33.80	54.35	69.30	77.30
49	35.25	51.35	70.35	79.30		**99**	35.90	54.30	71.85	77.80
50	36.25	51.50	71.80	77.40		**100**	36.00	51.05	70.50	75.75
Avg of 100 Attempts	**35.0845**	**52.5575**	**70.5685**	**77.47**						

Case 1: Existing from Local and DmRT follows Local -> Shared -> Ideal

1. Execution Time for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	341.96	406.69	298.27	393.29		**51**	330.87	419.99	281.97	374.71
2	311.37	415.58	295	368.32		**52**	339.8	417.45	290.82	351.28
3	329.7	414.63	286.39	353.85		**53**	336.01	419.33	286.52	367.52
4	320.77	411.41	291.08	353.35		**54**	343.85	405.68	296.31	387.9
5	323.16	416.49	286.24	375.69		**55**	313.29	411.02	281.41	393.93
6	338.61	414.21	290.74	385.19		**56**	321.12	406.5	290.32	372.9
7	311.72	419.85	285.79	389.9		**57**	300.78	410.47	281.89	358
8	325.4	402.63	284.36	362.24		**58**	302.81	408.62	293.62	388.93
9	306.24	408.78	289.99	370.56		**59**	323.8	404.87	286.69	353.48
10	318.2	404.29	298.98	352.12		**60**	323.13	417.92	284.99	388.35
11	337.47	402.31	294.83	365.16		**61**	333.12	409.18	292.88	368.34
12	327.68	418.88	293.87	382.04		**62**	338.16	411.05	283.57	376.08
13	332.1	404.16	286.78	362.41		**63**	303.07	408.05	295.32	378.36
14	312.31	417.95	283.74	391.11		**64**	345.93	405.54	291.41	368.1
15	341.61	404.96	293.83	360.22		**65**	300.86	416.9	280.97	381.08
16	301.82	414.45	289.57	377.06		**66**	348.62	416.19	281.43	354.78
17	338.64	412.14	285.16	357.39		**67**	325.49	407.03	289.68	357.04
18	344.43	413.11	296.76	391.91		**68**	323.29	410.17	296.34	369.04
19	335.31	412.84	296.78	366.81		**69**	315.12	408.85	282.05	381.27
20	348.84	403.64	289.18	372.45		**70**	331.37	416.52	288.22	356.52
21	339.61	403.47	296.95	372.78		**71**	332.99	416.02	297.28	367
22	310.64	403.3	297.99	356.21		**72**	322.69	412.1	283.01	381.65
23	303.29	406.44	295.08	387.05		**73**	336.8	417.18	282.5	398.04
24	302.4	404.96	283.35	357.9		**74**	305.66	408.64	294.43	394.97
25	334.78	404.92	285.51	359.67		**75**	330.99	413.99	281.65	399.86
26	325.03	406.83	295.64	399.54		**76**	301.4	416.85	294.6	391.38
27	328.94	410.59	283.47	387.55		**77**	317.01	416.76	284.54	368.35
28	328.54	416.2	282.46	396.18		**78**	312.23	409.93	294.78	379.25
29	319.83	409.39	291.99	374.12		**79**	339.81	409.98	299.08	383.12
30	302.38	404.83	297.09	369.08		**80**	316.81	413.24	295.93	352.96
31	347.89	417.47	290.65	361.39		**81**	311.88	409.04	294.34	362.16
32	319.99	414.09	282.86	394.41		**82**	330.68	406.56	283.98	367.18
33	302.16	409.9	288.15	378.78		**83**	340.27	417.09	296.59	355.61
34	340.06	407.65	295.44	358.62		**84**	303.7	413.33	289.34	379.02
35	346.16	411.34	287.71	392.85		**85**	334.53	416.1	282.96	355.12
36	322.15	411.47	285.06	362.14		**86**	333.77	411.05	299.71	367.15
37	336.01	407.54	289.59	393.89		**87**	343.9	416.63	296.81	371.2
38	326.59	409.21	288.26	367.34		**88**	334.47	402.39	292.62	379.85
39	305.84	413.71	283.42	352.24		**89**	324.72	419.95	290.6	371.29
40	340.14	419.24	285.27	371.75		**90**	331.86	417.96	290.45	351.38
41	345.22	409.42	296.07	395.72		**91**	347.19	400.96	280.1	379.28
42	347.25	410.08	296.65	385.65		**92**	303.29	411.52	293.76	399.85
43	344.13	405.19	289.71	361.31		**93**	345.61	411.57	299.05	379.93
44	317.02	403.89	291.78	383.18		**94**	342.28	403.67	280.39	359.26
45	310.09	410.76	284.38	354.37		**95**	332.24	417.7	292.03	386.28
46	324.21	416.37	289.21	373.66		**96**	305.35	403.27	296.67	380.55
47	311.48	408.61	292.76	397.43		**97**	338.29	400.32	293.92	377.38
48	324.89	412.97	295.23	361.43		**98**	336.74	410.65	295.38	395.46
49	327.07	418.16	297.83	389.65		**99**	344.37	406.43	294.77	393.25
50	310.41	404.05	284.51	360.83		**100**	330.7	407.42	291.17	395.83
Avg of 100 Attempts	**326.2426**	**410.8068**	**290.2026**	**374.3901**						

Case 1: Existing from Local and DmRT follows Local -> Shared -> Ideal

2. Fragmentation for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	44.5	37.07	23.97	9.66		**51**	41.31	38.12	19.51	10.34
2	45.53	34.36	21.77	11.08		**52**	41.3	37.46	22.99	11.4
3	42.27	36.11	22.99	11.64		**53**	43.33	36.73	19.28	9.36
4	44.87	37.75	21.6	12.89		**54**	42.27	38.08	20.44	8.81
5	44.67	34.09	21.85	10.87		**55**	45.24	36.1	21.1	9.91
6	41.83	35.05	22.22	11.07		**56**	45.54	38.7	22.51	9.38
7	41.37	34.54	19.85	8.71		**57**	45.37	34.48	20.48	8.07
8	43.37	36.81	21.89	12.22		**58**	42.69	38.3	21.72	10.37
9	42.67	34.19	21.7	12.77		**59**	45.09	36.19	23.45	11.78
10	44.49	34.49	23.82	10.29		**60**	44.81	37.9	23.1	8.11
11	42.13	36.52	22.01	13		**61**	44.68	38.01	19.42	8.64
12	42	37.7	22.21	11.86		**62**	45.74	37.84	22.07	12.81
13	45.13	34.44	19.95	10.62		**63**	45.17	35.94	20.96	12.89
14	41.71	34.85	22.71	12.38		**64**	41.32	34.63	22.4	12.22
15	42.2	36.33	20.21	11.84		**65**	43.83	37.91	22.84	8.34
16	43.19	37.75	20.41	10.45		**66**	44.73	34.75	20.13	12.16
17	42.97	37.1	21.24	10.54		**67**	43.21	38.44	19.95	12.33
18	43.33	35.18	23.92	12.89		**68**	44.58	37.31	22.3	11.19
19	41.3	38.79	20.61	8.6		**69**	42.07	37.91	21.48	10.22
20	41.47	36.02	19.25	8.22		**70**	42.43	38.51	23.75	8.74
21	43.3	38.9	22.72	12.26		**71**	44.51	36.45	23.52	10.42
22	44.6	38.62	21.61	12.83		**72**	41.52	38.57	21.06	12.03
23	41.34	38.24	23.55	10.69		**73**	42.13	37.47	19.71	9.01
24	44.17	36.87	23.09	9.5		**74**	44.72	35.22	19.85	8.51
25	45.37	38.17	23.64	10.66		**75**	43.91	37.23	22.3	9.59
26	41.44	38.81	22.81	10.48		**76**	45.34	36.02	21.97	8.06
27	43.9	36.6	19.84	10.87		**77**	43.75	36.67	22.48	9.51
28	43.09	38.81	19.26	11.45		**78**	45.15	35.03	23.09	9.09
29	44.76	34.57	21.93	10.02		**79**	45.04	37.46	22.15	10.27
30	42.04	34.02	21.03	10.55		**80**	42.47	36.52	22.18	8.66
31	44.01	38.57	20.5	10.18		**81**	45.23	35.9	23.48	10.13
32	42.6	36.99	19.94	9.88		**82**	43.03	38.08	22.95	10
33	44.7	34.08	22.15	12.43		**83**	42.67	37.1	21.11	10.31
34	45.15	37.69	21.04	11.36		**84**	42.31	36.02	20.09	9.83
35	43.31	35.29	21.23	9.14		**85**	44.33	36.5	21.62	10.47
36	43.67	38.78	20.22	10.65		**86**	43.23	37.05	23.16	9.59
37	41.47	34.82	22.55	11.95		**87**	42.36	38.7	23.39	9.26
38	42.82	35.23	20.81	9.81		**88**	43.21	36.58	23.2	12.47
39	42.43	36.27	23.51	9.36		**89**	45.81	35.28	19.8	9.79
40	41.85	38.95	20.23	11.79		**90**	41.13	36.35	22.36	10.95
41	42.33	38.76	20.55	9.49		**91**	43.5	36.48	22.22	8.94
42	44.61	35.81	23.77	12.63		**92**	42.16	34.41	19.43	10.47
43	44.6	38.88	20.22	10.75		**93**	44.37	38.07	19.06	8.67
44	41.08	38.95	20.64	8.26		**94**	42.16	36.16	21.93	8.66
45	44.83	36.88	19.89	9.05		**95**	42.17	36.26	23.58	12.53
46	44.7	34.26	21.66	10.45		**96**	43.63	34.01	21.81	10.11
47	45.31	37.29	20.28	8.79		**97**	44.88	35.18	20.02	12.55
48	41.96	37.04	23.02	12.65		**98**	44.27	35.98	20.08	11.88
49	45.45	35.74	19.21	12.77		**99**	45.04	35.28	22.54	8.39
50	45.79	38.76	20.17	11.54		**100**	43.95	37.36	23.47	10.35
Avg of 100 Attempts	**43.50**	**36.68**	**21.59**	**10.51**						

Case 1: Existing from Local and DmRT follows Local -> Shared -> Ideal

3. No. of Request Satisfied for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	56	63	75	95		**51**	60	63	78	95
2	56	62	76	95		**52**	56	61	75	96
3	60	62	78	97		**53**	60	62	77	95
4	56	64	76	95		**54**	57	64	79	95
5	57	61	75	94		**55**	58	62	77	96
6	58	62	76	93		**56**	56	61	78	94
7	60	62	77	97		**57**	56	65	78	95
8	56	62	75	93		**58**	57	61	78	95
9	57	61	77	95		**59**	60	63	75	96
10	58	64	76	93		**60**	57	63	76	95
11	57	61	79	95		**61**	58	63	77	94
12	59	60	74	94		**62**	59	60	74	92
13	59	64	79	95		**63**	57	62	78	96
14	55	62	78	96		**64**	58	64	75	93
15	56	64	78	95		**65**	60	64	79	92
16	59	62	78	93		**66**	56	62	77	95
17	57	61	76	94		**67**	60	61	74	92
18	58	62	79	95		**68**	56	65	75	93
19	55	61	79	94		**69**	59	64	79	96
20	55	63	75	95		**70**	55	65	76	94
21	59	64	79	96		**71**	60	62	75	95
22	56	61	77	97		**72**	59	63	75	93
23	60	64	77	96		**73**	57	62	79	94
24	56	61	78	92		**74**	57	62	76	93
25	56	61	75	93		**75**	60	60	75	97
26	56	60	77	96		**76**	58	64	78	95
27	60	64	77	92		**77**	57	62	75	92
28	60	64	78	93		**78**	60	61	78	95
29	55	62	77	97		**79**	58	60	78	94
30	59	61	77	96		**80**	58	61	78	95
31	58	63	76	95		**81**	58	64	74	97
32	58	60	78	94		**82**	58	60	79	97
33	55	61	75	95		**83**	60	63	76	94
34	57	64	74	93		**84**	58	61	75	95
35	55	64	79	97		**85**	55	64	74	94
36	60	64	76	92		**86**	58	61	74	97
37	56	62	76	95		**87**	57	63	77	96
38	60	62	79	97		**88**	55	65	76	96
39	58	60	77	95		**89**	56	64	75	95
40	60	63	76	94		**90**	57	60	74	94
41	59	64	79	96		**91**	56	61	76	96
42	59	62	77	92		**92**	55	64	79	96
43	55	61	75	93		**93**	58	63	77	97
44	57	62	76	97		**94**	58	65	77	95
45	57	61	79	95		**95**	59	61	77	93
46	57	64	77	92		**96**	59	60	77	93
47	56	64	76	96		**97**	56	62	78	94
48	58	61	76	93		**98**	59	62	76	93
49	58	65	75	96		**99**	56	64	76	94
50	57	64	78	95		**100**	57	62	74	97
Avg of 100 Attempts	**57.5068**	**62.3301**	**76.6088**	**94.6614**						

Case 1: Existing from Local and DmRT follows Local -> Shared -> Ideal

1. Execution Time for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	2019.94	2919.8	1449.83	2380.93		**51**	2041.16	2830.52	1672.48	2372.22
2	1975.91	2834.86	1360.55	2344.18		**52**	1958.95	2931.8	1687.8	2379.66
3	1986.51	2975.02	1593.76	2367.7		**53**	2075.66	2932.15	1638.35	2355.73
4	1973.94	2858.23	1575.51	2237.03		**54**	2038.4	3200.18	1431.11	2213.1
5	2038.76	2911.8	1451.6	2271.93		**55**	1914.82	2940.16	1649.61	2320.75
6	1997.75	3064.49	1562.78	2289.13		**56**	1901.82	3023.58	1488.49	2289.57
7	2096.33	3071.17	1478.77	2363.4		**57**	2066.87	2975.69	1565.68	2281.19
8	1968.31	3012.5	1610.85	2395.72		**58**	2062.56	2948.25	1685.75	2385.87
9	1929.27	3006.98	1674.03	2255.17		**59**	2039.23	3039.59	1503.78	2349.4
10	2045.43	3187.03	1626.12	2304.1		**60**	1997.83	2873.6	1434.89	2365.57
11	2036.7	2801.17	1329.42	2226.66		**61**	2015.15	2903.04	1662.25	2380.33
12	1995.62	2901.72	1456.22	2392.09		**62**	2064.3	2819.16	1683.41	2333.87
13	2034.2	3099.26	1443.55	2209.67		**63**	2032.04	3179.02	1459.28	2256.83
14	2099.23	2854.39	1555.85	2344.81		**64**	1990.31	2817.53	1678.85	2230.97
15	2024.05	2838.6	1516.28	2215.67		**65**	1943.89	3046.49	1342.02	2287.3
16	1905.85	2855.63	1509.04	2396.89		**66**	2053.12	2972.21	1639.21	2377.04
17	2041.47	3153.95	1455.8	2295.03		**67**	1967.89	3070.85	1567.45	2306.31
18	2093.23	3176.76	1481.94	2385.01		**68**	2042.65	2920.36	1304.75	2222.75
19	2097.8	2911.72	1655.32	2373.83		**69**	2087.2	2898.03	1398.1	2368
20	2098.24	3130.28	1331.57	2283.23		**70**	1957.56	3067.95	1656.7	2323.73
21	1938.53	2862.16	1432.54	2221.01		**71**	2097.01	2947.05	1565.04	2270.54
22	1905.65	2959.85	1656.91	2298.6		**72**	1954.36	3185.68	1348.55	2293.53
23	2072.92	3062.53	1531.29	2379.02		**73**	2013.68	2896.72	1438.95	2379.64
24	2040.81	2883.03	1680.37	2208.67		**74**	1965.36	2808.31	1594.17	2224.02
25	1921.97	2968.58	1590.6	2397.92		**75**	2017.03	2921.88	1663.72	2346.04
26	1917.47	2992.48	1690.79	2206.38		**76**	2029.53	3114.97	1363.06	2263.31
27	1909.26	3111.65	1643.18	2306.16		**77**	2062.55	3152.32	1412.02	2245.39
28	2087.72	2920.66	1657.57	2251.92		**78**	1977.78	3085.65	1576.51	2249.53
29	2084.11	3062.95	1390.97	2297.59		**79**	1930.2	2910.07	1446.57	2250.94
30	1962.65	2820.28	1446.21	2283.97		**80**	1969.53	3197.37	1312.21	2381.78
31	2062.94	2920.02	1508.89	2341.84		**81**	1942.94	3200.33	1693.6	2374.24
32	2052.23	3041.17	1510.61	2353.43		**82**	1944.25	3058.63	1458.36	2255.21
33	2043.27	3061.96	1483.94	2322.38		**83**	2077.44	2804.44	1681.17	2247.64
34	1921.06	3058.09	1500.58	2215.02		**84**	2051.76	2900.44	1342.41	2243.17
35	2065.77	3043.57	1404.67	2268.31		**85**	2043.75	3182.87	1325.95	2227.22
36	2097.64	3059.21	1391.71	2265.42		**86**	2096.35	2929.11	1588.45	2254.57
37	2015.11	3059.98	1373.26	2309.62		**87**	2037.23	2814.88	1518.92	2373.61
38	2072.48	2801.31	1572.04	2337.35		**88**	1978.55	3129.56	1640.97	2228.97
39	2054.11	2919.8	1663.08	2249.21		**89**	2032.7	2856.09	1597.51	2357.61
40	2035.35	3198.27	1314.32	2375.7		**90**	1900.15	3100.59	1639.01	2216.77
41	2003.86	2972.94	1511.87	2322.55		**91**	2023.28	3158.37	1680.49	2239.46
42	2025.24	2967.1	1353.6	2358.79		**92**	2003.29	2960.95	1482.71	2340.29
43	2011.26	2847.86	1500.46	2332.3		**93**	2084.67	2984.85	1682.22	2367.11
44	2056.55	2903.24	1546.86	2322.88		**94**	1991.23	2947.53	1446.71	2216.28
45	1947.07	3156.75	1478.09	2335.24		**95**	2080.85	3013.36	1572.69	2281.67
46	1910.66	3013.83	1466.53	2282.45		**96**	2084	3110.5	1584.79	2337.14
47	2000.15	2995.95	1581.43	2225.56		**97**	2077.76	3158.51	1532.53	2249.99
48	2024.15	2953.36	1518.77	2219.97		**98**	1919.09	2822.33	1628.95	2376.34
49	1992.44	2848.45	1447.51	2224.4		**99**	2001.72	3051.42	1557.47	2371.25
50	2060.66	3075.21	1300.37	2393.14		**100**	1941.34	2944.89	1440.06	2218.76
Avg of 100 Attempts	**2013.324**	**2988.474**	**1522.335**	**2303.212**						

Case 1: Existing from Local and DmRT follows Local -> Shared -> Ideal

2. Fragmentation for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	54.71	42.68	28.49	14.22		**51**	52.46	42.72	27.44	16.22
2	51.41	45.77	31.39	14.43		**52**	51.84	43.2	31.57	14.28
3	50.61	42.81	27.14	16.74		**53**	53.29	43.41	29.75	14.94
4	52.43	45.21	29.89	16.46		**54**	50.28	46.23	27.31	13.16
5	54.96	44.88	30.1	17.01		**55**	53.52	42.61	28.76	15.22
6	53.52	46.66	31.97	15.85		**56**	54.79	46.62	28.35	14.73
7	52.86	45.27	31	14.54		**57**	52.04	46.51	28	16.33
8	50.77	42	28.21	16.37		**58**	54.27	44.25	30.94	17.86
9	52.29	46.27	31.57	13.52		**59**	53.33	44.38	28.73	16.34
10	51.53	45.93	29.39	13.74		**60**	52.85	44.84	27.01	15.73
11	52.06	42.4	30.08	17.3		**61**	52.62	46.16	29.32	13.54
12	54.03	42.43	28.82	14.28		**62**	52.28	45.59	31.71	17.69
13	52.35	46.97	29.08	14.59		**63**	51.2	42.26	28.1	16.76
14	50.73	42.51	30.12	17.08		**64**	53.73	46.8	30.86	16.67
15	52.83	45.11	29.68	15.73		**65**	53.7	43.65	28.54	16.59
16	50.12	42.17	29.08	15.01		**66**	50.98	45.34	30.46	16.87
17	51.4	46.12	29.07	15.49		**67**	52.16	44.69	30.53	13.81
18	54.57	43.36	31.74	16.65		**68**	53.28	44.27	30.75	13.33
19	52.24	43.92	30.98	17.7		**69**	52.86	44.6	27.91	14.03
20	53.76	43.62	31.73	13.64		**70**	51.01	42.89	30.04	13.23
21	51.91	43.45	29.31	17.62		**71**	52.61	44.05	30.99	16.57
22	54.11	44.48	27.92	14.15		**72**	54.9	46.09	27.54	15.34
23	54.7	45.51	27.95	17.3		**73**	51.52	44.23	27.27	17.7
24	51.96	43.57	30.49	17.77		**74**	52.6	42.3	28.92	13.95
25	53.29	46.06	31.55	14.77		**75**	52.6	45.48	27.67	14.3
26	52.22	42.99	29.01	16.47		**76**	50.67	46.06	31.38	16.89
27	52.77	46.35	30.87	16.24		**77**	52.63	45.5	29	14.42
28	52.54	45.77	29.67	17.72		**78**	53.77	46.05	30.13	15.47
29	50.92	42.88	31.28	14.92		**79**	54.17	44.9	30.27	15.72
30	52.17	46.6	28.53	15.35		**80**	53.44	46.64	28.69	17.93
31	53.63	43.59	31.51	14.89		**81**	50.21	44.96	32	15.24
32	52.81	42.62	31.09	17.42		**82**	52.09	43.57	27.39	17.97
33	51.42	44.5	28.01	15.68		**83**	54.26	42.42	27.69	13.48
34	51.49	44.84	31.97	16.86		**84**	50.63	43.77	29.42	17.13
35	50.78	46.54	30.01	13.01		**85**	50.77	46.73	30.36	14.73
36	53.52	44.03	30.39	13.03		**86**	53.19	42.76	27.37	13.91
37	52.5	43.41	31.64	14.01		**87**	54.72	44.95	27.89	17.78
38	54.55	46.4	27.24	15.89		**88**	54.91	43.43	28.83	15.56
39	50.46	42.29	28.95	16.3		**89**	54.51	43.24	31.28	13.48
40	51.77	43.68	28.1	15.75		**90**	52.72	44.48	30.69	13.19
41	51.4	45.44	27.91	17.36		**91**	51.45	42.98	30.55	15.66
42	51.62	43.29	30.78	17.59		**92**	53.45	44.51	30.79	13.52
43	53.91	46.04	30.77	15.95		**93**	51.16	45.57	30.63	15.04
44	53.44	46.82	31.79	17.93		**94**	50.09	42.13	28.27	17.19
45	50.3	45.89	27.89	16.94		**95**	53.7	45.84	29.38	14.24
46	53.24	42.37	30.15	16.99		**96**	54.5	46.15	29.42	17.35
47	50.18	42.03	29.9	13.2		**97**	54.07	43.42	27.06	16.71
48	52.51	46.38	29.59	15.9		**98**	51.06	43.87	31.98	13.35
49	53.64	46.81	28.61	15.25		**99**	54.49	42.89	28.94	16.15
50	51.56	43.84	27.06	16.99		**100**	51.03	44.89	31.32	15.51
Avg of 100 Attempts	**52.5491**	**44.4944**	**29.5867**	**15.6241**						

Case 1: Existing from Local and DmRT follows Local -> Shared -> Ideal

3. No. of Request Satisfied for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	46.1	52.1	63.4	88.5		**51**	46.9	54.2	67.7	86.2
2	46.6	52.5	67.5	85.6		**52**	46.9	53.0	64.6	87.9
3	44.2	52.6	64.4	88.6		**53**	46.5	55.9	63.7	85.5
4	45.1	55.7	67.7	86.4		**54**	47.2	54.4	67.2	86.5
5	43.2	56.9	67.9	89.4		**55**	44.2	53.6	63.5	86.4
6	43.1	55.3	63.6	87.2		**56**	46.2	52.7	66.0	88.2
7	44.4	55.0	67.6	89.9		**57**	46.4	56.9	65.9	87.8
8	46.7	52.9	64.5	87.3		**58**	44.4	54.2	66.2	86.4
9	44.5	52.4	65.1	86.3		**59**	44.6	52.1	65.2	88.0
10	45.1	56.0	63.1	87.9		**60**	45.4	53.4	64.3	90.0
11	46.0	54.4	65.8	88.2		**61**	47.0	52.0	66.3	89.4
12	43.8	53.3	66.5	87.1		**62**	43.5	54.1	63.1	85.7
13	46.7	53.1	64.4	85.7		**63**	47.5	55.8	65.7	89.1
14	47.8	53.5	65.5	89.8		**64**	45.0	53.6	67.5	85.5
15	43.6	53.2	63.5	87.4		**65**	43.5	52.8	65.7	88.2
16	45.7	53.8	66.8	88.3		**66**	44.3	52.4	67.6	87.9
17	44.4	52.8	66.3	85.6		**67**	45.1	55.3	66.0	85.3
18	43.6	55.4	67.9	85.7		**68**	47.8	54.0	63.0	88.7
19	47.9	52.1	66.7	85.1		**69**	43.5	54.6	63.7	85.7
20	43.6	56.6	64.7	85.2		**70**	43.1	52.6	63.2	87.8
21	44.3	56.1	67.6	86.0		**71**	44.9	53.8	65.3	88.9
22	46.7	54.8	67.4	88.3		**72**	43.5	53.1	65.2	89.3
23	43.7	56.1	67.1	89.1		**73**	45.4	52.6	65.8	87.0
24	44.8	53.2	65.9	85.8		**74**	46.7	54.2	67.5	85.4
25	47.7	52.9	67.7	88.8		**75**	43.2	54.7	67.7	87.5
26	43.3	53.0	65.1	89.7		**76**	46.0	56.2	65.0	86.0
27	47.3	55.5	67.4	85.6		**77**	43.9	55.2	66.7	86.7
28	45.1	54.1	66.0	85.4		**78**	46.7	56.9	65.5	89.2
29	45.5	54.1	64.2	89.9		**79**	45.0	56.2	66.8	86.5
30	43.1	52.3	63.8	85.2		**80**	43.9	56.2	64.3	85.3
31	43.1	56.8	67.2	89.9		**81**	45.3	55.8	67.2	88.3
32	43.2	56.2	64.2	85.4		**82**	43.5	54.2	64.5	85.1
33	46.3	53.0	67.7	89.9		**83**	46.7	56.3	66.9	88.9
34	46.5	52.2	64.1	89.1		**84**	45.0	52.1	64.4	86.5
35	45.5	55.8	66.6	87.3		**85**	43.7	52.7	67.3	88.3
36	45.4	55.1	67.6	86.0		**86**	45.8	52.3	63.4	89.8
37	45.6	54.4	67.6	89.6		**87**	43.1	55.0	64.8	86.4
38	47.0	52.3	65.1	86.7		**88**	46.4	56.3	64.4	87.5
39	43.7	52.2	67.3	87.2		**89**	43.6	54.7	63.5	88.2
40	43.5	56.4	63.1	86.1		**90**	44.8	54.9	67.3	89.9
41	46.7	56.5	63.5	86.6		**91**	47.7	55.5	66.8	85.5
42	48.0	53.5	64.4	85.8		**92**	43.4	56.6	63.3	88.2
43	46.9	56.1	64.2	87.5		**93**	44.5	54.5	65.3	89.4
44	45.0	52.4	67.5	89.0		**94**	43.5	55.7	65.1	85.3
45	46.9	55.5	67.6	88.2		**95**	47.4	52.5	63.9	87.8
46	44.2	53.0	66.8	85.4		**96**	45.6	53.3	66.0	89.9
47	43.4	52.3	63.6	88.8		**97**	45.2	53.7	63.7	88.3
48	47.4	55.3	66.0	85.1		**98**	47.1	56.6	67.3	86.3
49	44.9	53.3	63.4	86.3		**99**	46.3	54.9	64.9	89.0
50	46.9	54.5	65.6	87.7		**100**	44.3	53.0	67.4	89.4
Avg of 100 Attempts	**45.2403**	**54.2546**	**65.6095**	**87.4181**						

Case 1: Existing from Local and DmRT follows Local -> Shared -> Ideal

1. Execution Time for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	3176.23	4385.59	2087.63	3364.51		**51**	3121.89	4346.96	2046.34	3392.71
2	3292.47	4311.71	2008.64	3394.4		**52**	3197.17	4374.28	2037.07	3313.05
3	3213.74	4387.95	2049.54	3401.04		**53**	3176.07	4320.8	2051.85	3352.87
4	3109.66	4315.37	2015.25	3401.93		**54**	3126.23	4388.58	2039.8	3310.34
5	3167.11	4305.77	2005.98	3361.23		**55**	3294.46	4335.86	2078.6	3342.02
6	3251.2	4308.96	2064.08	3394.74		**56**	3101.62	4321.29	2079.64	3318.28
7	3290.37	4300.64	2070.61	3381.88		**57**	3200.58	4336.71	2017.82	3330.87
8	3185.84	4329.7	2027.06	3393.77		**58**	3187.92	4397.99	2089.64	3328.08
9	3274.74	4384.05	2073.55	3360.99		**59**	3147.32	4369.48	2009.38	3366.56
10	3273.34	4320.93	2067.71	3312.79		**60**	3187.43	4370.62	2093.44	3371.57
11	3128.32	4389.55	2059.49	3324.06		**61**	3259.99	4309.38	2086.67	3378.03
12	3140	4364.21	2068.59	3365.15		**62**	3168.57	4362.77	2067.56	3348.38
13	3157.82	4348.99	2022.13	3334.97		**63**	3234.69	4347.66	2043.79	3315.66
14	3214.77	4367.52	2003.55	3384.8		**64**	3251.81	4367.06	2029.28	3408.17
15	3150.68	4315.28	2043.58	3347.53		**65**	3219.44	4336.52	2099.17	3380.36
16	3175.95	4315.62	2030.93	3386.84		**66**	3137.28	4323.97	2082.45	3313.47
17	3260.03	4340.98	2058.03	3368.2		**67**	3290.07	4392.02	2010.59	3318.9
18	3266.39	4317.86	2036.98	3406.84		**68**	3151.65	4344.91	2054.92	3378.41
19	3184.64	4343.34	2042.24	3359.89		**69**	3269.75	4339.88	2084.33	3342.55
20	3157.01	4316.69	2041.32	3345.03		**70**	3285.53	4300.48	2074.68	3334.06
21	3176.03	4381.91	2089.18	3400.45		**71**	3284.41	4313.95	2053.9	3377.38
22	3249.59	4368.28	2077.05	3385.05		**72**	3281.57	4300.79	2025.26	3408.68
23	3270.38	4322.5	2023.37	3374.16		**73**	3290.3	4397.06	2044.39	3313.76
24	3203.52	4301.9	2071.02	3396.27		**74**	3137.6	4388.14	2096.52	3396.33
25	3185.22	4352.88	2012.48	3336.97		**75**	3264.47	4399.38	2019.64	3389.03
26	3213.01	4396.02	2038.21	3355.06		**76**	3171.32	4308.51	2003.29	3335.08
27	3164.43	4380.8	2084.14	3356.01		**77**	3145.61	4373.29	2026.19	3408.3
28	3209.21	4362.13	2049.29	3380.93		**78**	3213.3	4333.68	2098.23	3335.23
29	3166.85	4348.84	2017.91	3345.74		**79**	3271.54	4333.67	2050.02	3362.61
30	3273.98	4391.16	2041.43	3314.93		**80**	3241.86	4373.05	2059.45	3373.15
31	3217.06	4373.21	2084.04	3348.64		**81**	3174.56	4378.98	2087.66	3392.76
32	3206.49	4348.94	2001.58	3368.14		**82**	3225.02	4305.92	2046.95	3335.37
33	3228.37	4375.8	2001.11	3387.21		**83**	3175.72	4301.45	2000.95	3346.47
34	3197.17	4356.02	2054.38	3378.54		**84**	3217.33	4376.31	2057.68	3360.8
35	3299.68	4382.28	2015.19	3404.56		**85**	3183.03	4384.63	2096.63	3409.09
36	3283.28	4316.33	2044.84	3393.77		**86**	3102.08	4390.88	2063.62	3341.79
37	3191.31	4393.45	2029.38	3332.3		**87**	3179.44	4302.66	2000.74	3328.2
38	3294.87	4319.25	2044.37	3365.58		**88**	3107.02	4301.29	2002.06	3318.7
39	3175.16	4393.09	2087.26	3315.31		**89**	3213.61	4341.71	2073.82	3405.56
40	3217.49	4321.53	2054.53	3399.89		**90**	3100.69	4371.59	2010.72	3400.55
41	3174.76	4313.22	2058.09	3395.13		**91**	3102.35	4393.92	2003.87	3342.81
42	3149.51	4309.05	2016.67	3371.75		**92**	3100.05	4332.58	2053.47	3359.58
43	3269.6	4343.75	2080.03	3349.21		**93**	3147.87	4396.62	2017.93	3379.13
44	3266.63	4376.54	2014.22	3313.54		**94**	3190.26	4342.08	2061.32	3322.11
45	3150.6	4315.69	2026.3	3389.22		**95**	3163.59	4334.76	2097.64	3331.38
46	3251.65	4310.74	2097.45	3403.15		**96**	3276.68	4347.33	2056.51	3366.61
47	3128.59	4365.72	2002.36	3372.71		**97**	3284.56	4317.05	2095.26	3361.7
48	3225.69	4399.7	2097.02	3326.47		**98**	3182.2	4357.43	2082.36	3339.74
49	3244.15	4369.98	2000.78	3333.82		**99**	3153.98	4364.96	2063.71	3386.72
50	3150.81	4337.07	2064.44	3357.09		**100**	3159.17	4315.69	2020.7	3370.2
Avg of 100 Attempts	**3202.561**	**4348.651**	**2049.025**	**3361.854**						

Case 1: Existing from Local and DmRT follows Local -> Shared -> Ideal

2. Fragmentation for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	54.71	42.68	28.49	14.22		**51**	52.46	42.72	27.44	16.22
2	51.41	45.77	31.39	14.43		**52**	51.84	43.2	31.57	14.28
3	50.61	42.81	27.14	16.74		**53**	53.29	43.41	29.75	14.94
4	52.43	45.21	29.89	16.46		**54**	50.28	46.23	27.31	13.16
5	54.96	44.88	30.1	17.01		**55**	53.52	42.61	28.76	15.22
6	53.52	46.66	31.97	15.85		**56**	54.79	46.62	28.35	14.73
7	52.86	45.27	31	14.54		**57**	52.04	46.51	28	16.33
8	50.77	42	28.21	16.37		**58**	54.27	44.25	30.94	17.86
9	52.29	46.27	31.57	13.52		**59**	53.33	44.38	28.73	16.34
10	51.53	45.93	29.39	13.74		**60**	52.85	44.84	27.01	15.73
11	52.06	42.4	30.08	17.3		**61**	52.62	46.16	29.32	13.54
12	54.03	42.43	28.82	14.28		**62**	52.28	45.59	31.71	17.69
13	52.35	46.97	29.08	14.59		**63**	51.2	42.26	28.1	16.76
14	50.73	42.51	30.12	17.08		**64**	53.73	46.8	30.86	16.67
15	52.83	45.11	29.68	15.73		**65**	53.7	43.65	28.54	16.59
16	50.12	42.17	29.08	15.01		**66**	50.98	45.34	30.46	16.87
17	51.4	46.12	29.07	15.49		**67**	52.16	44.69	30.53	13.81
18	54.57	43.36	31.74	16.65		**68**	53.28	44.27	30.75	13.33
19	52.24	43.92	30.98	17.7		**69**	52.86	44.6	27.91	14.03
20	53.76	43.62	31.73	13.64		**70**	51.01	42.89	30.04	13.23
21	51.91	43.45	29.31	17.62		**71**	52.61	44.05	30.99	16.57
22	54.11	44.48	27.92	14.15		**72**	54.9	46.09	27.54	15.34
23	54.7	45.51	27.95	17.3		**73**	51.52	44.23	27.27	17.7
24	51.96	43.57	30.49	17.77		**74**	52.6	42.3	28.92	13.95
25	53.29	46.06	31.55	14.77		**75**	52.6	45.48	27.67	14.3
26	52.22	42.99	29.01	16.47		**76**	50.67	46.06	31.38	16.89
27	52.77	46.35	30.87	16.24		**77**	52.63	45.5	29	14.42
28	52.54	45.77	29.67	17.72		**78**	53.77	46.05	30.13	15.47
29	50.92	42.88	31.28	14.92		**79**	54.17	44.9	30.27	15.72
30	52.17	46.6	28.53	15.35		**80**	53.44	46.64	28.69	17.93
31	53.63	43.59	31.51	14.89		**81**	50.21	44.96	32	15.24
32	52.81	42.62	31.09	17.42		**82**	52.09	43.57	27.39	17.97
33	51.42	44.5	28.01	15.68		**83**	54.26	42.42	27.69	13.48
34	51.49	44.84	31.97	16.86		**84**	50.63	43.77	29.42	17.13
35	50.78	46.54	30.01	13.01		**85**	50.77	46.73	30.36	14.73
36	53.52	44.03	30.39	13.03		**86**	53.19	42.76	27.37	13.91
37	52.5	43.41	31.64	14.01		**87**	54.72	44.95	27.89	17.78
38	54.55	46.4	27.24	15.89		**88**	54.91	43.43	28.83	15.56
39	50.46	42.29	28.95	16.3		**89**	54.51	43.24	31.28	13.48
40	51.77	43.68	28.1	15.75		**90**	52.72	44.48	30.69	13.19
41	51.4	45.44	27.91	17.36		**91**	51.45	42.98	30.55	15.66
42	51.62	43.29	30.78	17.59		**92**	53.45	44.51	30.79	13.52
43	53.91	46.04	30.77	15.95		**93**	51.16	45.57	30.63	15.04
44	53.44	46.82	31.79	17.93		**94**	50.09	42.13	28.27	17.19
45	50.3	45.89	27.89	16.94		**95**	53.7	45.84	29.38	14.24
46	53.24	42.37	30.15	16.99		**96**	54.5	46.15	29.42	17.35
47	50.18	42.03	29.9	13.2		**97**	54.07	43.42	27.06	16.71
48	52.51	46.38	29.59	15.9		**98**	51.06	43.87	31.98	13.35
49	53.64	46.81	28.61	15.25		**99**	54.49	42.89	28.94	16.15
50	51.56	43.84	27.06	16.99		**100**	51.03	44.89	31.32	15.51
Avg of 100 Attempts	**52.5491**	**44.4944**	**29.5867**	**15.6241**						

Case 1: Existing from Local and DmRT follows Local -> Shared -> Ideal

3. No. of Request Satisfied for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	36.45	51.05	58.00	83.40		**51**	34.95	48.50	59.80	82.15
2	35.30	48.15	61.90	83.65		**52**	35.85	50.50	60.95	82.65
3	36.75	48.10	58.75	81.10		**53**	37.00	49.20	59.20	83.80
4	35.85	48.65	61.05	82.00		**54**	33.95	51.45	62.00	83.40
5	33.85	50.00	59.30	82.90		**55**	33.35	51.05	61.40	82.50
6	36.00	51.15	59.50	82.60		**56**	35.25	49.45	59.15	81.75
7	36.15	49.75	59.85	83.80		**57**	36.30	48.20	59.85	82.60
8	36.35	51.45	61.15	82.45		**58**	36.45	51.00	61.65	81.20
9	36.20	48.65	59.10	82.00		**59**	35.95	49.10	58.10	81.50
10	35.00	48.10	58.25	82.90		**60**	36.05	50.50	61.75	83.70
11	34.00	49.80	61.80	84.00		**61**	36.60	49.60	60.50	81.55
12	36.80	51.30	60.85	81.50		**62**	35.65	50.45	61.35	82.80
13	35.65	48.05	59.90	82.35		**63**	36.95	49.90	60.85	83.75
14	33.30	51.20	60.90	81.75		**64**	35.35	49.35	59.05	81.45
15	34.20	48.35	60.80	82.30		**65**	34.85	49.55	58.80	81.80
16	33.35	48.05	61.40	81.20		**66**	36.20	49.05	59.45	81.25
17	34.90	49.75	58.95	83.20		**67**	34.30	49.60	60.30	81.20
18	33.55	51.70	60.60	82.30		**68**	34.95	51.65	60.75	82.50
19	34.85	48.15	60.55	83.10		**69**	34.60	48.95	58.00	83.30
20	34.60	51.15	60.75	83.75		**70**	36.15	49.60	60.75	81.95
21	33.50	51.20	59.40	83.10		**71**	34.60	51.05	60.15	81.30
22	35.25	48.20	60.75	81.70		**72**	36.55	50.75	59.15	82.55
23	36.90	50.30	60.75	83.75		**73**	35.35	49.30	59.20	82.25
24	34.65	50.50	58.50	81.45		**74**	33.40	50.00	58.40	83.40
25	34.10	50.40	59.05	81.25		**75**	34.15	50.80	61.80	82.80
26	33.20	50.20	58.70	83.50		**76**	34.00	51.75	60.15	82.65
27	36.20	51.70	60.30	83.20		**77**	33.85	50.05	59.70	81.55
28	34.90	52.00	59.95	82.65		**78**	33.85	50.70	60.45	82.00
29	34.20	48.50	61.80	83.00		**79**	34.05	50.45	58.45	81.70
30	36.00	49.30	58.50	81.85		**80**	34.20	51.95	59.00	81.40
31	33.50	51.75	58.50	81.00		**81**	36.50	48.25	58.85	81.40
32	33.80	48.35	60.55	81.85		**82**	35.30	49.20	61.55	81.45
33	35.35	48.10	59.80	81.05		**83**	35.45	51.50	61.45	83.25
34	33.75	51.75	58.10	81.90		**84**	35.70	50.65	59.50	82.20
35	35.15	48.00	60.00	82.70		**85**	36.40	50.80	60.15	82.90
36	33.50	48.80	60.80	81.55		**86**	33.00	50.90	59.00	81.05
37	35.15	50.60	61.05	81.60		**87**	33.60	51.60	61.30	81.75
38	35.05	48.00	58.10	83.15		**88**	33.30	50.00	60.75	82.15
39	34.00	50.75	58.85	81.85		**89**	34.45	49.60	59.70	81.85
40	36.70	49.45	61.10	82.75		**90**	34.55	50.40	61.70	82.20
41	35.10	51.30	59.40	81.60		**91**	33.25	50.75	60.45	82.70
42	36.50	48.05	59.65	83.65		**92**	35.45	48.10	60.10	82.05
43	34.15	51.70	61.30	83.50		**93**	33.55	48.80	60.05	83.90
44	35.25	52.00	60.00	81.05		**94**	33.55	48.50	59.00	82.75
45	34.95	50.90	60.55	82.40		**95**	36.00	49.30	62.00	81.15
46	34.90	51.10	61.20	83.85		**96**	34.65	50.65	59.95	81.95
47	36.75	50.00	59.45	83.30		**97**	35.35	51.85	59.60	83.40
48	34.85	50.50	61.15	82.10		**98**	36.05	50.65	59.95	83.90
49	34.70	51.60	59.00	83.35		**99**	35.25	48.40	60.00	81.35
50	33.10	51.00	61.70	81.40		**100**	36.35	51.25	59.05	82.70
Avg of 100 Attempts	**35.006**	**50.0315**	**60.055**	**82.3775**						

Case 2: Existing From Local and DmRT From Ideal

1. Execution Time for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	316.01	419.58	302.59	252.17		**51**	361.08	413.82	295.59	240.96
2	327.94	412.49	301.01	239.84		**52**	357.85	411.65	302.84	239.97
3	346.73	425.31	302.83	253.82		**53**	344.25	423.04	288.39	236.63
4	317.13	413.39	288.35	251.54		**54**	320.03	418.78	300.77	248.93
5	341.09	422.73	298.06	255.48		**55**	361.64	416.69	288.83	237.8
6	314.34	412.53	286.67	248.18		**56**	347.84	423.46	291.15	236.62
7	337.02	429.84	298.32	240.68		**57**	318.17	423.86	304.1	236.05
8	356.95	419.09	286.85	243.93		**58**	341.33	420.37	297.62	248.92
9	337.34	411.59	294.28	245.08		**59**	331.38	426.01	304.22	240.7
10	316.41	416.94	289.48	237.8		**60**	323.33	427.18	304.42	240.66
11	333.38	421.75	290.41	238.7		**61**	341.58	423.74	304.06	243.96
12	327.5	419.9	296.91	243.33		**62**	336.87	420.53	293.32	244.71
13	322.04	430.22	294.65	251.85		**63**	320.72	418.17	303.51	242.75
14	345.32	425.79	301.68	253.1		**64**	334.14	412.35	291.43	237.42
15	349.9	423.25	286.32	248.87		**65**	328.99	426.08	303.11	249.31
16	343.91	419.54	289.34	241.2		**66**	357.89	414	291.4	243.69
17	341.02	419.44	292.9	255.26		**67**	357.54	425.41	303.27	250.81
18	341.88	429.54	290.78	252.86		**68**	319.74	414.31	303.54	235.75
19	357.45	425.01	293.06	252.62		**69**	356.43	430.06	302.73	243.14
20	328.49	417.94	304.91	246.16		**70**	360.55	411.77	301.31	238.68
21	360.97	416.57	303.19	249.31		**71**	352.48	430.71	287.64	248.35
22	358.36	416.51	304.85	244.2		**72**	338.2	413.29	299.23	251.95
23	316.49	424.02	304.24	243.13		**73**	341.2	412.86	297.07	235.75
24	331.35	428.26	299.44	243.43		**74**	328.97	416.16	304.33	237.6
25	321.1	422.04	301.92	241.53		**75**	343.01	420.17	288.67	255.1
26	334.53	421.05	287.93	246.42		**76**	344.03	418.53	290.47	253.05
27	352.02	415.35	297.02	245.12		**77**	333.85	418.89	299.25	238.75
28	316.74	429.73	298.08	246.17		**78**	362.6	430.9	295.09	250.77
29	359.03	413.64	304.58	238.46		**79**	362.27	412.84	291.6	237.46
30	343.95	415.75	295.65	254.42		**80**	352.45	414.38	298.29	255.46
31	324.26	422.73	295.91	251.59		**81**	341.77	414.59	303.96	242.3
32	351.77	421.03	302.35	242.27		**82**	325.34	419.33	293.96	237.08
33	318.33	414.65	291.4	249.7		**83**	323.73	430.11	305.21	246.5
34	316.89	419.62	304.07	254.44		**84**	356.19	417.66	302.96	249.99
35	338.41	418.92	287.28	254.24		**85**	314.55	426.02	302.24	250.08
36	320.84	422.08	302.32	247.86		**86**	333.16	424.42	293.77	250.35
37	361.55	418.45	300.68	243.41		**87**	336.48	419.61	288.88	241.22
38	345.94	426.5	297.54	235.66		**88**	323.59	423.75	290.74	251.33
39	321.15	413.04	295.41	238.02		**89**	347.12	414.05	291.83	237.49
40	349.65	424.8	296.97	251.59		**90**	321.83	424.71	288.47	249.51
41	346.6	430.78	290.07	245.57		**91**	356.85	427.3	292.46	236.59
42	347.95	429.98	298.87	245.46		**92**	320.52	430.47	293.09	251.9
43	362	414.45	298.33	242.18		**93**	337.61	419.99	299.85	244.18
44	352.91	427.24	295.4	245.27		**94**	330.42	419.83	304.4	238.28
45	330.21	420.67	287.97	246.56		**95**	346.82	424.56	302.7	239.05
46	330.68	426.57	299.99	249.54		**96**	359.85	417.03	302.68	254.43
47	314.64	426.78	289.38	241.48		**97**	348.1	428.4	302.81	254.52
48	319.46	413.59	287.14	244.72		**98**	331.74	411.88	288.03	242.69
49	316.57	415.79	292.29	243.5		**99**	324.52	411.93	295.35	254.62
50	337.04	416.59	287.83	244.25		**100**	342.05	413.32	286.04	250.05
Avg of 100 Attempts	**338.0589**	**420.5202**	**296.4418**	**245.4583**						

Case 2: Existing From Local and DmRT From Ideal

2. Fragmentation for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	44.44	33.95	22.81	13.41		**51**	45.85	32.65	23.52	13.94
2	45.7	33.52	23.27	15.09		**52**	46.37	35.1	22.39	15.29
3	43.6	32.44	23.54	17.27		**53**	45.08	33.31	24.98	15.65
4	45.42	33.77	23.68	14.15		**54**	46.74	33.29	22.44	15.02
5	44.11	32.36	26.18	15.44		**55**	45.88	35.61	23.38	16.31
6	43.37	35.43	24.55	14.39		**56**	46.48	35.14	22.33	17.37
7	43.34	35.19	26.06	17.36		**57**	46.77	32.9	25.68	14.5
8	46.38	33.33	23.06	16.71		**58**	44.1	35.11	26.19	15.53
9	46.91	32.92	25.6	14.88		**59**	46.55	34.24	26.06	14.44
10	43.83	33.25	23.55	13.69		**60**	44.35	34.37	24.78	16.88
11	44.08	34.84	24.24	13.68		**61**	45.61	33.31	22.61	17.24
12	47.24	32.06	22.51	14.31		**62**	44.18	31.74	23.28	17.12
13	44.58	35.18	24.4	16.48		**63**	44.79	34.29	22.56	14.9
14	46.92	32.38	22.74	15.4		**64**	44.02	33.68	23.86	17.2
15	44.53	33.13	22.47	16.8		**65**	46.19	32.39	23.39	15.51
16	47.13	32.21	23.22	15.17		**66**	43.92	35.56	25.9	16.32
17	45.38	32.8	23.25	16.9		**67**	45.91	35.19	24.47	15.91
18	44.93	34.36	23.17	14.31		**68**	45.16	34.6	23.05	14.61
19	44.19	34.64	24.8	13.53		**69**	43.47	34.96	23.6	15.91
20	44.43	34.83	24.4	13.98		**70**	45.17	32.74	25.45	16.17
21	46.68	34.6	24.52	15.26		**71**	45.41	33.49	23.96	15.56
22	43.69	34.23	25.64	14.87		**72**	44.34	33.56	22.44	16.57
23	43.87	32.56	23.83	14.94		**73**	46.78	32.53	22.69	16.52
24	46.55	32.06	22.44	15.95		**74**	44.94	33.34	24.97	14.86
25	47.09	34.8	22.8	14.02		**75**	46.95	33.04	22.59	14.92
26	46.07	34.78	23.24	14.92		**76**	46.68	32.36	24.31	17.21
27	45.14	31.73	26.12	14.36		**77**	43.89	33	22.57	17.13
28	46.64	33.99	23.06	17.21		**78**	43.6	32.87	23.76	14.26
29	43.61	33.95	22.88	13.68		**79**	46.22	32.92	25.06	13.92
30	44.07	33.01	24.47	17.18		**80**	44.66	33.03	25.43	15.48
31	46.39	32.85	22.85	14.95		**81**	47.27	35.65	25.85	16.16
32	44.45	31.98	23.8	15.27		**82**	46.27	35.21	23.77	15.86
33	44.13	34	24.65	14.75		**83**	45.06	33.88	26.1	16.66
34	46.58	34.29	22.46	14.35		**84**	43.38	34.41	25.27	16.17
35	46.14	35.59	26.05	14.21		**85**	46.97	33.83	22.78	13.84
36	43.69	34.8	22.64	16.76		**86**	45.24	31.83	22.34	15.44
37	46.77	35.57	22.23	13.96		**87**	47.3	34.51	22.71	15.96
38	45.46	35.67	24.03	17.13		**88**	47.11	31.97	23.08	15.45
39	45.95	33.08	25.96	13.58		**89**	43.74	32.12	23.64	16.97
40	45.76	32.24	24.19	15.19		**90**	46.42	33.82	23.53	15.65
41	47.27	33.59	25.6	15.02		**91**	44.27	33.12	23.48	16.19
42	47.02	32.17	25.92	15.67		**92**	44.06	33.28	24.84	16.93
43	45.76	32.52	24.02	17.03		**93**	43.41	32.67	22.69	16.29
44	44.27	34.25	24.07	15.17		**94**	43.5	33.99	26	15.95
45	46.73	32.17	23.5	16.05		**95**	45.85	34.58	26.07	15.08
46	43.59	33.11	23.23	14.58		**96**	43.61	32.57	25.07	15.39
47	43.45	33.87	25.16	14.78		**97**	45.24	35.62	24.69	16.34
48	45.43	33.06	25.36	14.19		**98**	47.02	33.1	23.79	16.28
49	46.9	33.38	25.82	14.51		**99**	44.87	35.39	23.2	15.72
50	43.43	33.2	22.62	14.96		**100**	43.89	31.7	25.11	14.94
Avg of 100 Attempts	**45.2763**	**33.6326**	**24.0237**	**15.4697**						

Case 2: Existing From Local and DmRT From Ideal

3. No. of Request Satisfied for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	56	65	78	88		**51**	59	62	76	92
2	56	66	77	92		**52**	57	66	76	89
3	56	66	77	92		**53**	59	63	78	91
4	56	65	75	90		**54**	58	62	77	90
5	60	67	75	88		**55**	57	65	78	91
6	59	66	79	89		**56**	58	64	76	92
7	59	65	76	88		**57**	58	65	77	89
8	56	66	77	91		**58**	59	65	77	90
9	59	63	76	90		**59**	57	63	77	89
10	57	63	77	90		**60**	56	66	79	90
11	59	64	78	88		**61**	59	67	77	91
12	60	64	76	91		**62**	56	64	76	89
13	58	63	75	90		**63**	59	64	76	88
14	56	64	78	92		**64**	58	64	79	91
15	57	63	76	88		**65**	57	63	75	90
16	59	65	78	92		**66**	59	64	77	91
17	57	64	78	88		**67**	58	62	75	88
18	59	64	77	88		**68**	56	63	76	88
19	56	62	76	88		**69**	56	64	79	90
20	59	64	79	90		**70**	60	62	75	88
21	58	63	75	89		**71**	56	66	75	91
22	57	62	76	90		**72**	56	65	77	90
23	58	64	76	88		**73**	58	63	75	89
24	59	64	75	91		**74**	57	62	75	92
25	58	63	75	91		**75**	57	62	75	89
26	60	66	76	90		**76**	56	66	78	89
27	57	62	76	89		**77**	59	66	78	89
28	59	63	78	91		**78**	59	66	79	90
29	56	66	76	92		**79**	58	65	78	92
30	58	65	76	88		**80**	60	65	78	91
31	58	64	76	90		**81**	56	64	76	92
32	56	66	79	89		**82**	57	66	78	92
33	59	66	77	92		**83**	58	66	76	89
34	57	65	79	91		**84**	58	66	75	91
35	59	64	79	90		**85**	57	64	78	91
36	57	63	78	92		**86**	59	63	78	89
37	56	66	78	92		**87**	59	65	77	90
38	57	66	76	88		**88**	56	66	77	91
39	58	63	78	91		**89**	59	64	77	89
40	59	64	75	92		**90**	58	63	78	92
41	58	66	76	89		**91**	56	65	77	92
42	59	62	77	88		**92**	57	62	79	91
43	60	63	77	90		**93**	59	62	79	90
44	59	65	78	89		**94**	58	62	76	92
45	58	63	78	89		**95**	58	63	77	89
46	59	66	78	89		**96**	56	65	79	88
47	58	66	78	90		**97**	60	66	78	89
48	58	62	76	92		**98**	59	65	77	88
49	57	65	77	89		**99**	58	64	77	92
50	58	63	76	92		**100**	57	64	76	88
Avg of 100 Attempts	**57.7885**	**64.1904**	**76.9458**	**89.9526**						

Case 2: Existing From Local and DmRT From Ideal

1. Execution Time for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	2034.08	2837.04	1573.89	1292.72		**51**	2080.88	2945.05	1710.57	1116.58
2	2105.76	2960.93	1717.19	1019.08		**52**	2144.96	2784.9	1335.15	1293.72
3	2099.59	2907.37	1570.82	944.65		**53**	1994.37	3011.75	1414.54	1018.58
4	1975.47	3067.58	1616.78	1220.68		**54**	2028.7	3084.37	1710.51	1175.21
5	2151.33	2830.95	1473.22	1218.42		**55**	2110.09	3063.71	1518.31	1311
6	2095.93	3026.8	1341.88	1217.83		**56**	2016.76	2931.64	1717.01	979.79
7	2003.43	3168.71	1476.89	1023.53		**57**	2115.31	3012.86	1575.08	1286.22
8	2063.06	2792.16	1669.89	984.77		**58**	2033.97	2880.24	1570.91	1088.13
9	2046.45	3118.78	1346.07	1133.86		**59**	1989.38	3069.45	1511.59	1137.3
10	2068.58	2876.06	1399	1131.03		**60**	2145.94	2821.1	1549.85	1055.12
11	2111.69	3095.37	1378.8	1307.66		**61**	1978.49	3125.85	1687.72	1147.08
12	1981.19	2928.21	1676.52	1009.62		**62**	2117.38	2928.51	1381.35	963.19
13	2016.35	2810.95	1443.73	1270.75		**63**	1962.53	2832.29	1380.94	981.57
14	1962.69	3124.45	1561.12	932.85		**64**	2087.86	3048.67	1553.55	1054.98
15	2140.44	2975.11	1628.4	1040.23		**65**	2124.81	3131.21	1647.75	1179.78
16	1974.22	2850.11	1634.04	1281.55		**66**	2039.88	3168.31	1427.44	1075.71
17	1974.94	2918.26	1400.26	1092		**67**	2004.77	3104.71	1604.65	1129.58
18	2141.16	3019.76	1598.72	1146.27		**68**	1959.29	3105.97	1362.32	1080.79
19	2019.55	3065.15	1645.87	1221.45		**69**	2012.43	2822.78	1446.81	958.38
20	2101.17	3136.52	1459.03	921.8		**70**	1968.55	2968.75	1711.03	1070.36
21	2126.75	2925.68	1348.72	941.28		**71**	1994.35	2843.26	1431.77	1255.67
22	2126.93	3054.08	1494.78	1241.01		**72**	2120.16	3139.82	1496.54	1269.34
23	2151.17	3059.37	1560.72	1151.31		**73**	2083.44	3168.16	1672.22	1148.07
24	2011.95	3092.95	1384.99	1221.1		**74**	2021.52	3162.71	1707.96	1241.55
25	2119.87	2909.96	1669.09	1005.01		**75**	2149.62	3154.46	1361.51	1268.27
26	2119.66	3037.73	1448.63	958.24		**76**	2029.81	3041.18	1613.83	1271.86
27	1992.36	3156.59	1475.35	1155.52		**77**	2065.88	2852.38	1723.45	1008.51
28	2002.64	2858.47	1721.33	1084.5		**78**	2097.15	3000.02	1661.96	1029.79
29	1993.21	2985.28	1458.94	1057.57		**79**	2097.33	3158.28	1724.88	1196.29
30	2100.37	3119.48	1479.05	1026.35		**80**	2067.71	2979.32	1432.38	954.64
31	2074.15	3149.84	1606.93	1203.27		**81**	2067.99	3171.22	1701.01	1234.87
32	2063.4	2798.69	1455.65	1124.49		**82**	2036.37	2930.94	1387.09	1206.76
33	2104.05	3175.02	1657.86	998.01		**83**	2007.44	2795.91	1547.73	1146.34
34	1998.83	3119.09	1670.84	1103.87		**84**	2035.73	3163.22	1660.44	1261.7
35	1982.23	2884.99	1347.4	1216.7		**85**	2044.05	3077.15	1472.86	1085.21
36	1963.64	2928.73	1422.8	1040.11		**86**	2026.12	3023.76	1597.1	1193.02
37	2058.92	3164.37	1563.68	1284.92		**87**	2106.48	2924.72	1431.26	923.04
38	2102.61	2934.47	1409.61	999.14		**88**	2101.03	2910.67	1699.59	1268.15
39	2136.2	2942.26	1501.04	1152.65		**89**	2067.8	3052.8	1480.47	1236.96
40	2140.73	2986.9	1720.66	1041.49		**90**	2118.26	3028.26	1390.62	982.96
41	2018.03	3074.08	1633.94	928.5		**91**	2021.93	2959.66	1484.45	1028.94
42	2150.17	3097.53	1349.39	1154.97		**92**	1954.33	3090.69	1588.2	929.57
43	2091.89	2832.32	1536.98	1275.17		**93**	2017.28	2914.4	1653.52	925.69
44	2018.9	3169.29	1681.03	1154.68		**94**	1978.9	2882.22	1396.55	1259.08
45	2012.57	2788.23	1585.94	1087.98		**95**	2000.17	3064.21	1356.58	1019.42
46	2034.94	3066.52	1331.62	1142.73		**96**	2148.29	3015.91	1428.2	1099.98
47	2113.04	3089.18	1725.82	1221.59		**97**	1969.21	2802.11	1711.75	1005.09
48	2009.41	2911.7	1684.99	1101.12		**98**	2049.41	2876.28	1728.56	1011.43
49	2088.14	2795.25	1465.94	1101.08		**99**	2027.19	2992.55	1662.52	992.27
50	1996.83	3020.4	1609.9	1137.43		**100**	2079.62	2867.01	1358.53	1303.43
Avg of 100 Attempts	**2054.716**	**2995.241**	**1539.964**	**1115.835**						

Case 2: Existing From Local and DmRT From Ideal

2. Fragmentation for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	52.44	46.62	28.9	20.19		**51**	52.69	43.34	31.24	20.69
2	54.45	43.28	29.18	18.52		**52**	53.9	45.49	29.31	19.83
3	53.39	45.51	31.63	21.15		**53**	53.07	46.4	30.55	19.2
4	55.5	45.36	30.69	17.57		**54**	53.86	44.61	30.47	19.68
5	54.03	45.81	31.35	19.11		**55**	54.35	46.75	31.97	20.65
6	53.91	46.23	30.23	19.96		**56**	53.19	44.07	29.35	19.94
7	53.92	42.85	29.14	20.47		**57**	52.32	44.67	28.6	18.83
8	55.36	45.74	28.28	18.35		**58**	55.74	43.77	30.12	21.01
9	54.81	44.34	29.82	17.69		**59**	51.57	45.7	29.79	19.63
10	54.12	44.04	31.3	20.07		**60**	55.48	43.43	29.31	20.79
11	53.95	45.95	30.82	20.31		**61**	51.76	45.7	30.86	18.01
12	53.27	45.07	31.76	19.79		**62**	54.3	43.92	31.05	18.92
13	52.83	44.25	30.71	20.23		**63**	51.67	46.3	30.74	21.34
14	52.99	44.47	30.85	18.61		**64**	54.71	46.59	29.86	17.84
15	54.69	43.07	31.93	18.48		**65**	55	46.45	28.83	19
16	53.69	43.35	29.84	21.37		**66**	54.26	45.28	31.92	20.67
17	53.25	46.36	31.89	20.77		**67**	53.08	43.72	28.9	20.45
18	52.36	43.73	29.45	18.16		**68**	52.8	43.34	30.99	17.61
19	54.23	46	28.55	19.73		**69**	53.43	44.13	32.19	20.44
20	53.35	44.78	30.49	17.86		**70**	52.85	44.33	28.41	19.7
21	54.96	45.19	30.07	20.89		**71**	55.68	45.62	31.37	20.88
22	53.74	46.51	28.5	20.45		**72**	52.71	43.53	31.89	18.82
23	55.49	46.51	28.55	18.72		**73**	53.87	46.07	29.56	18.07
24	51.84	46.12	31.5	20.39		**74**	51.94	46.33	32	18.78
25	53.67	42.84	28.97	20.89		**75**	53.76	44.42	29.08	20.69
26	55.18	43.54	30.92	20.53		**76**	53.73	46.78	29.66	18.42
27	53.45	45.01	31.34	19.83		**77**	52.11	46.28	30.42	20.22
28	55.21	46.56	30.2	20.32		**78**	53.85	44.08	29.37	20.12
29	56.46	43.84	30.07	21.11		**79**	52.03	46.39	30.6	21.08
30	53.1	43.28	31.38	19.86		**80**	53.77	43.14	29.54	18.02
31	52.42	43.72	30.08	21.04		**81**	54.26	45.68	31.18	19.49
32	54.74	45.67	28.54	19.73		**82**	56.07	43.11	29.17	20.37
33	51.7	45.5	31.09	18.13		**83**	56.4	42.98	30.85	20.25
34	55.79	46.68	32.05	18.99		**84**	56.31	45.15	31.7	20.09
35	54.72	45.5	29.79	18.52		**85**	54.72	44.35	30.17	19.89
36	55.7	45.73	31.18	18.68		**86**	53.48	45.35	32.05	17.84
37	51.75	44.59	29.66	20.95		**87**	51.93	46.5	29.8	20.04
38	52.93	46.27	28.37	17.55		**88**	53.29	45.83	31.08	21.03
39	55.21	42.88	28.87	18.43		**89**	53.6	45.6	30.05	17.79
40	53.2	43.17	30.42	18.14		**90**	53.85	46.68	32.12	17.67
41	54.02	44.03	30.7	17.89		**91**	53.16	42.87	30.27	20.68
42	52.25	43.21	30.84	18.3		**92**	54.21	44.83	30.84	20.22
43	51.89	44.13	29.35	17.73		**93**	55.4	46.07	30.85	19.81
44	51.95	46.38	29.95	20.18		**94**	56.43	46.09	30.85	20.42
45	52.71	43.05	31.41	19.01		**95**	52.2	45.29	31.02	19.27
46	54.63	44.26	29.98	18.36		**96**	54.13	46.12	30.73	19.27
47	56.31	46.39	31.66	18.62		**97**	52.08	46.16	29.15	20.35
48	53.55	45.42	28.41	19.5		**98**	54.23	44.84	28.23	21.19
49	55.86	43.1	30.75	20.99		**99**	51.99	43.25	29.22	18.56
50	52.97	44.47	29.75	18.6		**100**	56.37	42.93	31.11	17.98
Avg of 100 Attempts	**53.8153**	**44.9067**	**30.2955**	**19.5226**						

Case 2: Existing From Local and DmRT From Ideal

3. No. of Request Satisfied for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	47.2	54.8	67.5	82.0		**51**	45.4	52.4	64.8	81.7
2	46.5	53.4	66.2	83.1		**52**	46.1	54.7	65.6	82.3
3	45.0	54.8	64.5	83.2		**53**	47.2	56.6	66.7	81.8
4	48.3	54.7	65.6	84.7		**54**	47.0	52.6	64.5	84.5
5	45.8	53.8	64.3	81.9		**55**	46.3	52.5	66.0	82.3
6	44.9	55.9	66.5	82.0		**56**	46.4	55.8	64.3	84.1
7	44.7	53.0	64.5	82.9		**57**	46.5	55.3	65.0	84.0
8	45.9	55.8	64.7	82.4		**58**	48.4	56.0	67.0	83.1
9	45.3	54.0	66.4	82.7		**59**	45.8	57.0	64.0	81.2
10	47.0	54.6	67.4	82.5		**60**	45.3	54.6	67.0	83.5
11	45.7	57.0	66.2	84.3		**61**	47.6	52.5	67.2	82.4
12	48.3	52.3	65.1	84.8		**62**	44.7	56.4	67.5	80.9
13	46.9	55.4	67.1	83.4		**63**	46.9	54.4	66.2	84.2
14	45.2	53.4	65.9	81.0		**64**	45.8	55.0	66.0	83.8
15	46.9	54.2	65.4	82.5		**65**	45.5	52.9	66.9	81.8
16	46.8	52.6	65.1	83.5		**66**	44.7	54.8	64.2	82.2
17	46.7	56.5	64.9	81.9		**67**	44.7	55.4	66.3	82.7
18	48.4	55.0	68.0	81.1		**68**	46.1	54.3	65.2	84.4
19	45.8	53.5	65.5	83.6		**69**	47.2	55.1	67.9	84.0
20	46.1	54.6	65.0	83.8		**70**	46.5	54.7	67.7	83.3
21	46.3	55.6	66.4	81.1		**71**	45.9	56.1	68.0	83.7
22	45.0	54.6	66.5	81.0		**72**	46.3	55.1	67.1	81.2
23	45.5	55.7	65.4	83.0		**73**	47.0	54.2	65.8	82.4
24	46.8	55.8	64.3	81.0		**74**	47.6	54.4	66.3	83.2
25	45.9	55.2	67.0	84.8		**75**	47.9	54.4	66.1	81.1
26	46.4	52.3	66.5	80.9		**76**	46.4	52.3	64.3	82.1
27	47.6	54.4	66.4	81.0		**77**	48.6	53.0	64.7	82.9
28	44.9	53.3	66.3	83.7		**78**	44.7	53.8	66.8	81.0
29	45.1	53.0	65.8	83.0		**79**	46.4	52.9	64.6	83.7
30	46.9	54.7	64.6	82.3		**80**	47.3	56.9	64.6	81.9
31	47.1	53.5	66.1	84.0		**81**	48.5	52.9	66.0	83.0
32	47.9	53.9	65.3	83.0		**82**	47.4	53.5	67.1	81.4
33	48.2	55.7	67.6	82.7		**83**	48.3	57.1	65.7	84.3
34	47.6	54.1	65.3	82.6		**84**	48.6	52.4	67.7	84.1
35	46.2	54.9	67.3	81.4		**85**	47.4	56.0	66.1	82.1
36	45.5	53.6	66.9	83.2		**86**	45.6	54.0	67.5	84.7
37	47.4	55.2	64.1	83.4		**87**	44.9	54.3	64.3	82.7
38	45.6	55.0	64.5	83.9		**88**	47.3	52.9	65.9	82.4
39	47.0	52.8	67.6	83.3		**89**	45.5	54.1	65.0	83.8
40	45.6	55.8	66.2	81.2		**90**	46.3	57.0	67.9	84.0
41	44.6	53.1	67.5	84.1		**91**	47.0	56.6	64.0	81.4
42	46.9	54.5	66.9	83.2		**92**	47.4	52.6	65.4	84.1
43	45.5	56.2	64.1	83.3		**93**	45.8	56.2	65.5	83.2
44	48.0	52.8	64.2	84.2		**94**	45.2	53.6	65.9	80.9
45	47.0	56.0	66.8	84.8		**95**	47.9	54.7	65.7	83.1
46	44.8	56.0	66.5	81.6		**96**	47.9	55.1	66.3	84.6
47	46.1	56.0	66.6	83.3		**97**	47.9	52.3	65.7	82.7
48	45.1	55.0	66.3	83.6		**98**	45.8	52.5	64.3	84.1
49	45.1	56.3	64.9	83.1		**99**	45.1	56.6	65.9	81.3
50	44.7	56.4	64.9	84.8		**100**	47.6	53.9	67.4	83.4
Avg of 100 Attempts	**46.4232**	**54.5453**	**65.9168**	**82.8598**						

Case 2: Existing From Local and DmRT From Ideal

1. Execution Time for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	3334.88	4296.69	2008.71	1826.66		**51**	3310.18	4281.08	2072.03	1781.93
2	3348.1	4273.13	2029.38	1917.11		**52**	3209.48	4289.26	2077.28	1915.79
3	3307.65	4309.9	2018.03	1755.12		**53**	3260.51	4314.82	2020.61	1953.94
4	3383.96	4262.54	2026.81	1706.17		**54**	3331.42	4276.39	2015.45	1739.1
5	3314.6	4319.76	2076.18	1858.65		**55**	3343.4	4331.95	2061.44	1768.22
6	3284.38	4328.03	2105.99	1771.25		**56**	3357.65	4282.77	2052.94	1847.38
7	3289.78	4323.03	2077.37	1675.4		**57**	3245	4275.56	2066.89	1661.94
8	3326.89	4337.93	2025.94	1659.82		**58**	3223.57	4296.32	2093.87	1678.93
9	3205.94	4297.77	2063.45	1719.04		**59**	3274.74	4312.93	2035.96	1748.5
10	3353.39	4280.25	2059.41	1888.6		**60**	3382.55	4322.3	2061.9	1945.1
11	3256.28	4311.83	2054.98	1706.01		**61**	3350.71	4296.8	2055.37	1812.41
12	3260.63	4284.13	2009.68	1706.16		**62**	3214.63	4328.3	2100.79	1665.6
13	3337.12	4320.12	2062.12	1760.85		**63**	3256	4267.95	2059.74	1928.89
14	3190.15	4252.14	2095.16	1721.36		**64**	3315.17	4312.11	2098.56	1822.93
15	3340.94	4258.98	2040.21	1712.57		**65**	3233.54	4330.77	2094.88	1727.87
16	3275.13	4266.2	2065.19	1701.57		**66**	3330.71	4263.56	2067.94	1687.79
17	3324.01	4283.26	2091.38	1739.77		**67**	3249.19	4311.37	2045.96	1683.7
18	3230.45	4259.64	2082.61	1655.9		**68**	3308.66	4294.35	2057.06	1668.52
19	3331.41	4333	2059.76	1952.93		**69**	3262.94	4276.54	2043.27	1826.65
20	3266.37	4271.27	2100.77	1951.09		**70**	3367.27	4246.39	2073.04	1901.62
21	3243.54	4311.16	2086.13	1655.23		**71**	3221.8	4247.48	2051.05	1844.74
22	3273.69	4276.76	2096.7	1720.16		**72**	3243.25	4255.59	2081.95	1805.63
23	3364.47	4285.09	2083.52	1804.29		**73**	3322.89	4265.51	2010.97	1855.94
24	3340.9	4280.4	2061.45	1745.74		**74**	3325.66	4319.97	2098.67	1689.72
25	3231.14	4294.25	2078.33	1657.31		**75**	3241.49	4320.19	2085.02	1691.5
26	3273.66	4242.08	2078.27	1948.1		**76**	3229.17	4254.22	2106.88	1828.78
27	3309.59	4321.14	2075.96	1749.35		**77**	3191.53	4262.42	2052.48	1890.44
28	3271.25	4285.48	2027.85	1745.51		**78**	3191.28	4324.09	2085.2	1879.6
29	3226.91	4331.14	2100.41	1688.59		**79**	3374.83	4318.02	2045.19	1770.55
30	3217	4285.35	2068.58	1817.2		**80**	3221.21	4280.75	2085.17	1699.04
31	3320.22	4272.02	2038.19	1740.48		**81**	3282.12	4247.46	2088.97	1790.58
32	3307.23	4263.67	2069.58	1850.82		**82**	3223.9	4261.56	2095.62	1891.09
33	3245.14	4315.9	2049	1918.77		**83**	3270.89	4248.93	2048.2	1897.51
34	3297.91	4338.06	2045.15	1667.92		**84**	3335.04	4300.19	2106.6	1767.92
35	3252.17	4247.66	2070.43	1655.61		**85**	3234.68	4298.45	2040.19	1724.9
36	3300.06	4325.43	2087.86	1770.21		**86**	3354.63	4297.44	2104.59	1747.78
37	3350.01	4245.1	2009.42	1925.88		**87**	3263.63	4250.87	2062.68	1885.1
38	3293.44	4326.69	2102.62	1683.16		**88**	3381.03	4314.19	2077.3	1937.87
39	3239.81	4262.67	2028.99	1762.36		**89**	3376.4	4280.06	2062.93	1946.44
40	3214.53	4246.55	2059.93	1863.37		**90**	3382.58	4291.31	2097.16	1716.68
41	3294.65	4259.34	2044.55	1865.75		**91**	3239.64	4252.75	2037.23	1732.81
42	3347.8	4335.89	2087.53	1693.73		**92**	3259.56	4261.93	2016.31	1695.89
43	3218.3	4261.06	2100.75	1751.66		**93**	3253.2	4264.48	2081.78	1767.43
44	3223.14	4302.14	2094.83	1674.66		**94**	3278.2	4244.8	2057.36	1933.36
45	3190.76	4293.32	2044.21	1766.8		**95**	3278.35	4334.1	2091.1	1829.44
46	3246.1	4281.31	2045.42	1731.6		**96**	3226.84	4252.4	2056.63	1825.72
47	3262.43	4297.99	2070.91	1758.56		**97**	3378.53	4297.4	2067.71	1741.57
48	3188.24	4326.73	2037.54	1909.39		**98**	3357.56	4260.6	2097.38	1949.69
49	3280.71	4266.85	2031	1660.87		**99**	3241.93	4310.85	2008.09	1786.87
50	3225.62	4266.48	2085.99	1819.36		**100**	3283.12	4269.02	2100.79	1887.77
Avg of 100 Attempts	**3283.05**	**4288.16**	**2064.70**	**1785.68**						

Case 2: Existing From Local and DmRT From Ideal

2. Fragmentation for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	60.34	54.97	36.12	26.38		**51**	60.89	51.91	37.13	24.62
2	61.51	52.37	38.45	28.31		**52**	63.08	55.1	35.37	25.34
3	61.84	53.84	35.23	23.81		**53**	62.83	52.73	39.24	24.47
4	62.29	55.08	39.75	26.14		**54**	60.4	51.85	37.9	27.13
5	59.86	54.11	39	23.6		**55**	62.46	55.12	37.17	28.29
6	60.56	54.82	37.87	25.25		**56**	60	53.54	39.37	24.98
7	62.71	52.88	36.6	23.93		**57**	60.2	51.93	36.63	24.37
8	63.1	54.87	36.86	26.93		**58**	61.27	54.52	35.96	27.26
9	60.08	54.44	38.88	24.52		**59**	61.12	53.52	38.27	28.01
10	60.55	51.9	37.26	23.5		**60**	61.31	53.21	36.69	24.43
11	63.07	53.07	37.06	25.25		**61**	62.89	52.02	37.22	24.15
12	61.62	54.68	37.89	23.71		**62**	63.06	52.7	39.94	24.46
13	60.03	52.02	36.76	26.43		**63**	62.03	53.86	35.56	27
14	60.11	53.11	37.7	25.76		**64**	61.74	53.08	36.35	25.02
15	61.32	52.91	37.76	25.65		**65**	61.17	53.05	36.61	28.24
16	60.39	54.17	37.58	23.54		**66**	62.28	53.01	37.46	26.56
17	59.18	53.47	36.74	27.99		**67**	62.21	53.69	36.66	25.37
18	62.66	52.11	39.07	28.09		**68**	60.39	51.27	39.73	28.29
19	60.3	54.09	36.42	24.02		**69**	62.01	51.49	37.06	27.23
20	61.12	53.83	37.89	26.19		**70**	59.19	51.76	39.03	28.35
21	62.64	54.72	38.45	26.49		**71**	60.91	51.64	38.36	27.87
22	62.79	51.29	39.41	27.82		**72**	60.81	53.76	40.18	26.46
23	62.81	52.15	35.81	25.15		**73**	59.44	53.33	35.89	28.13
24	61.94	52.11	40.01	24.44		**74**	59.18	51.31	36.17	23.46
25	62.11	53.77	35.68	26.26		**75**	62.22	51.76	39.38	28.28
26	60.29	51.51	37.53	25.14		**76**	59.29	54.21	39.79	27.86
27	59.33	51.27	39.64	28.21		**77**	61.36	55.06	37.58	25.94
28	60.46	51.71	36.59	27.2		**78**	62.16	54.64	37.66	27.38
29	61.91	52.07	35.33	28.16		**79**	60.13	54.51	38.77	23.67
30	62.74	54.63	36.78	28.28		**80**	59.81	51.99	36.17	28.34
31	60.96	53.21	39.95	23.5		**81**	59.56	54.02	39.39	26.23
32	60.98	51.82	35.39	27.42		**82**	59.85	53.4	37.24	26.63
33	62.59	52.98	39.83	24.91		**83**	60.91	52.86	38.54	25.18
34	62.92	53.83	40.04	25.28		**84**	60	53.8	39.16	26.87
35	61.96	52.65	38.85	24.98		**85**	61.44	53.55	36.83	23.54
36	62.96	52.13	39.79	25.86		**86**	61.69	52.91	35.64	27.47
37	62.57	51.53	38.97	24.74		**87**	60.47	51.88	39.9	25.74
38	60.86	54.53	37.32	24.9		**88**	59.34	53.72	36.47	26.75
39	60.7	53.05	35.87	23.79		**89**	61.37	52.22	39.7	26.25
40	60.88	54.18	35.47	24.34		**90**	60.45	54.66	36.94	27.6
41	60.82	51.28	39.89	27.14		**91**	61.29	52.77	35.92	24.72
42	60.52	55.08	36.5	26.58		**92**	61.46	53.3	38.63	25.26
43	60.73	53.74	39.22	28.04		**93**	59.69	52.51	39.92	27.76
44	59.72	52.84	40.17	27.2		**94**	59.44	52.13	38.54	25.87
45	62.73	54.17	37.33	27.08		**95**	61.65	54.77	36.96	27.2
46	60.8	53.32	36.65	25.71		**96**	59.76	52.45	39.92	26.9
47	60.66	51.75	37.5	26.83		**97**	62.37	51.55	38.47	28.39
48	61.73	52.44	35.4	24.81		**98**	59.38	51.64	37.55	25.59
49	60.27	52.44	36.31	23.48		**99**	60.03	53.26	40.03	25.84
50	59.66	51.8	35.58	25.5		**100**	59.42	51.48	38.79	27.16
Avg of 100 Attempts	**61.1009**	**53.0719**	**37.7599**	**26.0615**						

Case 2: Existing From Local and DmRT From Ideal

3. No. of Request Satisfied for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	36.25	48.85	58.30	75.70		**51**	36.65	48.75	59.50	73.45
2	34.80	51.55	61.00	73.70		**52**	36.85	51.85	59.65	74.00
3	34.35	48.85	60.85	75.75		**53**	34.15	51.45	58.15	73.15
4	33.95	48.75	59.00	72.25		**54**	34.85	50.70	61.65	72.75
5	33.10	50.05	60.90	75.50		**55**	34.20	51.70	58.20	75.85
6	35.15	49.15	59.40	73.00		**56**	36.45	48.10	59.10	72.50
7	35.45	50.45	58.50	75.85		**57**	34.45	50.75	58.85	75.00
8	34.85	48.90	60.90	72.10		**58**	34.00	51.00	59.80	75.05
9	35.60	50.30	61.90	74.00		**59**	36.05	48.25	61.60	72.75
10	33.00	50.90	60.00	74.30		**60**	34.15	48.40	58.55	74.50
11	34.75	48.20	61.35	75.65		**61**	33.10	51.65	61.50	75.05
12	35.00	51.10	61.30	73.55		**62**	36.30	49.00	58.20	73.25
13	36.20	50.75	61.60	72.10		**63**	34.40	48.90	59.00	72.05
14	33.55	48.25	60.80	73.95		**64**	36.95	51.45	61.20	75.00
15	33.05	51.45	60.30	73.55		**65**	34.65	50.05	58.05	75.30
16	34.75	48.90	60.90	74.30		**66**	34.60	51.80	59.90	74.60
17	36.80	49.10	60.00	73.15		**67**	33.55	49.70	58.65	75.25
18	35.60	50.50	58.75	72.15		**68**	34.25	48.00	59.00	75.90
19	33.20	48.55	61.25	75.85		**69**	36.75	50.75	60.15	72.00
20	35.35	51.05	60.15	74.35		**70**	35.85	49.40	58.05	74.05
21	33.50	49.55	61.30	72.25		**71**	36.95	49.80	60.95	74.15
22	36.05	51.30	59.00	74.30		**72**	36.70	48.30	60.45	74.70
23	37.00	50.40	61.80	74.85		**73**	34.45	48.70	60.90	75.15
24	36.45	50.80	58.60	74.60		**74**	34.30	49.10	58.15	75.95
25	33.35	49.30	58.00	74.00		**75**	35.15	49.25	59.45	74.45
26	34.50	48.45	59.95	74.20		**76**	35.00	51.70	59.10	72.60
27	36.25	49.20	60.50	72.60		**77**	35.55	51.65	60.35	74.05
28	34.05	50.55	59.70	74.25		**78**	34.05	50.85	59.70	72.80
29	34.60	50.75	59.00	75.40		**79**	34.65	51.75	61.50	75.45
30	33.55	50.55	60.30	75.45		**80**	36.55	51.75	58.60	73.00
31	36.10	49.30	61.40	73.35		**81**	33.20	48.05	59.90	73.80
32	35.45	49.15	59.20	73.10		**82**	35.15	48.00	59.35	75.00
33	34.05	48.10	58.25	74.50		**83**	34.00	51.55	59.00	75.80
34	36.35	50.25	60.40	72.40		**84**	33.50	51.40	59.85	73.70
35	35.40	51.05	61.95	72.05		**85**	33.20	49.65	61.70	72.40
36	34.95	50.55	58.80	75.90		**86**	35.65	48.20	59.40	75.65
37	33.40	49.50	60.75	73.20		**87**	33.25	49.05	60.05	73.75
38	33.55	51.75	59.70	74.60		**88**	34.65	50.50	60.55	74.05
39	35.85	51.50	58.55	73.05		**89**	34.10	49.00	58.65	72.25
40	36.30	49.70	61.30	74.35		**90**	33.75	49.40	59.10	74.65
41	33.70	49.10	58.75	75.65		**91**	36.00	49.30	60.25	72.55
42	34.55	49.00	61.95	75.15		**92**	36.40	49.65	58.20	72.85
43	35.70	49.50	59.60	73.00		**93**	33.25	51.40	59.20	75.55
44	34.20	48.50	58.25	73.70		**94**	37.00	50.65	61.40	74.85
45	34.65	50.15	58.65	74.10		**95**	36.65	49.85	60.20	73.90
46	33.30	51.50	61.70	74.65		**96**	34.15	49.55	60.10	74.75
47	34.10	49.10	58.15	75.40		**97**	35.35	48.50	60.10	73.65
48	33.20	48.70	60.70	74.85		**98**	34.50	50.90	60.60	73.50
49	36.25	51.30	61.50	75.70		**99**	36.65	50.95	60.65	74.95
50	33.05	51.25	59.05	74.25		**100**	36.95	50.75	58.65	75.00
Avg of 100 Attempts	**34.9105**	**49.962**	**59.887**	**74.1195**						

Case 3: Existing and DmRT Both from Ideal

1. Execution Time for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	353.77	431.32	311.98	245.41		**51**	351.01	431.04	326.33	257.07
2	385.41	440.73	314.82	251.2		**52**	380.98	440.19	312.9	250.72
3	393.82	454.92	304.59	247.47		**53**	356.83	442.66	305.45	241.66
4	388.45	458.95	335.99	240.09		**54**	369.79	452.61	319.48	259.44
5	365.02	454.27	333.68	240.62		**55**	381.35	454.94	308.97	253.43
6	371.13	438.78	331.64	248.7		**56**	350.67	440.42	312.98	241.08
7	391.62	439.24	315.72	244.27		**57**	376.48	432.81	313.4	246.26
8	355.12	452.83	309.69	257.23		**58**	377.1	450.86	326.97	253.7
9	359.71	459.21	319.62	251.38		**59**	390.34	436.74	330.11	254.6
10	399.55	444.82	320.94	252.64		**60**	387.17	438.24	335.82	246.23
11	367.75	456.28	313.82	256.18		**61**	370.12	450.69	315.11	249.67
12	368.88	446.37	334.49	245.89		**62**	369.3	445.3	331.79	242.52
13	375.72	441.17	323.05	241.73		**63**	373.67	437.04	311.41	242.46
14	357.06	433.88	313.36	241.64		**64**	359.17	455.56	329.94	258
15	365.69	445.51	320.17	240.95		**65**	384.36	449.79	327.36	240.06
16	351.04	432.11	331.12	250.51		**66**	368.76	448.73	332	252.46
17	386.08	432.5	325.97	243.61		**67**	385.19	432.28	311.91	258.61
18	371.31	454.89	301.81	247.38		**68**	380.96	450.16	328.4	250.21
19	375.39	433.52	325.72	242.4		**69**	370.26	434.54	310.35	256.91
20	367.61	437.94	319.13	243.85		**70**	376.95	440.97	328.02	241.71
21	352.9	443.51	302	258.57		**71**	359.29	434.6	333.79	250.65
22	397.6	459.72	333.71	255.28		**72**	390.1	453.82	310.64	242.58
23	394.75	439.02	333.52	250.6		**73**	394.3	454.28	316.46	257.27
24	359.24	445.79	318.82	257.97		**74**	397.55	433.47	325.17	244.65
25	379.98	435.09	304.22	251.86		**75**	389.29	458.31	305.57	243.96
26	358.95	452.96	339.11	252.58		**76**	381.3	439.23	302.17	251.13
27	360.62	457.91	338.01	246.65		**77**	386.4	451.81	310.3	259.66
28	367.61	437.46	321.51	248.37		**78**	383.52	455.36	321.14	257.54
29	384.66	439.91	302.56	246.32		**79**	395.43	447.8	310.67	259.51
30	365.32	457.28	313.22	256.73		**80**	353.1	444.45	339.55	252.5
31	379	457.23	305.05	242.99		**81**	377.76	458.44	321.26	247.76
32	366.55	436.04	338.44	249.77		**82**	398.99	435.67	324.37	245.6
33	380.35	443.33	316.29	248		**83**	385.76	437.71	305.89	248.36
34	363.05	448.34	303.05	252.28		**84**	371.28	441.72	310.67	243.5
35	359.34	430.07	306.97	255.69		**85**	353.96	450.92	324.6	242.33
36	387.48	459.98	320.96	255.45		**86**	374.41	437.27	325.6	242.29
37	396.67	438.39	309.96	251.83		**87**	357.57	433.52	333.9	250.49
38	379.8	436.02	317.99	248.39		**88**	356.87	433.72	324.62	258.4
39	357.23	458.02	311.67	247.18		**89**	373.22	456.92	332.31	258.87
40	383.34	454.33	313.46	252.42		**90**	370.05	445.29	338.23	258.47
41	391.63	436.64	304.7	243.91		**91**	393.88	457.85	312.4	258.89
42	390.95	436.27	334.3	256.04		**92**	386.04	440.74	332.16	245.42
43	382.76	440.97	313.97	242.41		**93**	364.99	437.28	323.37	244.6
44	365.23	459.36	327.35	252.82		**94**	359.94	451.31	324.71	254.32
45	397.37	459.67	311.92	251.78		**95**	353.64	433.86	309.07	254.76
46	372.89	456.97	300.03	243.29		**96**	388.66	436.25	333.53	259.56
47	352.17	440.56	306.93	242.82		**97**	376.44	444.72	310.78	242.67
48	376.38	432.03	335.76	240.25		**98**	384.33	444.6	330.85	249.53
49	380.58	438.51	303.51	258.64		**99**	356.68	434.74	320.07	247.54
50	389.1	433.59	302.39	249.51		**100**	386.88	453.61	338.24	243.44
Avg of 100 Attempts	**374.8572**	**444.5905**	**319.6948**	**249.566**						

Case 3: Existing and DmRT Both from Ideal

2. Fragmentation for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	33.55	27.88	17.61	12.88		**51**	36.96	25.73	20.52	16.28
2	35.39	27.09	21.62	14.26		**52**	34.36	25.96	19.13	12.76
3	35.2	25.76	20.24	13.7		**53**	36.67	26.63	18.86	12.96
4	33.21	25.9	20.63	13.41		**54**	36.29	25.24	20.51	15.73
5	33.21	27.91	18.79	13.3		**55**	36.75	27.45	21.71	14.85
6	34.96	25.97	17.51	14.29		**56**	33.38	26.15	20.63	12.71
7	34.57	25.42	19.58	14.11		**57**	34.92	28.19	20.63	15.14
8	36.95	25.03	18.71	13.07		**58**	37.1	25.18	18.48	15.34
9	36.79	27.6	21.23	15.57		**59**	35.31	27	19.25	14.64
10	35.87	25.95	18.93	15.72		**60**	33.71	26.48	21.07	15.19
11	33.71	26.48	20.53	12.79		**61**	35.83	24.87	19.2	16.32
12	33.54	25.93	21.69	15.99		**62**	34.49	26.57	19.31	14.19
13	36.56	25.69	20.13	15.95		**63**	36.32	27.74	17.83	15.6
14	35.95	24.55	20.57	13.96		**64**	34.17	24.3	19.02	14.24
15	36.89	26.07	19.55	12.6		**65**	34.13	24.93	19.97	12.72
16	34.2	26.37	17.59	14.61		**66**	35.49	27.94	20.56	16.02
17	34.12	27.9	17.13	13.94		**67**	34.52	26.64	19.53	15.46
18	34.29	25.34	21.57	15.82		**68**	35.21	25.37	17.73	15.04
19	35.8	24.98	18.01	15.71		**69**	34.84	28.04	19.54	13.54
20	35.02	25.46	19.3	13.06		**70**	36.12	24.63	21.44	12.69
21	35.79	26.27	16.81	13.99		**71**	33.27	24.52	21.34	14.38
22	33.21	27.25	21.57	13.02		**72**	34.61	24.72	17.59	14.16
23	34.72	25.3	18.68	15.31		**73**	33.6	25.61	17.35	13.56
24	35.16	27.18	17.75	15.43		**74**	33.61	24.91	19.18	16.42
25	34.5	27.35	18.9	14.22		**75**	36.3	27.55	19.17	15.3
26	35.25	28.11	17.4	16.53		**76**	36.85	27.89	19.44	15.85
27	36.42	27.32	18.04	16.44		**77**	35.89	24.55	17.52	14.69
28	35.84	24.99	17.32	16.27		**78**	35.1	24.98	19.43	15.53
29	34.24	26.1	19.97	15.18		**79**	36.05	27.7	17.02	15.62
30	33.49	24.57	18.71	15.69		**80**	35.28	25.9	20.56	13.67
31	36.2	25.93	19.75	14.26		**81**	35.76	26.63	20.77	14.75
32	37	28.19	21.51	16.43		**82**	35.88	27.38	17.58	12.88
33	34.2	28.09	17.12	14.2		**83**	34.03	25.18	17.69	14.98
34	36.84	26.27	19.21	12.99		**84**	36.77	24.72	17.34	14.03
35	35.93	27.76	21.7	14.38		**85**	35.83	28.06	17.08	15.76
36	35.52	25.92	19.43	15		**86**	34.53	27.7	18.76	14.31
37	35.97	27.88	19.53	14.83		**87**	35.8	25.13	18.41	15.88
38	35.66	27.11	17.74	13.27		**88**	33.59	26.34	19.21	14.37
39	34.49	26.33	20.33	15.39		**89**	34.82	26.55	18.83	14.85
40	33.28	27.06	18.83	13.56		**90**	35.31	25.4	18.27	14.05
41	35	28.21	21.72	12.91		**91**	34.63	25.84	21.2	15.5
42	34.27	28.1	19.65	15.86		**92**	36.09	26.78	18.03	13.71
43	33.59	25.39	20.83	13.95		**93**	34.55	27.56	19.3	14.07
44	35.49	25.39	18.89	14.22		**94**	36.01	25.25	20.31	16.17
45	35.86	27.82	21.16	13.59		**95**	36.51	28.16	19.27	13.31
46	37.03	26.33	19.32	15.72		**96**	36.22	25.01	20.29	15.33
47	33.32	27.86	20.2	15.72		**97**	36.09	24.48	20.08	15.77
48	36.98	27.39	18.02	15.49		**98**	35.45	27.07	17.4	14.5
49	35.39	24.79	19.65	13.12		**99**	34.7	27.81	18.01	13.79
50	35.62	28.04	18.27	12.73		**100**	36.04	27.02	18.44	14.76
Avg of 100 Attempts	**35.2178**	**26.3902**	**19.2872**	**14.5781**						

Case 3: Existing and DmRT Both from Ideal

3. No. of Request Satisfied for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	67	72	84	89		**51**	69	72	83	87
2	67	72	81	91		**52**	69	73	82	88
3	67	72	81	91		**53**	68	72	83	88
4	69	71	85	89		**54**	66	71	83	90
5	69	72	82	90		**55**	69	70	85	88
6	68	70	82	88		**56**	66	70	85	87
7	67	70	82	89		**57**	66	74	82	91
8	69	72	83	90		**58**	69	71	81	92
9	66	74	83	88		**59**	68	74	81	90
10	68	74	85	90		**60**	68	72	84	90
11	69	70	84	91		**61**	66	73	83	90
12	68	74	81	88		**62**	69	71	85	90
13	66	71	85	88		**63**	66	72	84	89
14	68	74	82	88		**64**	69	74	83	88
15	67	74	85	90		**65**	67	72	83	89
16	67	72	83	91		**66**	69	71	85	87
17	66	73	81	89		**67**	69	70	82	89
18	69	74	81	90		**68**	69	74	85	88
19	69	74	82	90		**69**	66	73	85	91
20	69	74	81	92		**70**	68	70	84	92
21	68	72	82	89		**71**	69	74	84	91
22	68	72	82	92		**72**	68	71	83	90
23	67	71	85	91		**73**	67	74	83	88
24	69	73	82	87		**74**	67	74	83	89
25	67	73	84	90		**75**	69	70	82	89
26	66	70	82	87		**76**	66	71	82	89
27	66	72	84	89		**77**	66	72	83	92
28	68	74	83	87		**78**	67	70	82	89
29	66	72	82	91		**79**	68	70	83	91
30	68	72	85	89		**80**	67	73	81	90
31	66	72	82	91		**81**	67	73	83	88
32	67	71	83	91		**82**	68	71	83	90
33	66	73	83	87		**83**	66	73	85	87
34	66	74	85	90		**84**	69	71	82	88
35	66	70	82	91		**85**	68	74	85	91
36	68	75	83	89		**86**	67	71	83	87
37	66	72	81	89		**87**	66	70	84	87
38	66	73	83	89		**88**	67	71	83	90
39	67	72	83	89		**89**	67	70	82	90
40	69	72	84	87		**90**	66	74	84	90
41	69	73	84	88		**91**	69	74	85	87
42	69	72	84	87		**92**	67	74	82	88
43	67	73	84	89		**93**	68	74	84	88
44	66	73	84	87		**94**	67	73	82	89
45	68	75	82	88		**95**	69	72	83	91
46	69	73	84	87		**96**	68	70	83	89
47	69	74	85	90		**97**	66	70	84	89
48	66	72	83	89		**98**	69	73	85	91
49	69	71	82	91		**99**	68	70	83	89
50	66	72	83	88		**100**	66	74	84	92
Avg of 100 Attempts	**67.5358**	**72.1999**	**83.1096**	**89.1851**						

Case 3: Existing and DmRT Both from Ideal

1. Execution Time for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	2015.03	3092.89	1572.91	1135.68		**51**	2178.11	3172.66	1430.46	1070.35
2	2128.48	3133.45	1549.06	1183.93		**52**	2103.63	3094.45	1602.11	1288.17
3	2028.96	3134.05	1456.74	988.15		**53**	2074.68	3246.39	1450.57	1079.19
4	2053.69	3388.73	1452.07	1250.64		**54**	2086.03	3130.27	1596.12	1166.11
5	2193.15	3102.83	1533.23	1135.34		**55**	2124.84	3219.63	1584.69	1090.88
6	2125.35	3319.15	1499.6	991.21		**56**	2026.05	3171.7	1613.73	1143.28
7	2167.34	3403.51	1618.58	951.79		**57**	2173.72	3358.35	1476.32	1166.5
8	2094.63	3273.89	1621.15	1203.41		**58**	2046.46	3240.79	1500.49	1113.24
9	2095.65	3262.5	1441.84	1071.09		**59**	2056.56	3357.17	1619.23	1325
10	2147.67	3374.85	1482.34	968.63		**60**	2016.47	3368.22	1452.48	1229.67
11	2022.36	3109.91	1605.21	1209.37		**61**	2151.52	3404.19	1587.57	1287.03
12	2092.23	3334.33	1425.02	1104.1		**62**	2108.84	3119.4	1610.07	1088.84
13	2174.06	3322.3	1544.48	1299.22		**63**	2169.37	3131.92	1538.74	1184.32
14	2086.68	3289.04	1446.31	1145.42		**64**	2134.23	3435.79	1541.04	1016.32
15	2081.32	3303.84	1433.66	1099.06		**65**	2083.54	3320.71	1616.94	1072.48
16	2118.33	3236.31	1565.61	1335.9		**66**	2023.14	3422.14	1596.82	1342.56
17	2191.39	3119.68	1560.79	1305.35		**67**	2150.61	3279.29	1617.99	1068.64
18	2108.18	3310.46	1520.14	1298.41		**68**	2140.2	3372.14	1510.67	1040.01
19	2209.23	3219.2	1620.64	1258.66		**69**	2182.6	3392.61	1601.02	1151.01
20	2102.71	3181.71	1483.39	1092.3		**70**	2126.96	3164.98	1535.41	1196.59
21	2067.05	3346.64	1507.67	1045.99		**71**	2062.02	3173.51	1511.81	1033.72
22	2167.86	3240.34	1616.29	994.53		**72**	2019.36	3042.3	1512.22	1234.18
23	2104.7	3318.71	1558.22	1191.16		**73**	2181.2	3111.12	1550.23	1290.57
24	2112.58	3234.93	1431.8	1221.25		**74**	2207.85	3302.35	1546.25	1056.61
25	2094.51	3053.92	1523.3	1302.21		**75**	2131.63	3091.72	1621.79	1174.28
26	2023.45	3431.74	1610.86	1230.46		**76**	2093.71	3095.76	1511.44	966.84
27	2185.7	3337.67	1510.72	1144.63		**77**	2059.39	3174.24	1544.21	1333.24
28	2082.04	3073.06	1615.95	1301.83		**78**	2092.72	3042.53	1499.2	961.78
29	2050.66	3192.04	1457.91	1049.8		**79**	2123.94	3204.16	1593.62	1134.1
30	2024.16	3045.46	1440.68	1246.93		**80**	2100.33	3286.6	1461.75	1128.14
31	2119.36	3103.75	1499.34	1070.49		**81**	2036.28	3290.47	1536.01	1336.69
32	2199.5	3260.19	1437.45	1036.37		**82**	2123.45	3281.01	1616.73	1144.03
33	2093.12	3229.73	1575.74	957.92		**83**	2147.09	3313.96	1527.3	1322.44
34	2130.33	3225.15	1527.89	1308.06		**84**	2192.15	3198.82	1553.54	1058.19
35	2157.03	3060.49	1465.66	1297.28		**85**	2193.6	3124.45	1423.56	1110
36	2072.57	3357.15	1430.76	1099.45		**86**	2070.99	3154.96	1512.87	1237.66
37	2205.85	3416.37	1607.7	1158.02		**87**	2208.57	3110.82	1539.7	1287.71
38	2202.83	3418.44	1429.59	1092.43		**88**	2024.71	3385.27	1511.66	1303.31
39	2033.25	3049.5	1544.26	966.95		**89**	2026.42	3093.9	1600.74	1219.4
40	2103.46	3293.52	1463.12	1292.07		**90**	2036.58	3107.81	1603.62	1096.86
41	2056.3	3380.53	1505.43	968.3		**91**	2193.56	3389.48	1604.03	1264.77
42	2110.12	3087.18	1588.98	1193.99		**92**	2175.84	3284.78	1452.42	1181.45
43	2106.55	3122.38	1621.77	1189.81		**93**	2051.34	3196.62	1437.74	1211.74
44	2173.87	3339.35	1601.47	995.17		**94**	2067.23	3329.8	1464.18	1068.27
45	2017.19	3291.11	1463.16	1118.34		**95**	2058.4	3352.63	1474.81	1165.18
46	2198.38	3329.49	1532.3	1105.41		**96**	2114.43	3231.44	1590.84	1029.21
47	2133.67	3432.5	1484.12	986.14		**97**	2193.69	3346.98	1516.6	1131.49
48	2119.68	3321.62	1551.89	1174.83		**98**	2094.89	3046.28	1583.43	1274.88
49	2135.77	3309.13	1537.01	1066.52		**99**	2035.67	3121.62	1540.06	962.54
50	2194.73	3430.26	1426.83	976.78		**100**	2070.72	3119.55	1502.27	1298.19
Avg of 100 Attempts	**2110.58**	**3240.527**	**1530.277**	**1149.484**						

Case 3: Existing and DmRT Both from Ideal

2. Fragmentation for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	43.21	34.93	27.72	17.75		**51**	44.55	36.66	27.56	17.95
2	42.34	37.3	24.54	20.37		**52**	44.69	35.9	25.08	22.03
3	44.11	33.56	25.96	19.17		**53**	41.57	35.34	25.7	22.16
4	42.57	36.26	27.31	21.35		**54**	42.15	37.08	24.96	21.04
5	44.73	35.37	27.84	20.29		**55**	41	37.41	24.9	18.74
6	44.18	35.01	26.74	17.48		**56**	42.72	35.78	26.81	18.94
7	43.92	34.28	26.63	17.84		**57**	44.84	35.86	27.01	18.53
8	44.64	35.03	26.19	18.43		**58**	43.47	35.38	26.02	20.37
9	42.7	33.97	24.8	17.49		**59**	43.03	33.73	27.78	17.49
10	44.77	35.11	27.18	17.6		**60**	42.9	34.47	25.32	19.83
11	43.07	35.56	25.68	20.85		**61**	42.72	36.43	26.02	21.96
12	41.71	36.65	25.49	17.63		**62**	42.47	37.14	25.48	18.48
13	41.76	35.47	26.54	17.47		**63**	43.49	34.84	26.6	19.97
14	42.5	33.64	27.58	18.66		**64**	41.44	36.94	26.14	21.66
15	43.78	35.02	27.83	17.43		**65**	43.34	36.57	24.6	19.27
16	43.92	36.84	27.74	18.68		**66**	42.69	35.17	25.43	21.25
17	43.96	34	27.65	20.31		**67**	43.46	34.21	28.11	20.17
18	42.31	34.31	25.39	19.45		**68**	43.78	35.27	26.49	19.33
19	40.98	35.56	24.93	19.1		**69**	42.19	36.41	26.34	17.56
20	42.15	35.08	28.11	20.02		**70**	44.75	34.13	25.15	20.85
21	41.83	34.51	27.51	22.3		**71**	42.05	36.6	24.5	21.71
22	44.77	34.18	25.99	19.95		**72**	42.07	35.36	27.85	21.2
23	42.86	33.64	26.93	21.68		**73**	44.79	35.33	24.51	19.66
24	41.61	37.33	26.48	19.18		**74**	43.89	35.44	27.76	18.32
25	43.32	36.25	27.67	17.86		**75**	42.74	34.24	25.25	17.7
26	43.88	37.12	24.74	19.78		**76**	43.93	36.93	27.16	22.17
27	43.88	33.68	26.5	21.84		**77**	43.36	36.91	28.2	20.71
28	44.37	36.26	27.67	20.35		**78**	41.08	33.96	27.93	18.9
29	42.47	36.04	26.65	19.63		**79**	43.76	34.98	25.34	18.59
30	41.05	37.24	27.57	18.08		**80**	43.3	35.58	26.68	20.1
31	41.94	35.17	25.55	18.69		**81**	43.71	36.55	25.32	20.59
32	42.88	34.04	26.23	18.49		**82**	42.54	34.23	27.16	20.25
33	43.62	35.42	24.96	21.33		**83**	42.22	35.43	24.39	17.68
34	41.08	37.15	24.89	21.54		**84**	41.33	36.66	26.15	20.44
35	44.17	34.22	25.36	20.12		**85**	41.58	33.67	25.01	20.67
36	44.58	37.22	27.53	20.32		**86**	43.82	35.43	27.78	20.93
37	43.51	34.66	27.4	21.52		**87**	41.95	34.27	27.06	22.39
38	44.02	36.94	27.54	18.32		**88**	43.47	36.27	25.44	18.44
39	44.16	34.47	27.57	21.47		**89**	41.7	36.01	24.97	18.13
40	42.78	36.86	28.15	19.89		**90**	42.26	36.23	26.12	21.53
41	41.72	37.5	27.81	18.56		**91**	41.63	35.63	25.64	20.96
42	41.23	33.59	25.03	18.48		**92**	42.47	36.25	26.53	19.89
43	43.45	34.03	26.37	19.01		**93**	43.26	36.05	27	22.15
44	43.36	36.33	25.63	21.74		**94**	41.99	36.24	24.47	21.84
45	44.59	35.32	27.55	18.22		**95**	44.46	34.79	26.02	22
46	44.13	36.15	26	20.23		**96**	40.92	37.32	25.71	20.56
47	43.48	37.45	26.2	18.85		**97**	41.98	36.68	24.48	21.54
48	42.25	35.69	27.06	18.04		**98**	43.27	37.25	27.03	21.57
49	44.75	35.83	26.04	20.83		**99**	41.99	35.3	24.74	19.17
50	44.26	33.71	27.08	17.88		**100**	44.4	33.71	26.87	19.62
Avg of 100 Attempts	**43.0248**	**35.5497**	**26.3408**	**19.7854**						

Case 3: Existing and DmRT Both from Ideal

3. No. of Request Satisfied for all allocators 100 attempts (Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	55.5	63.5	72.0	80.3		**51**	53.3	62.3	73.5	82.0
2	54.2	64.4	73.1	80.0		**52**	57.5	63.9	75.0	81.9
3	55.2	65.0	72.2	82.3		**53**	56.7	63.8	73.1	80.9
4	54.3	63.7	71.7	80.5		**54**	54.3	66.7	73.0	83.1
5	55.4	64.4	71.8	81.4		**55**	57.5	62.7	74.3	79.2
6	58.1	66.5	75.3	81.4		**56**	53.8	62.9	73.6	79.6
7	55.8	62.7	74.0	79.4		**57**	54.7	64.3	74.2	81.2
8	54.8	63.8	71.6	82.7		**58**	55.4	62.5	75.0	80.7
9	55.5	65.7	72.0	79.4		**59**	57.6	65.9	72.8	81.1
10	53.7	66.6	71.6	82.0		**60**	53.6	66.5	75.2	82.3
11	55.9	66.0	75.0	82.2		**61**	58.2	65.9	71.4	79.5
12	53.8	63.6	71.9	82.6		**62**	54.4	65.1	75.2	80.9
13	55.4	65.4	72.1	80.6		**63**	55.3	64.2	75.3	80.2
14	55.8	62.8	72.3	82.6		**64**	55.9	67.2	72.8	79.3
15	55.0	63.6	73.8	82.0		**65**	57.9	65.6	74.0	81.2
16	55.0	63.7	71.5	81.1		**66**	54.0	66.4	72.3	81.4
17	55.9	62.7	71.3	80.1		**67**	53.7	64.4	75.3	81.9
18	55.9	64.6	72.8	79.4		**68**	55.6	67.3	72.5	83.2
19	55.3	67.3	74.0	81.8		**69**	56.2	65.6	74.1	80.0
20	56.2	66.1	74.1	80.5		**70**	55.8	66.1	73.0	81.6
21	54.6	66.8	71.8	81.6		**71**	56.6	62.6	73.5	82.8
22	58.2	64.0	72.3	82.7		**72**	55.0	63.7	73.0	79.3
23	53.8	62.9	71.6	82.8		**73**	54.0	66.1	74.5	81.6
24	57.4	62.5	74.8	81.3		**74**	56.9	63.3	74.1	80.6
25	56.7	62.8	74.0	81.7		**75**	54.4	64.0	72.0	80.9
26	53.5	62.8	72.3	81.6		**76**	55.0	65.5	74.7	81.6
27	54.5	65.0	72.6	80.2		**77**	54.1	62.6	72.6	79.3
28	57.9	65.1	74.8	79.9		**78**	54.0	64.3	74.5	81.3
29	55.6	66.5	75.1	79.9		**79**	58.0	65.1	74.7	79.5
30	57.2	66.8	71.5	81.6		**80**	56.8	65.9	74.9	79.4
31	53.3	65.4	73.6	80.8		**81**	54.6	66.7	75.0	82.3
32	57.9	64.5	73.5	81.5		**82**	54.2	64.8	72.9	81.9
33	55.5	62.8	74.4	81.1		**83**	55.1	66.7	71.7	81.6
34	57.4	63.6	75.1	80.2		**84**	58.2	66.1	73.1	81.8
35	57.6	66.4	73.6	82.4		**85**	53.3	62.8	74.2	80.5
36	53.5	66.3	71.8	81.7		**86**	53.6	63.4	71.4	80.0
37	55.6	62.4	74.3	79.8		**87**	53.4	63.7	74.9	79.8
38	56.4	65.1	73.7	82.2		**88**	54.7	63.9	73.8	82.2
39	57.9	62.7	72.6	81.8		**89**	56.3	64.9	71.4	83.2
40	56.6	66.6	71.7	80.2		**90**	53.3	65.2	73.6	81.0
41	57.9	67.3	72.2	82.1		**91**	54.5	65.7	75.2	83.1
42	57.0	66.7	71.6	82.7		**92**	53.3	64.5	71.7	80.7
43	54.7	66.6	74.8	82.1		**93**	57.7	64.9	74.7	79.7
44	58.2	67.2	71.8	80.6		**94**	57.6	62.9	74.8	81.5
45	57.0	66.8	71.5	82.7		**95**	55.4	65.2	73.0	82.7
46	53.9	62.9	75.0	80.5		**96**	56.2	66.7	71.4	81.7
47	56.7	63.4	71.8	82.9		**97**	53.5	62.4	75.0	82.4
48	54.8	66.2	75.2	81.3		**98**	54.5	63.8	72.7	79.6
49	57.4	67.1	73.2	81.0		**99**	57.1	63.3	75.3	81.0
50	56.6	62.5	72.0	79.9		**100**	55.2	62.5	74.7	80.6
Avg of 100 Attempts	**55.5939**	**64.722**	**73.3219**	**81.1776**						

Case 3: Existing and DmRT Both from Ideal

1. Execution Time for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	3231.3	4463.78	2125.59	1929.62		**51**	3204.49	4451.17	2164.38	1807.3
2	3215.27	4480.67	2169.75	1939.02		**52**	3199.52	4498.17	2178.71	1909.59
3	3335.99	4421.48	2161.43	1948.21		**53**	3261.08	4453.7	2130.85	1781.25
4	3327.79	4420.94	2185.11	1912.68		**54**	3345.39	4427.32	2162.95	1875.54
5	3308.11	4421.22	2204.64	1766.95		**55**	3339.68	4479.24	2205.61	1897.83
6	3210.39	4438.31	2173.1	1890.93		**56**	3186.74	4440.96	2160.18	1781.41
7	3202.77	4448.53	2161.78	1932.78		**57**	3260.89	4424.72	2192.64	1762.53
8	3196.96	4425	2131.69	1929.31		**58**	3342.69	4451.77	2164.36	1890.75
9	3268.49	4508.34	2144.1	1829.77		**59**	3201.57	4474.23	2195.2	1900.43
10	3344.73	4467.6	2177.05	1933.8		**60**	3238.13	4454.45	2170.37	1867.34
11	3342.48	4440.22	2204.36	1947.88		**61**	3294.7	4498.37	2145.87	1850.23
12	3191.04	4439.79	2124.74	1876.76		**62**	3312.65	4509.89	2195.14	1912.38
13	3309.29	4418.47	2180.72	1866.73		**63**	3340.73	4452.06	2213.74	1944.71
14	3289.21	4429.88	2141.18	1872		**64**	3235.1	4470.69	2178.16	1782.43
15	3297.9	4503.97	2170.56	1839.78		**65**	3365.46	4439.6	2203.67	1807.15
16	3366.61	4513.89	2176.34	1830.55		**66**	3317.04	4480.24	2156.16	1872.44
17	3263.85	4502.63	2159.76	1885.41		**67**	3256.58	4505.38	2163.02	1881.63
18	3245.33	4460.42	2129.95	1855.35		**68**	3247.58	4506.74	2164.38	1814.96
19	3247.46	4437.75	2164.92	1849.27		**69**	3190.99	4472.58	2200.78	1820.54
20	3191.04	4487.79	2140.91	1854.86		**70**	3306.06	4438.8	2149.77	1875.9
21	3360.15	4483.17	2183.19	1870.24		**71**	3280.21	4460.22	2141.1	1838.5
22	3343.22	4480.19	2142.26	1835.62		**72**	3330.77	4503.63	2170.63	1854.4
23	3233.99	4440.37	2151.02	1852.83		**73**	3367.83	4465.46	2209.96	1852.96
24	3340.42	4492.37	2152.53	1813.28		**74**	3221.09	4484.6	2220.25	1768.33
25	3256.72	4508.26	2148.08	1946.46		**75**	3276.72	4504.05	2144.79	1929.78
26	3325.96	4516.1	2197.07	1784.02		**76**	3261.01	4501.11	2172.32	1832.6
27	3303.63	4510.24	2126.43	1910.45		**77**	3333.67	4421.59	2134.79	1960.04
28	3245.08	4483.41	2152.02	1777.23		**78**	3254.87	4482.51	2200.27	1946.25
29	3288.39	4488.92	2168.6	1952.77		**79**	3194.3	4499.13	2178.69	1926.35
30	3317.64	4488.64	2130.59	1827.58		**80**	3179.96	4439.51	2216.96	1781.02
31	3358.54	4430.35	2160.52	1877.52		**81**	3276.72	4466.55	2152.4	1927.35
32	3189.63	4448.46	2187.06	1768.09		**82**	3317.29	4499.21	2164.54	1822.86
33	3340.58	4428.43	2205.14	1827.77		**83**	3317.4	4498.76	2209.07	1920
34	3340.36	4485.72	2131.41	1764.36		**84**	3288.6	4433.89	2203.29	1864.99
35	3283.04	4417.94	2190.51	1800.38		**85**	3371.41	4498.03	2206.38	1895.33
36	3319.23	4512.14	2213.78	1909.41		**86**	3341.53	4420.55	2161.37	1854.99
37	3197.56	4463.65	2138.41	1791.46		**87**	3194.33	4477.48	2160.8	1909.18
38	3206.05	4482.6	2221.25	1872.16		**88**	3350.29	4481.98	2165.71	1787.4
39	3202.65	4459.39	2153.12	1934.76		**89**	3208.21	4434.57	2145.02	1860.89
40	3357.95	4472.31	2202.73	1940.1		**90**	3190.39	4501.73	2216.22	1864.38
41	3374.05	4432.94	2169.96	1792.13		**91**	3290.15	4514.25	2163.85	1918.63
42	3366.26	4437.13	2177.77	1926.79		**92**	3278.1	4486.74	2215.75	1929.04
43	3190.24	4450.36	2202.84	1853.37		**93**	3335.91	4508.69	2218.92	1774.12
44	3293.69	4469.73	2165.7	1892.92		**94**	3224.46	4429.59	2196.54	1814.76
45	3237.79	4458.52	2175.46	1812.01		**95**	3341.31	4442.94	2187.55	1772.6
46	3208.17	4480.2	2195.58	1816.66		**96**	3241.8	4497.17	2180.79	1777.4
47	3258.85	4422.77	2188.5	1930.7		**97**	3207.57	4489.6	2219.02	1947.38
48	3234.45	4429.27	2152.1	1788.98		**98**	3355.91	4504.47	2171.56	1801.98
49	3275.33	4442.02	2128.65	1804.21		**99**	3265.33	4447.03	2189.3	1812.43
50	3326.1	4436.66	2173.42	1862.65		**100**	3236.85	4502.11	2138.65	1821.25
Avg of 100 Attempts	**3277.428**	**4467.102**	**2172.658**	**1860.321**						

Case 3: Existing and DmRT Both from Ideal

2. Fragmentation for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	52.25	43.22	35.76	24.91		**51**	54.64	44.2	34.28	22.84
2	53.37	45.19	34.09	22.91		**52**	53.87	43.77	36.46	26.41
3	51.11	42.88	36.9	23.34		**53**	52.27	45.27	34.94	27.13
4	50.9	44.45	37.24	23.99		**54**	50.96	42.12	36.67	23.91
5	51.32	42.91	35.03	26.76		**55**	54.54	43.34	37.95	24.76
6	51.09	44.93	36.81	27.03		**56**	51.62	45.55	36.4	23.43
7	52.52	45.06	36.41	26.9		**57**	52.04	42.84	35.85	25.73
8	52.88	44.2	36.1	25.84		**58**	50.7	42.64	34.42	23.09
9	53.91	43.5	35.55	25.09		**59**	51.01	45.21	37.68	26.17
10	51.1	44.76	35.57	23.82		**60**	52.27	43.66	37.66	27.25
11	51.47	42.71	34.41	26.4		**61**	50.96	43.9	34.75	23.24
12	51.75	44.68	37.58	23.27		**62**	53.79	43.69	34.32	26.15
13	51.87	42.47	37.37	24.08		**63**	53.96	43.08	37.51	27.39
14	53.63	45.42	37.55	23.42		**64**	53.63	45.27	34.52	26.21
15	51.65	44.5	34.46	26.75		**65**	51.5	41.93	36.62	27.22
16	52.39	43.77	37.86	23.52		**66**	52.15	44.9	35.41	24.4
17	53.91	43.66	35.87	24.65		**67**	50.82	43	35.8	26.91
18	52.02	42.3	35.37	23.82		**68**	52.27	44.12	35.75	24.38
19	51.51	45.33	37.2	24.29		**69**	53.13	42.82	36.72	24.47
20	52.73	44.91	34	24.79		**70**	50.93	45.18	34.97	26.86
21	50.99	45.4	36.45	25.87		**71**	53.86	43.5	37.77	25.1
22	51.77	45.53	34.67	24.21		**72**	51.74	41.88	37.13	27.19
23	51.82	42.06	34.75	25.84		**73**	53.57	43.26	34.42	25.71
24	53.72	43.68	35.74	23.75		**74**	52.5	44.17	36.51	23.07
25	53.6	42.46	35.13	24.87		**75**	52.2	44.52	35.69	23.01
26	53.19	42.8	36.82	24.42		**76**	54.03	44.45	35.19	25.92
27	51.95	43.68	35.56	23.34		**77**	54.62	45.2	36.32	27.66
28	51.64	43.32	35.4	26.92		**78**	53.6	43.84	37.92	23.08
29	51.07	41.88	34.49	23.58		**79**	54.4	44.04	37.14	25.37
30	54.13	41.72	37.78	24.95		**80**	52.19	42.55	37.12	27.45
31	52.63	44.03	37.44	24.79		**81**	52.7	43.87	36.2	23.43
32	51.59	43.29	35.57	26.9		**82**	53.72	44.09	36.54	26.58
33	52.08	43.41	37.93	27.69		**83**	52.37	41.62	35.68	22.84
34	52.1	42.72	36.85	26.17		**84**	54.29	43.45	34.5	24.32
35	52.26	44.98	35.81	25.68		**85**	51.07	43.48	37.38	25.48
36	53.8	43.99	34.32	26.05		**86**	51.95	41.78	35.77	23.24
37	54.62	42.78	35.33	25.2		**87**	54.41	43.23	35.38	22.89
38	53.12	45.09	37.55	25.89		**88**	51.72	42.48	34.23	23.79
39	53	44.77	34.38	24.27		**89**	51.45	42.8	34.25	25.19
40	52.07	45.15	36.88	23.44		**90**	52.83	41.98	36.38	27.19
41	51.37	42.45	34.81	26.38		**91**	54.28	45.2	34	24.24
42	54.24	42.23	36.87	26.5		**92**	53.4	44.69	36.66	24.34
43	54.54	42.57	36.76	23		**93**	54.37	41.71	36.39	24.32
44	51.85	45.27	35.22	26.93		**94**	51.61	43.37	34.94	26.23
45	54.13	44.78	36.68	26.23		**95**	52.65	45.27	35.18	25.56
46	53.41	42.23	37.06	23.8		**96**	52.96	42.81	35.26	24.74
47	50.94	44.72	35.38	25.22		**97**	52.6	42.37	35.3	27.08
48	51.47	42.1	37.76	26.38		**98**	52.09	43.92	34.38	22.75
49	53.61	44.49	35.92	27.34		**99**	53.56	42.79	36.11	24.09
50	51.99	44.33	37.48	25.14		**100**	54.27	42.45	35.13	25.98
Avg of 100 Attempts	**52.6015**	**43.6602**	**35.9747**	**25.1212**						

Case 3: Existing and DmRT Both from Ideal

3. No. of Request Satisfied for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	46.50	54.80	63.05	73.40		**51**	45.25	54.75	65.25	74.20
2	45.95	51.55	62.55	75.65		**52**	44.25	53.65	63.75	75.20
3	45.55	52.05	65.45	73.35		**53**	45.10	51.25	63.60	74.75
4	45.75	51.95	66.65	74.15		**54**	44.40	53.25	63.25	73.45
5	43.10	53.30	63.35	72.25		**55**	43.70	51.35	62.05	75.35
6	46.40	53.60	62.95	72.55		**56**	45.95	51.15	62.55	74.85
7	43.20	52.00	64.60	73.65		**57**	46.00	52.30	65.60	74.45
8	46.30	53.55	65.10	74.85		**58**	46.85	52.95	65.95	74.85
9	45.05	53.30	64.15	75.60		**59**	46.55	51.70	65.00	75.15
10	45.15	53.30	63.05	75.15		**60**	44.60	54.35	62.05	75.75
11	44.20	54.35	65.05	74.00		**61**	45.75	52.90	66.45	75.65
12	43.10	51.60	66.25	75.75		**62**	45.45	53.10	63.20	75.55
13	44.40	53.00	64.80	72.20		**63**	46.00	53.25	65.20	75.60
14	43.95	51.50	65.55	75.55		**64**	44.60	54.65	63.30	75.70
15	44.35	54.25	66.25	72.40		**65**	45.30	54.65	66.95	73.10
16	45.80	52.10	65.90	72.75		**66**	43.65	52.25	66.55	74.10
17	43.10	51.95	62.10	74.60		**67**	44.10	53.55	64.55	74.10
18	46.10	51.00	62.80	74.45		**68**	43.30	54.75	66.70	74.35
19	43.00	53.05	65.00	73.00		**69**	46.35	51.35	64.30	74.95
20	46.35	51.50	65.70	72.30		**70**	45.00	53.35	66.35	73.95
21	44.75	52.15	64.00	75.25		**71**	45.40	51.75	62.20	75.80
22	44.90	51.00	63.50	72.75		**72**	43.10	53.80	63.20	72.00
23	43.05	54.00	62.05	73.30		**73**	44.35	52.45	62.60	73.55
24	46.40	53.65	64.85	74.30		**74**	43.20	53.65	64.35	75.05
25	46.10	52.10	63.55	74.65		**75**	46.60	54.55	64.20	73.00
26	45.60	52.50	62.75	72.20		**76**	43.80	51.35	66.00	72.20
27	46.55	52.20	65.00	74.05		**77**	43.20	53.50	64.05	73.70
28	46.60	53.20	65.00	72.05		**78**	45.75	54.90	63.00	75.40
29	45.55	52.00	63.50	74.80		**79**	44.80	51.10	63.90	72.50
30	44.55	54.10	62.15	73.30		**80**	46.00	51.70	62.15	74.50
31	44.25	53.35	65.60	72.35		**81**	46.20	51.45	64.30	72.65
32	44.60	52.45	65.00	74.55		**82**	45.70	53.45	62.60	72.80
33	45.15	53.20	65.80	74.10		**83**	43.35	53.65	63.10	73.00
34	45.40	53.30	65.20	75.05		**84**	44.25	51.00	65.40	75.80
35	46.05	54.05	66.65	73.00		**85**	43.45	52.25	66.15	73.60
36	43.60	51.00	62.75	75.00		**86**	43.70	54.50	65.10	75.50
37	45.00	52.75	64.25	75.85		**87**	43.40	53.00	66.90	75.35
38	43.20	51.90	62.25	72.05		**88**	43.25	53.25	66.75	72.75
39	43.80	51.80	64.05	75.70		**89**	45.45	53.75	66.60	72.50
40	46.70	51.95	66.00	72.80		**90**	45.40	51.45	66.00	72.90
41	43.85	53.20	63.95	75.85		**91**	46.90	54.70	62.65	72.90
42	44.15	53.85	64.70	75.40		**92**	47.00	52.55	62.70	74.90
43	46.65	54.55	66.30	74.80		**93**	43.10	52.25	65.75	72.40
44	43.05	53.75	65.70	72.75		**94**	46.30	51.95	63.60	73.70
45	44.15	54.90	64.90	72.35		**95**	44.20	51.65	64.65	72.65
46	43.00	51.25	66.40	74.50		**96**	44.60	52.15	66.60	73.85
47	44.90	53.15	65.75	75.80		**97**	46.00	51.55	65.00	75.90
48	44.55	53.00	62.15	74.80		**98**	43.10	52.35	62.55	74.30
49	44.30	53.70	65.15	75.95		**99**	43.35	51.30	65.25	72.50
50	44.30	52.40	63.55	72.10		**100**	44.35	53.75	64.25	73.60
Avg of 100 Attempts	**44.834**	**52.813**	**64.469**	**74.053**						

Case 4: Existing and DmRT follow Local -> Shared -> Ideal

1. Execution Time for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	533.79	622.34	473.01	402		**51**	534.51	627.87	483.67	371.61
2	529.31	643.69	468	373.58		**52**	542.74	623.75	480.8	389.6
3	530.88	646.64	486.83	401.61		**53**	522.61	628.28	487.47	397.18
4	521.11	646.23	477.77	384.02		**54**	537.58	627.41	483.74	386.63
5	535.38	639.43	493.33	364.95		**55**	528.33	638.93	470.75	382.04
6	516.67	633.83	472.52	399.13		**56**	522.63	641.18	467.32	381.48
7	512.54	638.71	495.78	364.45		**57**	531.9	645.93	487.21	392.35
8	515.22	637.29	466.43	371.61		**58**	530.02	631.7	493.86	370.24
9	543.81	633.16	473.06	405.93		**59**	538.14	635.1	468.38	378.09
10	536.7	629.68	482.71	399.23		**60**	534.74	631.34	490.26	403.26
11	535.62	632.98	473.26	394.77		**61**	545.71	651.77	475.21	381.44
12	525.84	636.29	476.52	398.21		**62**	538.38	625.72	466.36	396.66
13	535.69	630.72	481.72	406.94		**63**	514.61	651.22	492.46	367.44
14	511.25	649.26	492.89	396.85		**64**	542.26	643.22	487.57	380
15	538.72	636.02	476.65	371.44		**65**	540.4	648.96	489.55	357.7
16	518.05	651.21	468.54	367.48		**66**	531.36	636.3	474.04	405.57
17	523.49	644.25	482.71	397.74		**67**	525.18	622.55	482.11	404.29
18	544.47	650.6	481.08	402.12		**68**	524.14	627.56	477.16	405.44
19	532.89	639.14	475.4	403.61		**69**	513.67	640.13	486.61	393.71
20	524.48	629.23	466.25	357.69		**70**	535.34	624.27	483.48	382.03
21	517.56	645.53	484.94	397.72		**71**	536.95	646.45	480.29	370.29
22	519.93	633.67	485.07	394.76		**72**	531.77	636.9	469.05	382.09
23	514.77	646.63	467.72	402.17		**73**	534.72	629.43	491.62	400.7
24	532.04	631.06	486.51	404.53		**74**	531.69	636.26	470.96	381.36
25	515.79	630.91	492.21	400.63		**75**	508.84	649.55	492.78	362.65
26	519.59	635.62	485.78	401.4		**76**	540.28	637.06	468.99	390.55
27	537.43	629.63	474.45	376.98		**77**	544.66	630.69	470.21	387.05
28	540.09	633.74	484.45	384.58		**78**	520.8	644.05	487.85	375.78
29	530.1	626.31	489.17	405.04		**79**	517.8	625.43	488.54	402.08
30	509.99	649.28	488.28	392.58		**80**	544.85	637.09	479.49	371.2
31	520.19	631.13	469.11	370.28		**81**	536.95	631.79	482.03	402.19
32	527.83	631.1	483.94	401.61		**82**	517.84	646.16	486.79	384.56
33	531.14	633.84	466.65	375.82		**83**	537.93	624.03	471.47	371.77
34	547.87	639.52	494.58	390.05		**84**	530.5	623.97	488.29	370.16
35	536.26	633.39	484.83	381		**85**	523.02	633.32	492.23	370.63
36	523.66	649.06	474.32	383.48		**86**	543.19	633.23	483.32	379.18
37	531.47	630.97	484.29	397.77		**87**	525.46	635.19	466.26	371.29
38	514.23	641.16	469.88	394		**88**	547.06	627.88	480.72	389.58
39	517.94	648.75	490.4	400.49		**89**	537.17	642.36	476.13	379.3
40	526.45	630.08	486.77	383.31		**90**	537.63	642.44	468.87	383.91
41	518.69	630.36	492.9	362.2		**91**	537.61	640.13	473.2	384.94
42	510.32	648.08	490.5	363.53		**92**	530.58	636.46	469.98	383.96
43	521.44	631.62	477.5	392.42		**93**	510.82	643.78	488.77	400.78
44	541.38	639.06	467.18	367.54		**94**	534.86	642.9	475.14	404.7
45	529.31	625.48	475.62	370.35		**95**	544.69	634.97	492.01	360.76
46	527.47	624.92	477.12	396.28		**96**	523.84	647.68	482.22	395.01
47	513.27	625.89	484.16	383.19		**97**	520.77	651.15	490.74	382.41
48	526.61	636.11	486.45	366.27		**98**	539.11	638.29	495.31	367.81
49	509.31	649.42	494.27	373		**99**	523.5	634.6	473.66	372.66
50	511.43	628.85	475.22	404.68		**100**	513.43	627.43	490.83	397.79
Avg of 100 Attempts	**528.5204**	**636.5573**	**480.8449**	**385.8492**						

Case 4: Existing and DmRT follow Local -> Shared -> Ideal

2. Fragmentation for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	29.32	22.4	15.44	12.14		**51**	33.16	22.66	18.08	9.57
2	31.8	23.61	18.42	11.66		**52**	32.91	22.32	16.97	9.71
3	31.8	22.68	16.81	9.63		**53**	29.56	23.49	15.87	12.11
4	29.87	24.96	18.33	11.29		**54**	33.25	25.18	18.45	11.53
5	30.38	21.93	17.18	10.26		**55**	29.59	23.9	17.08	8.45
6	30.14	25.6	18.36	10.02		**56**	32.84	25.77	16.87	8.45
7	31.05	22.91	18.23	10.62		**57**	31.72	25.36	15.42	9.06
8	32.04	22.94	17.78	12.04		**58**	32.38	22.98	17.5	9.87
9	31.58	24.32	16.89	12.11		**59**	33.22	24.29	19.01	10.29
10	31.56	22.43	18.96	12.22		**60**	30.81	22.51	19.1	11.97
11	31.37	24.34	15.97	10.25		**61**	31.29	22.57	16.79	12.25
12	31.08	23.62	17.33	12.04		**62**	30.39	24.02	16.13	8.8
13	32.05	23.5	16.26	12.26		**63**	29.5	22.59	18.01	10.09
14	31.87	24.48	16.82	10.99		**64**	31.93	25.78	19.24	9.74
15	29.96	24.33	15.71	8.74		**65**	29.8	23.25	18.91	8.51
16	31.12	23.14	18.36	9.63		**66**	32.72	24.53	17.45	10.86
17	32.55	25.05	18.14	12.05		**67**	32.46	23.06	18.56	9.05
18	32.61	25.32	15.3	11.83		**68**	33.18	25.53	17.62	9.27
19	31.29	22.55	17.78	10.88		**69**	32.24	25.84	17.34	9.78
20	31.27	24.2	15.43	9.62		**70**	32.99	24.3	17.13	11.06
21	32.88	23.92	19.26	8.54		**71**	29.33	24.39	17.39	12.2
22	32.95	25.77	16.4	9.84		**72**	32.85	25.09	17.21	8.31
23	32.44	23.32	16.47	8.83		**73**	31.47	24.13	18.85	11.8
24	32.8	21.91	17.36	12.15		**74**	30.29	24.63	17.45	10.1
25	32.52	25.28	17.94	11.55		**75**	32.56	24.17	16.29	11.11
26	30.73	22.59	18.73	11.8		**76**	32.57	24.79	15.96	8.75
27	31.12	24.79	16.98	8.47		**77**	29.81	25.74	17.59	9.96
28	30.21	23.82	16.81	10.07		**78**	32.43	24.53	16.85	9.02
29	31.36	23.52	16.27	12.06		**79**	31.16	23.24	19.11	9.53
30	29.43	22.55	16.05	10.85		**80**	33.11	23.34	18.51	9.37
31	30.77	22.1	18.66	9.77		**81**	32.54	22.73	17.02	10.04
32	29.31	24.26	16.54	10.52		**82**	31.91	22.11	18.12	8.99
33	30.63	24.22	17.96	8.87		**83**	30.99	24.29	19.01	9.22
34	29.91	22.36	17.64	8.94		**84**	32.82	25.21	16.17	11.54
35	30.29	23.24	16.91	11.61		**85**	33.1	21.94	16.49	10.27
36	33	24.85	18.57	8.97		**86**	29.34	24.87	18.88	9.59
37	30.06	24.67	16.55	10.3		**87**	30.66	22.72	19.27	8.42
38	31.95	22.68	17.98	11.51		**88**	30.08	22.43	17.22	11.84
39	31.73	25.37	19.14	9.13		**89**	30.81	24.06	16.18	12.17
40	30.29	22.24	18.21	10.86		**90**	32.13	24.82	17.61	11.51
41	31.53	23.51	19.29	10.04		**91**	32.84	23.37	19.3	10.77
42	29.38	22	17.2	11.87		**92**	32.28	24.8	17.13	10.89
43	31.01	23.78	18.01	11.2		**93**	33.21	25.77	18.75	10.25
44	33.14	23.67	18.59	11.94		**94**	30.82	23.2	16.18	9.65
45	30.21	24.64	17.2	11		**95**	32.26	23.72	15.92	10.52
46	32.94	25.33	18.24	8.97		**96**	30.84	25.47	17.75	10.07
47	32.69	22.91	18.3	11.68		**97**	31.98	24.58	18.95	10.4
48	30.95	25.21	18.89	12.21		**98**	32.69	23.76	17.9	11.07
49	31.84	25.61	15.75	11.54		**99**	32.17	22.25	16.47	9.27
50	31.3	24.94	19.19	9.63		**100**	30.41	22.19	17.91	9.14
Avg of 100 Attempts	**31.4948**	**23.8764**	**17.5356**	**10.4119**						

Case 4: Existing and DmRT follow Local -> Shared -> Ideal

3. No. of Request Satisfied for all allocators 100 attempts

(Best Case i.e. for 100 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	70	79	87	96		**51**	73	78	85	92
2	71	77	86	93		**52**	70	78	87	95
3	73	78	86	96		**53**	70	78	86	93
4	72	76	86	95		**54**	71	79	89	94
5	71	77	85	93		**55**	71	80	87	95
6	72	78	89	95		**56**	72	77	87	96
7	74	77	85	92		**57**	72	77	89	94
8	72	77	86	93		**58**	72	79	89	93
9	70	80	86	94		**59**	72	79	85	95
10	73	79	89	92		**60**	74	80	87	92
11	71	77	85	93		**61**	71	78	89	92
12	72	77	89	96		**62**	70	77	85	93
13	71	79	88	94		**63**	73	80	85	93
14	72	77	85	91		**64**	72	80	88	96
15	72	78	88	93		**65**	70	80	89	95
16	73	80	89	95		**66**	71	77	88	93
17	71	80	86	93		**67**	73	80	88	94
18	72	77	88	96		**68**	72	79	85	96
19	71	78	89	96		**69**	71	79	88	93
20	71	77	86	94		**70**	73	77	88	94
21	71	78	85	94		**71**	73	78	85	96
22	74	77	85	93		**72**	71	80	88	95
23	72	78	87	93		**73**	70	80	87	91
24	72	78	87	91		**74**	73	77	84	94
25	73	79	84	96		**75**	73	79	87	95
26	73	78	89	96		**76**	72	79	86	92
27	74	79	88	94		**77**	70	77	85	94
28	70	80	88	95		**78**	70	79	88	93
29	73	79	85	96		**79**	71	80	87	92
30	73	77	88	94		**80**	71	79	88	91
31	73	78	87	94		**81**	72	76	86	92
32	72	76	87	92		**82**	70	80	87	95
33	71	78	88	92		**83**	73	78	87	92
34	74	80	84	94		**84**	70	78	88	93
35	73	80	87	92		**85**	70	80	89	94
36	70	77	86	95		**86**	72	77	85	94
37	73	77	87	94		**87**	70	77	88	92
38	72	77	85	94		**88**	74	79	87	92
39	73	77	84	92		**89**	71	78	87	96
40	70	80	89	94		**90**	73	77	89	95
41	73	77	85	93		**91**	72	77	88	93
42	72	79	88	93		**92**	71	80	88	94
43	73	76	89	95		**93**	73	79	86	94
44	71	78	86	94		**94**	70	76	88	96
45	72	76	87	92		**95**	73	76	88	92
46	70	77	86	93		**96**	70	76	89	94
47	72	76	85	93		**97**	72	79	87	95
48	71	80	88	95		**98**	72	79	88	94
49	73	80	85	93		**99**	71	77	87	95
50	72	80	85	94		**100**	72	80	84	94
Avg of 100 Attempts	**71.7431**	**78.1612**	**86.8777**	**93.7361**						

Case 4: Existing and DmRT follow Local -> Shared -> Ideal

1. Execution Time for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	2961.45	4135.01	2583.1	2331.78		**51**	2887.63	4163.1	2467.59	2174.54
2	2889.99	4154.99	2465.44	2299.77		**52**	2930.68	4182.98	2569.76	2352.54
3	2913.53	4203.54	2514.8	2335.41		**53**	2939.24	4141.21	2641.43	2295.25
4	2882.96	4115.25	2487.22	2289.62		**54**	2959.1	4153.78	2561.58	2308.57
5	2913.04	4125.67	2472.92	2218.57		**55**	2954.71	4174.39	2495.43	2289.16
6	2898.94	4210.49	2633.92	2302.17		**56**	2941.5	4179.98	2619.2	2161.35
7	2937.71	4161.86	2458.21	2318.37		**57**	2959.79	4150.84	2608.18	2304.92
8	2886.7	4138.92	2563.45	2250.51		**58**	2900.4	4194.15	2590.86	2213.18
9	2934.8	4153.37	2627.62	2345.19		**59**	2966.78	4185.15	2541.32	2168.88
10	2906.78	4168.5	2495.26	2210.59		**60**	2897.23	4165.26	2556.41	2258.5
11	2956	4144.37	2624.96	2293.06		**61**	2914	4148.59	2494.9	2315.91
12	2934.78	4183.37	2475.86	2164.7		**62**	2898.97	4196.36	2495.7	2183.77
13	2937.94	4140.56	2646.56	2289.34		**63**	2915.03	4189.14	2636.62	2232.08
14	2954.86	4136.36	2551.02	2215.26		**64**	2889.92	4191.11	2487.44	2225.15
15	2914.19	4159.54	2450.07	2342.94		**65**	2925.12	4125.93	2569.39	2338.21
16	2937.16	4176.98	2496.73	2272.53		**66**	2895.24	4212.57	2642.45	2207.25
17	2968.76	4131.19	2479.18	2254.62		**67**	2931.25	4180.28	2495.32	2179.58
18	2885.66	4199.61	2481.72	2220.33		**68**	2896.26	4167.46	2621.6	2157.75
19	2901.84	4177.22	2495.87	2202.42		**69**	2909.43	4138.78	2615.47	2286.72
20	2895.94	4153.11	2527.54	2188.32		**70**	2890.85	4157.15	2477.53	2295.4
21	2934.01	4177.1	2474.35	2182.76		**71**	2950.57	4161.83	2524.1	2296.89
22	2888.52	4129.06	2493.09	2208.18		**72**	2896.89	4156.71	2492.18	2185.07
23	2920.9	4202.05	2581.7	2196.49		**73**	2944.98	4188.13	2647.51	2300.35
24	2935.32	4114	2472.79	2334.14		**74**	2960.23	4178.97	2495.97	2239.77
25	2889.99	4143.28	2592.28	2231.4		**75**	2906.86	4148.38	2640.72	2326.99
26	2899.3	4125.24	2458.35	2309.09		**76**	2890.6	4208.92	2473.19	2185.96
27	2902.29	4194.87	2620.66	2254.32		**77**	2935.61	4148.12	2506.59	2325.16
28	2928.37	4145.61	2496.1	2224.87		**78**	2979.13	4204.71	2459.38	2307.7
29	2980.67	4116.61	2466.42	2160.13		**79**	2962.08	4176.23	2456.34	2228.87
30	2946.1	4162.41	2479.42	2276.1		**80**	2977.86	4198.88	2622.14	2203.27
31	2899.5	4192.49	2614.84	2175.53		**81**	2941.52	4213.06	2511.95	2235.99
32	2974.41	4133.71	2540.81	2210.05		**82**	2974.96	4152.72	2571.83	2338.76
33	2904.94	4161.68	2586.65	2256.11		**83**	2953.82	4208.11	2572.71	2209.15
34	2974.57	4182.19	2535.72	2169.14		**84**	2973.16	4196.87	2644.27	2262.08
35	2897.05	4155.72	2485.95	2284.79		**85**	2902.25	4182.53	2527.61	2209.83
36	2898.37	4174.92	2538.42	2166.94		**86**	2943.03	4196.76	2474.51	2195.47
37	2945.84	4117.74	2546.47	2276.62		**87**	2971.45	4134.51	2499.08	2296.32
38	2897.72	4156.42	2567.64	2331.04		**88**	2961.04	4211.51	2555.56	2278.57
39	2944.09	4145.33	2487.51	2338.44		**89**	2932.58	4162.93	2505.46	2233.49
40	2896.82	4188.25	2592.11	2234.26		**90**	2922.76	4162.65	2529.16	2195.34
41	2951.59	4190.69	2574.51	2206.84		**91**	2901.67	4144.21	2517.37	2224.91
42	2945.89	4144.44	2595.76	2337.79		**92**	2943.36	4206.59	2464.84	2294.68
43	2887.27	4129.62	2534.75	2248.86		**93**	2920.04	4173.1	2638.2	2194.31
44	2937.12	4188.54	2553.67	2175.45		**94**	2910.06	4154.79	2486.61	2222.01
45	2974.88	4180.92	2630.9	2230.14		**95**	2975.84	4192.28	2611.58	2283.28
46	2892	4135.78	2484.27	2308.94		**96**	2953.12	4174.58	2622.96	2342.64
47	2888.98	4130.62	2578.67	2261.04		**97**	2962.73	4193.44	2467.71	2330.84
48	2893.07	4178.7	2625.14	2214.3		**98**	2932.7	4198.18	2510.2	2281.54
49	2982.21	4206.46	2590.37	2186.95		**99**	2944.55	4139.52	2601.13	2252.52
50	2893.93	4135.25	2475.09	2246.07		**100**	2956.07	4166.89	2526.8	2189.1
Avg of 100 Attempts	**2928.034**	**4166.439**	**2541.517**	**2252.019**						

Case 4: Existing and DmRT follow Local -> Shared -> Ideal

2. Fragmentation for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	40.47	31.63	22.18	16.62		**51**	40.79	30.45	20.89	15.32
2	38.76	31.77	21.7	13.87		**52**	37.66	30.8	23.73	15.41
3	41.11	30.41	21.53	14.43		**53**	38.2	31.76	23.61	16.4
4	39.65	32.27	21.89	16.57		**54**	37.17	33.27	21.73	15.55
5	37.48	31.6	21.9	13.78		**55**	41.31	32.01	22.8	16.09
6	40.35	30.31	21.32	13.83		**56**	40.67	31.79	23.68	13.45
7	37.46	32.1	24.18	13.47		**57**	37.79	31.09	21.32	13.61
8	39.86	32.25	24.22	13.28		**58**	39.22	33.69	23.32	13.77
9	40.16	31.83	21.74	15.63		**59**	38.37	31.27	21.92	14.81
10	39.83	33.03	24.27	14.32		**60**	37.31	32.46	24	15.79
11	38.8	30.21	22.6	16.16		**61**	40.15	33.64	24.27	15.94
12	37.51	32.41	23.13	14.03		**62**	37.96	30.36	24.78	14.2
13	40.48	32.33	21	16.01		**63**	37.17	31.61	24.75	13.75
14	37.11	31.44	23.02	13.73		**64**	36.63	31.01	21.69	15.7
15	40.56	32.71	23.62	12.99		**65**	37.23	32.59	23.05	14.35
16	36.78	31.58	22.33	13.94		**66**	38.79	32.7	21.42	13.39
17	37.99	30.35	24.28	15.78		**67**	40.25	33.63	23.02	15.1
18	37.54	32.29	22.66	13.6		**68**	36.69	30.55	22.15	15.16
19	38.33	31.57	24.66	13.8		**69**	38.57	30.44	22.11	14.35
20	37.16	31.25	20.88	13.06		**70**	36.51	29.92	23.05	15.63
21	39.18	30.11	20.97	16.29		**71**	37.62	30.76	24.54	13.77
22	39.35	31.34	21.24	16.06		**72**	37.25	33.38	23.38	13.15
23	41.38	30.78	24.45	13.65		**73**	37.62	29.86	22.39	13.36
24	39.03	32.39	24.71	15.6		**74**	37.48	32.4	21.71	14.4
25	40.07	30	21.64	16.02		**75**	38.8	31.59	23.42	15.85
26	38.29	29.94	22.66	16.42		**76**	41.15	30.15	23.38	13.48
27	37.25	29.91	22.24	14.49		**77**	38.85	32.47	23.54	14.85
28	40.39	31.79	23.52	13.27		**78**	39.69	30.06	23.28	15.52
29	39.58	29.9	24.49	16.84		**79**	40.34	30.79	24.33	16.74
30	41.16	32.6	21.22	15.46		**80**	39.91	33.56	23.65	16.48
31	37.62	31.33	23.11	16.66		**81**	38.67	32.92	22.55	15.09
32	38.35	32.82	22.41	16.64		**82**	39.19	31.42	22.34	16.18
33	38.51	30.45	22.51	16.56		**83**	38.21	31.91	24.68	16.16
34	39.91	31.87	23.95	16.4		**84**	37.07	31.16	21.47	15.41
35	37.65	32.4	24.16	16.81		**85**	40.93	30.42	22.19	15.21
36	39.95	32.3	23.49	14.69		**86**	39.73	30.95	24.64	16.48
37	41.39	32.69	23.38	15.02		**87**	40.69	30.94	24.18	14.75
38	38.3	32.82	22.28	14.55		**88**	37.12	31.69	24.68	16.45
39	40.66	30.99	21.17	15.29		**89**	41.03	32.73	21.94	13.68
40	36.68	30.56	23.48	16.46		**90**	40.72	29.71	24.13	13.6
41	38.08	30.48	23.63	15		**91**	40.17	31.46	24.35	16.67
42	40.34	31.65	20.88	14.28		**92**	36.73	32.45	24.29	14.47
43	40.51	30.68	20.87	13.6		**93**	37.61	31.06	22.19	15.3
44	39.06	33.56	21.85	16.06		**94**	36.88	31.84	21.03	15.57
45	38.26	33.5	22.9	15.36		**95**	38.43	31.55	23.5	15.67
46	40.33	33.17	23.36	14.59		**96**	39.03	32.76	24.72	13.49
47	39.3	30.07	22.84	13.73		**97**	40.6	32.92	24.03	16.44
48	36.76	31.27	23.68	15.52		**98**	40.78	31.41	23.33	14.62
49	40.32	33.09	20.83	15.19		**99**	38.04	31.48	22.49	15.07
50	36.91	32.26	23.73	13.85		**100**	38.76	31.65	22.9	16.17
Avg of 100 Attempts	**38.895**	**31.6255**	**22.913**	**15.0111**						

Case 4: Existing and DmRT follow Local -> Shared -> Ideal

3. No. of Request Satisfied for all allocators 100 attempts

(Average Case i.e. for 1000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	63.7	72.0	78.8	85.6		**51**	63.7	71.2	79.4	86.9
2	61.4	71.2	80.3	89.3		**52**	60.4	71.1	78.5	86.8
3	61.8	72.1	80.1	87.4		**53**	63.3	71.5	78.7	85.6
4	62.3	71.2	80.9	86.1		**54**	61.1	71.1	78.0	88.9
5	60.2	68.9	78.5	87.7		**55**	63.2	69.7	79.6	86.5
6	64.1	72.6	81.1	89.9		**56**	60.4	70.9	78.7	87.5
7	61.8	71.5	81.5	86.1		**57**	63.4	70.8	77.8	90.1
8	61.1	70.4	81.8	87.3		**58**	60.6	68.9	82.3	89.6
9	61.1	71.5	81.3	89.8		**59**	61.5	70.1	81.5	85.6
10	63.0	69.7	81.1	86.7		**60**	60.2	69.9	81.9	87.7
11	61.6	71.4	81.0	87.7		**61**	60.4	71.2	78.4	85.4
12	60.6	69.6	81.6	87.8		**62**	62.0	71.1	79.7	87.5
13	64.0	69.0	80.5	88.9		**63**	64.1	69.2	81.0	88.5
14	62.6	69.8	79.8	86.7		**64**	60.8	68.9	81.8	89.3
15	60.3	71.1	77.3	88.0		**65**	63.4	70.4	79.7	85.3
16	63.4	70.8	80.7	88.2		**66**	61.2	72.2	80.9	85.5
17	64.0	70.3	80.0	89.3		**67**	60.2	68.9	79.4	86.9
18	60.7	69.2	78.4	89.8		**68**	61.4	71.8	77.9	87.0
19	61.5	70.4	79.1	87.4		**69**	64.1	69.1	79.7	86.2
20	63.0	69.4	82.3	87.2		**70**	61.3	70.5	82.0	85.7
21	61.8	71.0	77.6	87.1		**71**	60.5	69.0	79.9	89.3
22	61.3	70.1	78.9	86.1		**72**	60.6	72.4	81.5	85.7
23	61.4	68.7	79.9	88.6		**73**	61.2	70.1	82.2	85.5
24	61.3	68.8	81.5	87.1		**74**	60.5	70.0	81.5	90.1
25	63.8	70.0	81.9	86.0		**75**	63.7	68.8	81.6	87.0
26	61.8	71.1	79.5	89.4		**76**	62.1	69.8	77.5	88.8
27	64.1	69.4	80.9	85.4		**77**	60.2	71.9	77.4	88.5
28	61.1	72.5	81.1	85.5		**78**	61.8	69.3	79.0	88.8
29	62.4	68.8	80.3	89.3		**79**	60.8	72.5	78.3	86.9
30	62.9	71.9	82.1	86.6		**80**	61.7	71.8	82.1	88.8
31	64.1	70.7	78.0	89.7		**81**	62.3	71.6	81.8	85.8
32	62.5	68.9	78.1	89.0		**82**	64.2	71.7	78.3	85.8
33	61.7	71.7	81.1	85.4		**83**	61.8	70.6	77.8	86.6
34	61.6	72.3	79.6	88.3		**84**	63.8	71.4	80.3	90.3
35	63.4	70.1	77.4	88.5		**85**	60.4	72.3	80.0	86.9
36	63.4	69.3	80.7	89.2		**86**	60.9	71.0	80.0	86.5
37	64.1	71.6	79.6	86.1		**87**	60.5	70.9	79.0	86.8
38	63.8	72.2	77.9	87.2		**88**	61.7	68.9	78.5	89.3
39	61.1	71.4	78.0	89.8		**89**	60.3	68.8	79.7	85.9
40	63.5	71.6	77.4	88.5		**90**	63.8	69.2	78.9	86.0
41	64.2	71.8	81.4	86.0		**91**	63.5	72.1	82.0	89.1
42	60.4	70.8	78.8	88.3		**92**	62.5	72.0	81.8	88.0
43	61.4	72.4	82.0	89.8		**93**	60.9	72.4	78.8	86.5
44	62.1	68.8	78.0	86.7		**94**	63.9	71.0	80.2	85.5
45	61.1	72.2	77.8	86.4		**95**	60.6	69.7	81.0	86.6
46	63.5	69.5	79.7	86.6		**96**	64.1	72.6	80.1	86.2
47	63.6	70.6	82.2	88.2		**97**	61.7	71.7	79.8	89.7
48	60.6	71.5	79.3	88.5		**98**	61.8	71.5	79.1	89.8
49	62.0	71.1	81.7	86.1		**99**	62.7	71.3	78.2	87.0
50	60.7	69.0	79.9	85.6		**100**	60.2	70.8	80.4	89.4
Avg of 100 Attempts	**62.0389**	**70.6678**	**79.9134**	**87.5092**						

Case 4: Existing and DmRT follow Local -> Shared -> Ideal

1. Execution Time for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	4232.74	5117.09	3680.29	3405.7		**51**	4229.8	5099.91	3670.06	3457.63
2	4181.59	5125.08	3657.81	3324.56		**52**	4229.32	5146.33	3731.19	3268.49
3	4220.87	5073.53	3679.32	3463.85		**53**	4187.97	5148.85	3663.21	3301.11
4	4199.61	5057.61	3665.91	3419.66		**54**	4209.36	5137.96	3656.52	3357.87
5	4242.11	5116.67	3647.92	3282.3		**55**	4264.7	5152.28	3687.14	3398.56
6	4189.44	5086.62	3665.97	3423.57		**56**	4257.23	5094.73	3642.52	3405.23
7	4251.88	5142.29	3681.81	3362.05		**57**	4231.28	5063.76	3704.74	3461.42
8	4203.87	5076.19	3719.67	3301.84		**58**	4216.26	5132.15	3663.71	3278.1
9	4230.35	5056.65	3720.06	3410.67		**59**	4224.88	5118.24	3678.59	3405.84
10	4236.27	5120.76	3670.91	3338.49		**60**	4258.67	5138.7	3669.01	3405.59
11	4204.89	5110.94	3717.61	3358.73		**61**	4206.96	5099.21	3648.92	3284.78
12	4261.7	5145.27	3698.36	3432.28		**62**	4191.36	5127.93	3698.02	3341.53
13	4278.28	5116.07	3714.68	3346.05		**63**	4262.57	5150.18	3660.87	3398.6
14	4271.28	5065.95	3641.32	3463.1		**64**	4193.68	5056.93	3713.92	3447.74
15	4272.41	5118.87	3732.49	3292.61		**65**	4244.09	5091.24	3661.98	3297.76
16	4279.31	5086.9	3651.06	3391.69		**66**	4229.08	5151.4	3665.04	3274.45
17	4262.04	5119.14	3666.23	3274.02		**67**	4241.62	5137.74	3639.4	3319.88
18	4225.45	5086.35	3694.73	3292		**68**	4244.49	5102.98	3656.59	3397.52
19	4268.08	5139.95	3709.34	3294.67		**69**	4262.23	5109.69	3664.93	3308.93
20	4200.22	5089.66	3707.1	3446.14		**70**	4233	5060.25	3689.11	3424.45
21	4273.61	5130.92	3663.88	3285.19		**71**	4265.1	5126.15	3735.1	3282.39
22	4181.77	5117.42	3704.02	3360.08		**72**	4181.62	5122.34	3724.57	3298.78
23	4239.51	5144.32	3656.88	3298.78		**73**	4276.54	5085.05	3676.82	3402.1
24	4224.66	5095.55	3709.63	3440.84		**74**	4265.49	5142.72	3640.49	3328.06
25	4239.86	5126.24	3668.95	3408.07		**75**	4272.81	5102.6	3723.01	3369.03
26	4223.69	5055.53	3680.54	3464.77		**76**	4208.67	5154.05	3701.51	3342.14
27	4228.43	5098.5	3651.85	3392.78		**77**	4182.51	5099.87	3677.55	3377.21
28	4267.32	5116.76	3720.07	3462.38		**78**	4229.98	5090.19	3682.08	3421.09
29	4213.49	5121.84	3662.31	3429.7		**79**	4246.72	5146.36	3723.52	3333.71
30	4182.7	5111.84	3637.68	3457.02		**80**	4193.56	5085.75	3652.28	3341.91
31	4256.96	5078.24	3712.67	3427.52		**81**	4198.22	5109.18	3660.65	3333.87
32	4217.41	5101.53	3640.86	3313.02		**82**	4221.67	5118.04	3662.16	3450.11
33	4220.49	5090.84	3687.5	3266.25		**83**	4225.99	5078.83	3645.36	3308.41
34	4185.54	5135.94	3722.04	3348.14		**84**	4227.09	5129.71	3710	3452.91
35	4260.37	5155.13	3651.97	3429.02		**85**	4279.09	5091.72	3723.38	3452.02
36	4195.93	5151.96	3672.5	3276.07		**86**	4218.42	5089.36	3651.83	3328.93
37	4271.51	5150.03	3732.13	3461.66		**87**	4199.35	5101.01	3643.3	3396.24
38	4277.83	5102.44	3705.19	3312.26		**88**	4260.2	5120.43	3688.69	3313.71
39	4186.76	5136.92	3673.34	3445.45		**89**	4188.83	5106.17	3652.42	3358.07
40	4249.39	5131.69	3665.02	3378.12		**90**	4214.22	5078.6	3678.53	3387.53
41	4242.62	5142	3675.93	3314.47		**91**	4275.2	5061.25	3694.67	3281.34
42	4194.54	5093.05	3717.78	3416.37		**92**	4207.14	5070.99	3697.53	3310.22
43	4180.25	5151.48	3707.03	3454.24		**93**	4257.6	5090.74	3722.34	3306.34
44	4248.28	5140.56	3712.81	3331.01		**94**	4247.3	5072.16	3655.5	3409.26
45	4228.28	5069.82	3653.4	3422.73		**95**	4213.38	5083.52	3722.9	3281.58
46	4220.3	5058.76	3705.69	3430.46		**96**	4192.57	5062.39	3736.7	3380.43
47	4249.25	5125.39	3643.59	3462.71		**97**	4260.51	5123.91	3730.57	3280.82
48	4230.05	5097.34	3707.54	3365.21		**98**	4275.94	5116.27	3724.59	3338.32
49	4245.49	5106.78	3731.02	3276.85		**99**	4221.53	5073.87	3701.04	3416.15
50	4213.58	5057.47	3652.56	3363.18		**100**	4267.39	5061.95	3695.96	3433.74
Avg of 100 Attempts	**4231.555**	**5107.635**	**3684.495**	**3367.702**						

Case 4: Existing and DmRT follow Local -> Shared -> Ideal

2. Fragmentation for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	47.32	37.38	31.82	20.85		**51**	48.82	37.65	30.44	20.35
2	48.98	36.99	32.48	17.98		**52**	46.22	37.09	30.7	17.58
3	47.55	38.6	30.16	20.68		**53**	47.14	39.41	31.76	17.77
4	47.63	39.17	29.38	18.69		**54**	47.59	38.28	32.53	17.29
5	49.08	37.5	29.53	18.59		**55**	47.27	38.84	32.32	20.25
6	46.9	37.09	30.09	18.4		**56**	45.9	37.2	31.62	20.95
7	47.29	37.37	30.76	17.55		**57**	48.47	38.38	29.68	18.25
8	45.84	39.23	29.2	19.52		**58**	47.1	39.54	30.81	17.45
9	47.26	40.02	32.23	20.93		**59**	47.06	39.64	29.9	19.82
10	46.32	36.15	29.24	19.39		**60**	48.78	38.68	29.1	17.16
11	48.63	38.03	32.13	17.83		**61**	45.74	39.81	31.92	20.14
12	49.08	39.02	30.71	18.09		**62**	46.23	38.35	30.45	18.84
13	46.92	37.17	31.09	17.22		**63**	46.07	37.32	30.95	18.92
14	45.74	38.23	28.92	18		**64**	48.35	37.45	30.86	20.68
15	48.03	39.39	30.93	20.13		**65**	45.51	39.6	31.99	19.95
16	49.2	38.07	29.89	18.52		**66**	47.72	38.15	32.37	17.91
17	47.19	36.35	29.65	19.42		**67**	49.01	37.72	32.55	18.47
18	46.78	38.42	31.33	17.18		**68**	46.25	39.66	31.96	17.3
19	47.51	38.77	29.22	18.58		**69**	47.94	39.2	32.41	20.32
20	46.46	39.72	31.25	20.24		**70**	48.48	37.52	30.57	19.55
21	49.38	37.83	29.82	19.09		**71**	49.15	38.44	31.21	17.4
22	49.36	38.01	31	20.33		**72**	46.18	36.71	31.29	18.94
23	46.1	39.46	30.94	19.27		**73**	45.56	37.78	29.09	17.69
24	48.73	36.62	30.08	20.67		**74**	47.07	36.72	32.42	19.91
25	48.17	37.83	31.36	17.5		**75**	45.91	38.02	30.96	20.06
26	45.72	36.28	29.24	17.87		**76**	48.01	37.39	30.42	19.72
27	46.81	38.52	30.77	20.66		**77**	45.6	37.05	30.79	17.12
28	45.47	39.15	31.53	20.72		**78**	47.04	38.3	29.83	17.62
29	45.74	38.47	31.86	20.2		**79**	48.59	39.21	29.22	18.42
30	45.91	38.31	29.14	18.57		**80**	46.51	39.51	32.04	20.65
31	49.35	37.98	32.78	17.61		**81**	47.55	38.74	31.33	19.15
32	47.98	37.21	30.99	20.53		**82**	47.35	39.7	31.34	20.62
33	46.96	36.5	31.38	19		**83**	48.54	37.3	31.5	20.63
34	47.72	38.61	30.23	20.85		**84**	45.65	39.32	31.07	20.98
35	49.33	39.67	29.17	20.22		**85**	47.38	38.37	31.3	18.34
36	48.48	38.56	29.71	20.21		**86**	46.78	38.06	31.66	17.86
37	46.67	39.38	30.23	18.6		**87**	47.04	38.72	31.22	20.4
38	46.69	36.56	29.72	20.92		**88**	49.29	39.39	29	17.4
39	48.78	38.8	29.71	18.26		**89**	48.7	39.24	31.99	17.87
40	48.16	36.7	32.35	20.19		**90**	47.54	39.98	31.03	17.86
41	46.48	37.52	31.77	17.78		**91**	48.29	37.59	32.71	19.12
42	47.55	38.81	31	20.04		**92**	46.43	38.2	29.42	19.75
43	45.67	37.57	29.74	21.01		**93**	45.98	39.42	30.53	18.15
44	48.11	38.11	31.64	20.3		**94**	45.43	39.23	31.11	18.33
45	46.9	37.33	31.22	18.2		**95**	46.52	37.02	31.59	19.14
46	48.81	38.89	29.89	18.35		**96**	48.8	37.67	31.74	20.93
47	48.78	40.02	30.66	20.93		**97**	47.18	38.41	31.05	18.93
48	48.61	36.54	29.25	19.05		**98**	47.7	39.4	31.19	21.09
49	47.79	36.42	29.5	18.45		**99**	46.67	40.02	32.84	17.22
50	48	39.31	29.72	20.21		**100**	46.34	37.94	32.53	19.51
Avg of 100 Attempts	**47.3835**	**38.2598**	**30.8472**	**19.1314**						

Case 4: Existing and DmRT follow Local -> Shared -> Ideal

3. No. of Request Satisfied for all allocators 100 attempts

(Worst Case i.e. for 2000 block requests)

Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT		Attempt No.	Dlmalloc	tcmalloc	TLSF	DmRT
1	49.20	59.00	73.65	82.30		**51**	49.05	61.00	72.10	82.40
2	49.55	60.85	72.30	83.35		**52**	49.05	58.95	73.65	83.60
3	51.55	58.40	72.35	81.80		**53**	50.20	61.30	72.25	81.45
4	49.90	58.50	74.00	81.85		**54**	51.40	59.65	74.65	81.40
5	52.15	58.75	73.00	82.50		**55**	50.40	58.75	75.80	82.50
6	49.05	58.55	75.50	82.70		**56**	51.70	61.65	74.60	83.70
7	49.20	59.85	75.00	83.90		**57**	49.35	61.65	72.35	83.90
8	50.85	61.65	73.25	82.45		**58**	49.30	58.65	72.20	83.10
9	52.20	60.00	74.90	81.20		**59**	49.10	61.00	74.55	83.55
10	50.00	61.00	74.00	83.95		**60**	52.45	59.00	72.80	81.85
11	50.60	58.45	74.70	84.00		**61**	51.80	59.55	74.40	81.25
12	52.60	59.85	74.55	81.05		**62**	49.60	60.90	74.50	82.15
13	51.00	61.00	75.20	81.00		**63**	50.55	59.80	74.60	82.45
14	49.10	59.15	73.35	83.95		**64**	51.00	60.60	72.80	82.50
15	52.75	60.10	72.45	83.60		**65**	50.90	61.85	73.25	83.50
16	51.95	58.45	74.70	82.20		**66**	52.05	60.10	75.85	81.85
17	49.80	58.70	74.25	81.60		**67**	50.25	58.35	73.55	83.40
18	49.25	60.70	75.15	83.10		**68**	49.45	61.75	74.85	82.25
19	49.40	58.45	75.10	82.15		**69**	52.80	58.10	75.75	81.00
20	51.75	58.25	75.05	83.45		**70**	49.80	60.75	75.10	82.30
21	52.75	58.95	72.10	82.40		**71**	52.70	59.55	72.55	81.05
22	51.20	61.35	75.25	83.15		**72**	49.60	60.85	72.10	82.10
23	52.65	60.55	75.15	83.20		**73**	49.60	59.40	74.35	81.70
24	52.50	61.75	72.25	83.90		**74**	50.10	59.95	74.40	83.10
25	52.75	60.30	75.45	83.75		**75**	50.25	60.55	75.05	82.15
26	49.65	60.05	74.45	83.40		**76**	50.40	61.55	73.60	81.55
27	50.40	60.20	72.00	82.25		**77**	51.95	61.65	74.30	82.35
28	49.95	58.40	75.50	81.60		**78**	50.20	58.25	74.70	81.70
29	50.55	59.95	72.15	83.30		**79**	51.45	61.85	74.10	81.30
30	50.80	60.30	76.00	83.60		**80**	49.10	61.00	72.30	81.55
31	51.15	59.45	73.90	81.55		**81**	49.10	60.75	72.95	83.20
32	50.80	58.25	72.00	84.00		**82**	49.80	59.25	73.30	83.55
33	51.90	59.90	75.00	82.00		**83**	51.15	61.75	72.35	83.95
34	52.95	59.80	74.25	83.85		**84**	52.05	59.75	75.40	81.85
35	49.65	60.10	74.00	82.65		**85**	50.45	60.55	74.40	83.55
36	49.15	60.80	73.25	82.70		**86**	50.65	58.25	75.45	81.85
37	49.15	60.95	75.80	83.05		**87**	52.65	62.00	72.55	84.00
38	52.20	60.00	72.75	82.55		**88**	51.95	61.95	75.05	81.65
39	51.50	58.30	73.25	83.25		**89**	51.00	59.90	72.55	82.70
40	50.95	60.35	73.50	83.35		**90**	51.20	59.35	75.15	83.10
41	49.60	61.50	74.00	82.15		**91**	51.65	58.00	73.80	82.75
42	49.10	61.35	72.25	81.85		**92**	49.65	58.85	72.70	83.95
43	51.95	59.60	75.35	81.55		**93**	50.60	61.50	73.05	83.70
44	51.40	60.10	74.50	81.15		**94**	51.50	60.25	72.55	84.00
45	52.95	59.90	72.45	82.50		**95**	52.55	58.10	72.20	83.30
46	52.05	59.60	75.55	83.05		**96**	51.00	58.40	73.85	83.10
47	50.45	59.25	72.90	83.50		**97**	52.45	58.95	72.60	82.45
48	49.70	62.00	74.55	82.00		**98**	51.15	61.15	73.55	84.00
49	49.75	59.45	72.15	82.35		**99**	49.45	60.50	75.90	82.40
50	51.85	59.20	72.75	83.00		**100**	52.15	61.00	75.00	83.80
Avg of 100 Attempts	**50.8095**	**59.9945**	**73.883**	**82.662**						

Memory Management in Real-Time Operating System

A Synopsis

Submitted in partial fulfillment of the

requirements for the award of the degree

of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE & ENGINEERING

By

SHAH VATSALKUMAR HASMUKHBHAI

FOTE/878

Guided By,

Dr. APURVA SHAH

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

FACULTY OF TECHNOLOGY & ENGINEERING

THE MAHARAJA SAYAJIRAO UNIVERSITY OF BARODA

VADODARA-390002 (INDIA)

JUNE 2018

ABSTRACT

The memory allocation algorithms have been analyzed and worked upon broadly, but there is less attention given to the multiprocessor architecture and real-time operating system as well. Most of the algorithms are applicable for the general-purpose operating system and do not fulfill the necessities of real-time systems. Moreover, limited allocators designed to support real-time systems which are not completely scalable for multiprocessors. In the 21st century, as we have entered into an era of high-performance computing, the demand for multi-core architecture has gained momentum. NUMA architecture based systems are the outcome of this tendency and offer an organized scalable design. However, existing dynamic memory allocators are not capable of performing on a multiprocessor architecture and do not comply with real-time system requirements as well. Researches have proved that the existing memory allocators for any operating systems which support NUMA architecture are not suitable for real-time applications. Hence, there is a need to have a dynamic memory allocator which can perform well on SMP and NUMA based soft real-time systems, with better execution time and less fragmentation.

This research is carried out in the same direction to achieve the aforementioned goal of a dynamic memory allocator for real-time systems. 1. Dynamic memory allocator ***DmRT*** *for symmetric multiprocessing (SMP) and Non-Uniform Memory Access (NUMA) architecture based real-time operating system has been designed and implemented which provides consistent and optimum execution time, less memory fragmentation, as well as satisfying a maximum number of the memory request, compared to other existing allocators. 2. There are so many simulators available to simulate different test cases for scheduling in a real-time operating system like Litmus-RT, Mark3, rtsim, etc., but till date, none of the simulators is available for simulating memory management algorithm for RTOS. Hence,* ***MemSimRT*** *has been designed to simulate various memory allocators for both SMP as well as NUMA architecture based RTOS.*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

1.1 Introduction to Real-Time Operating System

RTOS denotes "Real-time Operating System" which is basically a type of an operating system which provides support to the real-time applications by giving an accurate result within the time limit [4][17]. Real-time Operating System can be mainly classified into two categories: 1) hard real-time system and 2) soft real-time system depends on how rigorously it follows the task accomplishment deadline.

Table 1.1: Difference between General Purpose OS and RTOS [4]

	RTOS	**General Purpose OS**
Determinism	Deterministic	Non-deterministic
Preemptive kernel	All kernel operations are preemptable	Not Necessary
Priority Inversion	Have mechanisms to prevent priority inversion	No such mechanism is present
Task Scheduling	Scheduling is time-based	Scheduling is process based
Latency	Have their worst-case latency defined	Latency is not of a concern Purpose OS
Application	Typically used for embedded applications	General purpose OS is used for desktop PCs or other general purpose PCs

A real-time system can be categorized into three different categories on the basis of its criticality [17]:

- **Hard:** A real-time task/system is considered to be hard if generating the outcomes after its deadline may create terrible significances on the system under control. For example,

automotive systems, and nuclear-plant governing systems, etc.

- **Firm:** A real-time task/system is considered to be firm if generating the outcomes after its deadline is of no use for the system, but does not create any destruction. For example, railway ticket reservation system.
- **Soft:** A real-time task/system is considered to be soft if generating the outcomes after its deadline still provides usefulness for the system, however affecting a performance degradation. For example, multimedia applications on the mobile phone.

1.2 Features of RTOS [4] [17]

There are various features of RTOS like Synchronization, Interrupt Handling, Timer and clock, Real-Time Priority Levels, Fast Task, Preemption and Memory Management among them our focus is towards Memory Management.

Real-time operating system for huge and standard sized application are predictable to offer virtual memory, not only to achieve the demands of memory but to provide the memory request of non-real-time applications as well such as different types of editors, browsers, etc. A real-time operating system normally has small memory size by comprising only the essential features for an application [12].

1.3 Memory Management

Generally, memory management of Real-time operating system can be categorized as static memory management and dynamic memory management [35]. Table 1.2 [4] shows the fundamental difference between static memory management and dynamic memory management.

Table 1.2: The Fundamental difference between static and dynamic memory management

	Static Memory management	Dynamic Memory Management
1	Memory allocation is done at compile or design time.	Memory allocation is done at runtime or during execution.

2	Static memory allocation is a fix process which means requisite memory for a specific process is already identified, and after allocating memory no modifications can be done during execution.	Dynamic memory allocation needs memory manager to maintain which portion of the memory is allocated and which portion of the memory is unallocated. Due to this when a process requests memory, it can allocate memory and when the task is done then deallocate it.
3	Allocation and deallocation of memory are not performed during execution.	Memory bindings are established and demolished during execution.
4	Extra memory space required.	Less memory space required.

1.4 Problem Statement

Since last four to five decades, the majority of the operating systems have been used dynamic memory allocation for processing which requires communicating explicitly with memory allocator component. Though memory allocation algorithms have been analyzed and worked upon broadly since 1960, it has been observed that there is less attention given to the multiprocessor architecture and real-time operating system as well. Most of the algorithms have been designed such that, they are applicable for the general-purpose operating system and do not fulfill the necessities of real-time systems [37]. Moreover, limited allocators designed to support real-time systems which are not completely scalable for multiprocessors. In the 21st century, as we have entered into an era of high-performance computing, the demand for multi-core architecture has gained momentum. NUMA architecture based systems are the outcome of this tendency and offer an organized scalable design indicating that a few dynamic memory allocators are available. However, these dynamic memory allocators are not capable of performing on a multiprocessor architecture and do not comply with real-time system requirements as well. Researches have proved that the existing memory allocators for any operating systems which support NUMA architecture are not suitable for real-time applications. Hence, there is a need to have a dynamic memory allocator which can perform well on SMP and NUMA based soft real-time systems, with

better execution time and less fragmentation. This research is carried out in the same direction to achieve the aforementioned goal of a dynamic memory allocator for real-time systems.

1.5 Objectives of Memory Management Algorithm

The research in dynamic memory management for real-time systems is one of the unconquered areas primarily because real-time applications impose different requirements on memory allocators from general-purpose applications. Actually, most significant requirements in real-time systems are the investigation of scheduling which should be achieved to decide if the response time of real-time application can be bounded to fulfill the timing restriction of execution. This investigation should consider the impression of multiprocessor architecture settings like concurrency, lock contention, cache misses and traffic on the bus. Effect of all these problems on NUMA architecture systems, related to dynamic memory management could be defined as follows:

1) Reduce memory fragmentation
2) Restricted execution time
3) Increase node-based locality
4) Reduce false sharing
5) Reduce memory access to the remote nodes
6) Reduce lock conflicts

1.6 Research Contributions

1. Dynamic memory allocator **DmRT** has been designed and implemented for symmetric multiprocessing system which provides consistent and optimum execution time, less memory fragmentation, as well as satisfying a maximum number of the memory request, compare to other existing allocators.
2. As per the need of high-performance computing, a dynamic memory allocator **DmRT** for NUMA architecture based real-time operating system has been designed and implemented

which provides consistent and optimum execution time, less memory fragmentation as well as satisfying a maximum number of the memory request.

3. There are so many simulators available to simulate different test cases for scheduling in a real-time operating system like Litmus-RT, Mark3, rtsim, etc. but till date, no such simulator is available for simulating memory management algorithm for RTOS. Hence MemSimRT has been designed to simulate various memory allocators for both SMP as well as NUMA architecture based RTOS.

   Download MemSimRT using following QR code

Chapter 2
Literature Review

2.1 Dynamic Memory Management Algorithms

Memory management is the key feature of the real-time operating system. This section describes certain memory management algorithms for general-purpose as well as the real-time operating system. There are conventional as well as unconventional algorithms for dynamically allocation/deallocation of memory. All traditional algorithms can be considered as conventional algorithms. 1) Sequential Fit Algorithm 2) Buddy Allocators 3) Doug Lea(DLmalloc) 4) Half-Fit 5) TLSF 6) tcmalloc 7) Hoard 8) Smart Memory Allocators

1) Sequential Fit Algorithm

It can be categorized into four types [4] [17] [37]: Best Fit, Next Fit, First Fit and Worst-Fit.

a. Best Fit: Its name itself suggest that each time the allocator tries to search out the smallest unallocated memory block which is big enough to fulfill the application's request.
b. Next Fit: The array of unallocated blocks is to be found from the location where the previous search suspended, returning the next memory block which is big enough to fulfill the request.
c. First fit: The array of unallocated blocks is to be found from scratch, returning the first memory block which is big enough to fulfill the request.
d. Worst fit: The array of unallocated blocks is searched, returning biggest existing unallocated memory block.

The time complexity of this algorithm is O(n).

2) Buddy Allocator Algorithm

This algorithm [31] uses an array of link lists of the unallocated blocks. Each list for allowable block size. For example list of 2N block size like 200 kb, 400 kb, 800 kb so on. The

buddy allocator finds the smallest block which is large enough to hold the request from the list of unallocated blocks. If the unallocated block list is empty, then it will search a block from another list which is larger than a request, then select and split the block [6][10]. A block must be divided into the same size blocks, i.e., 400 kb block will split into two 200 kb blocks. In the same way, the block may be merged with its adjacent block of the same size, and this is possible if the adjacent block has not been divided into the smaller block.

3) Dlmalloc

This algorithm was proposed by Doug Lea in 1996. Later on, its extended version has been designed by Gloger in 2006 [20] and by Free Software Foundation in 2012. This algorithm occupies a huge number of static size arrays known as small-bins to allocate a small memory block. Bins occupy unallocated blocks which are having sizes not more than 256 bytes. Every bin comprises unallocated blocks of equal size. If demanded memory block's size is not more than 256 bytes, the algorithm tries to find for existing blocks in the bins using best-fit policy or large enough to satisfy the request.

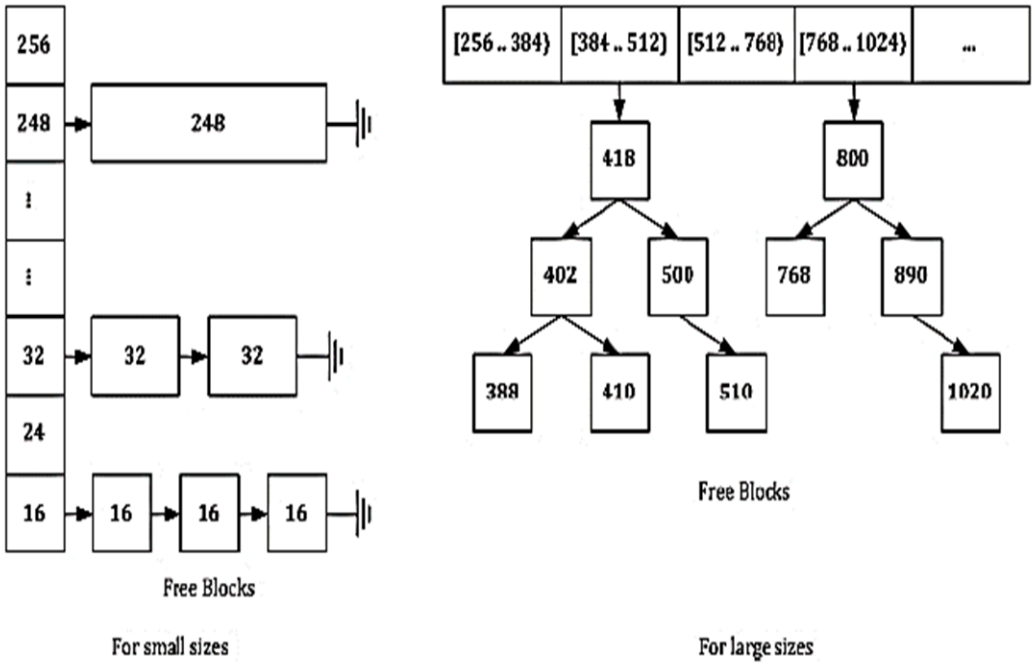

Figure 2.1: Organization of DLmalloc algorithm [37]

If the demanded memory block's size is larger than 256 bytes and smaller than some fixed value (normally 256 Kb), then algorithm tries to search existing blocks in an array known as tree-bin, which is having a tree structure for the memory block. As shown in Figure 2.1 tree-bins accumulate a collection of the bin. Nodes in the tree structure act as a small-bin, comprising the blocks of equal size. Any demand of block size beyond the fixed value, the algorithm transfers the requests to the operating system through some specific system call.

4) Half-Fit

This algorithm has been designed and implemented by Ogasawra [29] [30] in 1995. This algorithm uses bitmapped fit strategy and achieving execution time in constant manner. As it is using bitmap policy for allocating release block, it is slow. Use of bitmap is nothing but only to maintain the status of unoccupied lists. Its time complexity of time is O(1).

This algorithm maintains a segregated list of a single level. In this list, unallocated blocks of different size are connected. It takes unallocated blocks of the required size from unallocated block list through which request will be satisfied. Figure 2.2 shows this example.

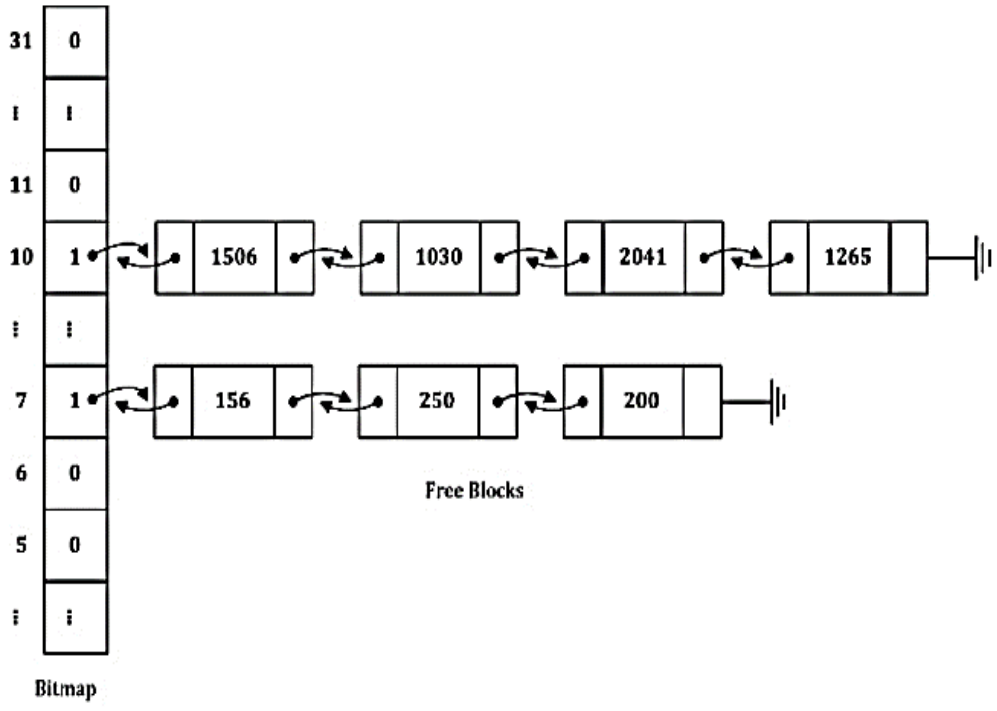

Figure 2.2: Organization of Half-fit algorithm [29] [37]

It has specific allocation/deallocation methodology to avoid searching using bitmaps because of its constant execution time. If the requested size of the memory block is r, then index i may be computed by this equation [30]:

$$i = \begin{cases} 0 & if\ r\ is\ 1 \\ \lfloor \log_2(r-1) \rfloor + 1 & otherwise \end{cases} \quad 2.1$$

Where i, specifies the unallocated memory block lists whose width vary from 2^i to $2^{i+1} - 1$. After computing the value of i, an unallocated block is occupied from the unallocated block list indexed by i. If there is no unallocated block in the list, the subsequent unallocated block list will be searched.

If the size of an assigned memory block is more than the demanded memory block size, an unallocated block from the unallocated block list will be split into two distinct memory blocks of sizes r1 and r2 before assigning for allocation then the remaining memory block r2 will be put into the matching unallocated block list. For deallocation, released memory block will be directly merged with neighboring memory block if the corresponding block is free/unallocated.

This algorithm is suitable for real-time operating systems because of its constant time complexity.

5) TLSF

TLSF is one of the best available dynamic memory allocation algorithm stands for two-level segregated fit algorithm, unlike segregated list allocator, this algorithm having two level of segregated lists of unallocated memory blocks in which each list maintain the unallocated blocks of predefined size range [25][26][28].

The first-level of list (FLI) splits unallocated memory blocks into various parts which are apart by the power of two like 2, 4, 8, 16 onwards. The secondary level known as second-level lists splits each first level list by a user-defined variable known as Second Level Index. TLSF structures are shown in Figure 2.3.

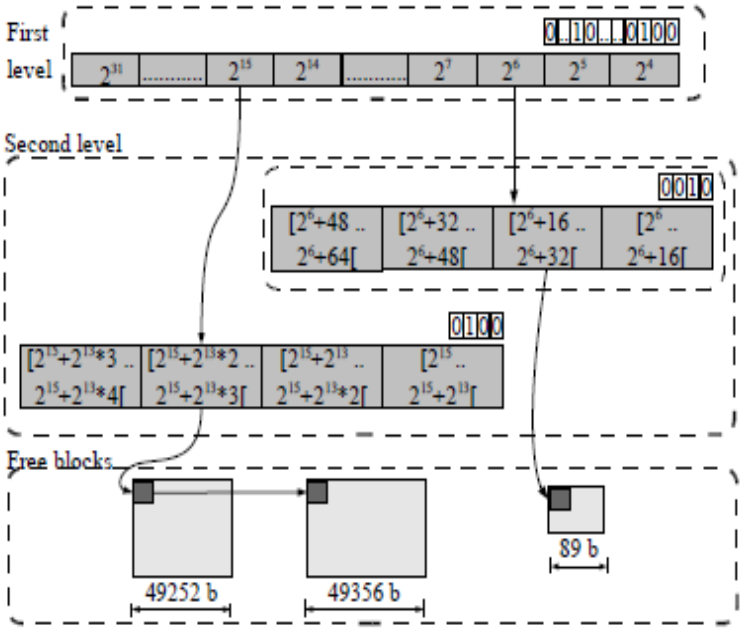

Figure 2.3: Simple TLSF Structure [25][26]

This algorithm provides bounded execution/response time. This algorithm is best suitable for the real-time operating system.

6) tcmalloc

This algorithm is developed and implemented by Sanjay Ghemawt in 2010 [36]. It is an extremely accessible algorithm which associates both global heap structure and threads private heap multiprocessor architecture. To allocate small memory block which size range from 4 bytes to 32 Kb, this allocator allocates private local heap to each thread. Hence, small size memory block allocation does not require synchronization mechanism for the thread.

To allocate large memory blocks which ranges from 32 Kb to 1 Mb, this allocator maintains a global heap structure which is collectively used by all available threads. As this global heap is shared by threads, some kind of synchronization mechanism should be used to offer mutual exclusion. Hence, it employs spin-lock mechanism. If any applications demand a huge memory block whose size is beyond 1Mb, then allocator forwards the request to the existing operating system using a system call or APIs. Figure 2.4 shows the structure of tcmalloc allocator. The time complexity of this allocator is O(1).

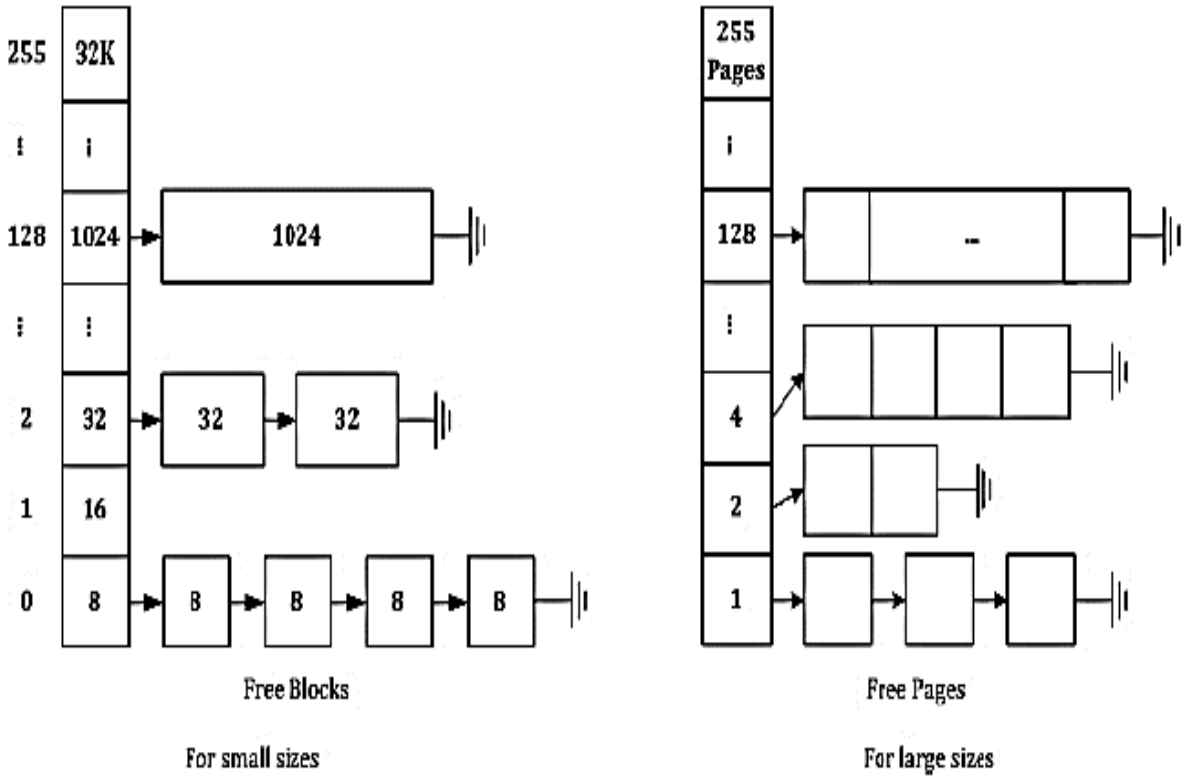

Figure 2.4: Organization of the tcmalloc algorithm [36]

7) Hoard

This algorithm was designed by Bergar in 2000 [17], and it is popular due to its speed and scalable in the environment of the multiprocessor system. This allocator also employs a segregated class mechanism, but unlike tcmalloc it maintains private heap to each processor and to avoid heap conflict it maintains a global heap. Other than private processor heap and global heap, it also maintains private heap per thread to allocate smaller size memory blocks which size less than 256 bytes. Threads which are executing on the same processor can also share private processor heap.

Hence, to allocate any blocks whose size is less than 256 bytes, it first searches it into a heap of the thread if it fails, then it searches into private processor heap, and if it is also full, then it searches into a global heap. So its time complexity is O(n), where n is the number of chunks of the memory block which is known as super-block., Figure 2.5 shows the structure of this algorithm.

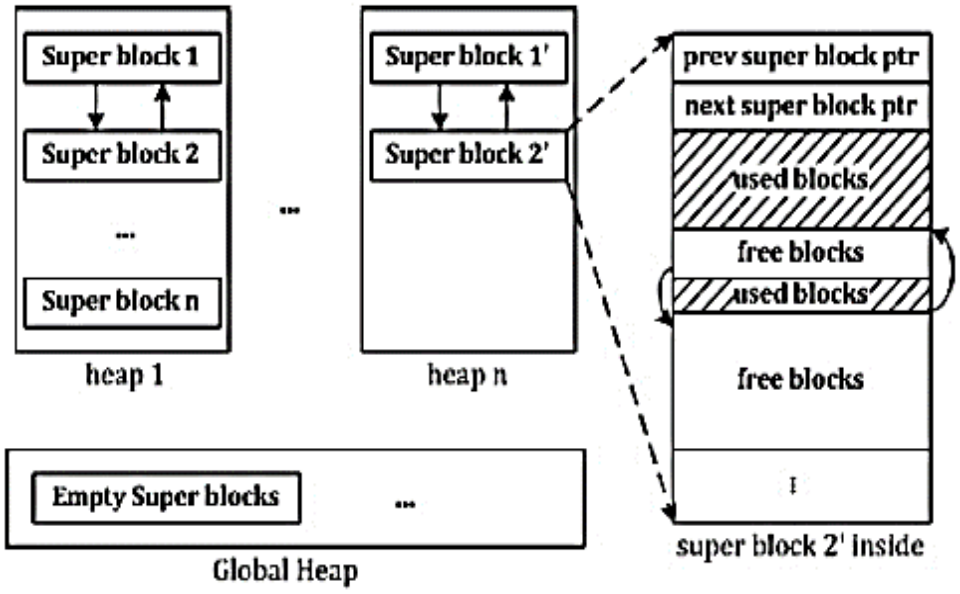

Figure 2.5: Organization of Hoard algorithm [2]

8) Smart Memory Allocator Algorithm

This algorithm has been proposed by Ramakrishna M, Jisung Kim, Woohyong Lee and Youngki Chung in 2008 [47]. This is a custom type of dynamic memory algorithm having the best response time and less memory fragmentation. This algorithm divides memory blocks into two categories. One is short-lived, and another is long-lived memory blocks. The short-lived memory blocks are allocated in the direction of lowest level to highest level from heap while long-lived memory blocks are allocated from highest level to lowest level [47]. The used space of heap grows from highest level to lowest level as well as lowest level to highest level. Initially, entire heap memory is unallocated, and there is only one unallocated memory block for each short as well as long-lived memory. The heap space is divided equally into two blocks. When the heap grows from both sides, the virtual border between these two can easily modify according to the dynamic memory request.

As this algorithm predicting memory object life scope, it can easily allocate memory block from either short-lived or long-lived memory pool [9]. So it can have best response time with lower fragmentation. It is implemented with a lookup table and hierarchical bitmaps which are improved version of the multilevel segregated mechanism.

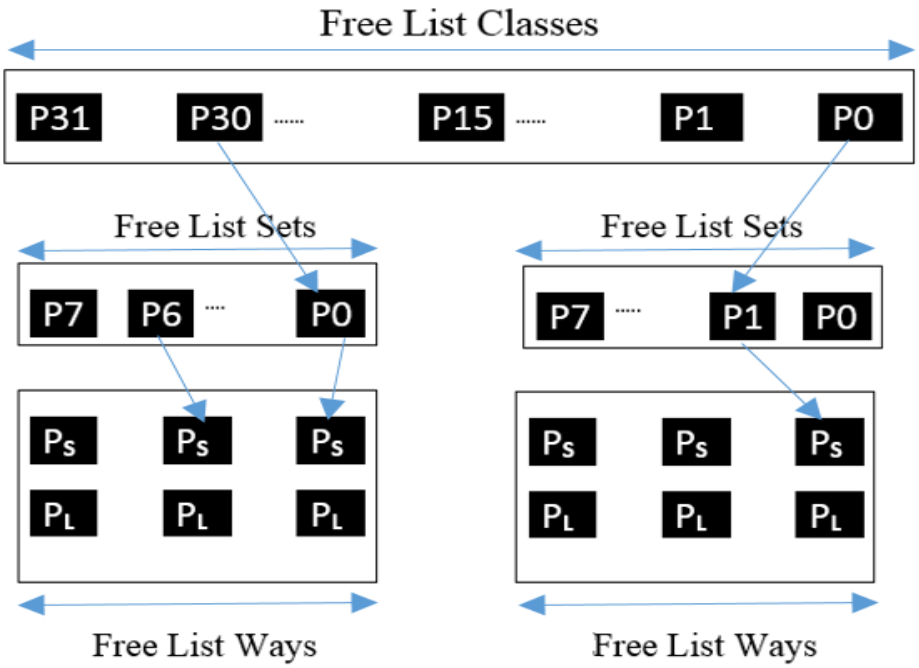

Figure 2.6: Structure of Smart Memory Allocator [41] [47]

2.2 Summary

Table 2.1: Summary of all Allocators

Memory Management Algorithms	Parameters		
	Allocation	**Fragmentation**	**NUMA Support**
Sequential fit	O(n)	Acceptable	No
Buddy System	$O(\log_n 2)$	Unacceptable	No
Doug Lea(DLmalloc)	O(m)	Acceptable	No
tcmalloc	O(1)	Acceptable	Yes
Hoard	O(n)	Acceptable	No
Half-fit	O(1)	Unacceptable	No
TLSF	O(1)	Acceptable	No
Smart Memory Allocator	$O(\log_2 n)$	Unacceptable	No

Table 2.1 shows in the worst-case, the time complexity of the allocators as well as fragmentation by the allocator is acceptable or not and whether it provides support to NUMA architecture or not. In the table, n is the heap size, m is the tree's depth. Concerning real-time systems segregated Fit, tcmalloc and Half-fit are the only algorithms which satisfactorily provides the desired objectives also provide a constant execution time.

Chapter 3
DmRT for SMP & NUMA

In this chapter, a new dynamic memory allocator for the real-time operating system will be discussed. It has been proposed, designed and implemented for Symmetric multiprocessing (SMP) NUMA (Non-uniform memory access) architecture. And its name is **DmRT** stand for Dynamic memory manager for Real-Time systems. This allocator has been designed to achieve constant and minimum execution time, low fragmentation and satisfying a maximum number of request for the memory block. Furthermore, the DmRT has been compared with the existing dynamic memory allocators of the real-time operating system.

All the design principals such as strategies, policies, and mechanisms will be explained then the structure of DmRT, and its results will be discussed in this section.

3.1 Design Principals

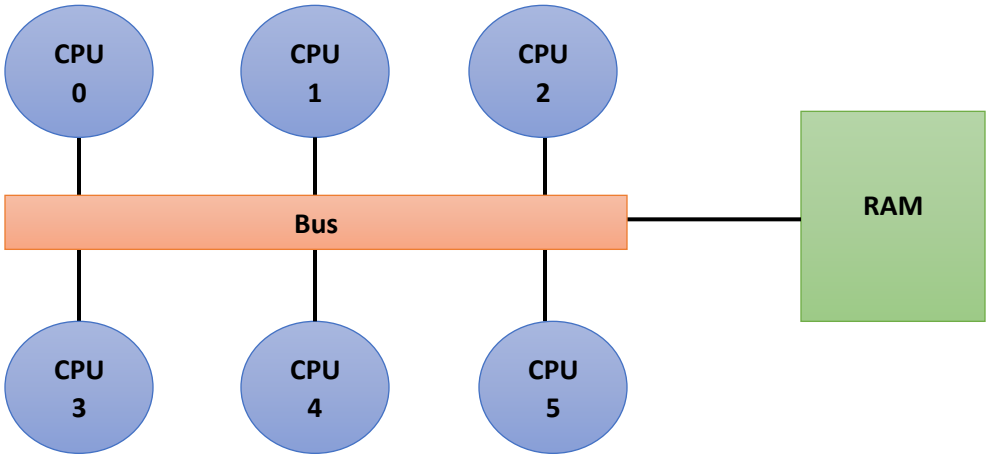

Figure 3.1: SMP Architecture

As shown in Figure 3.1, Symmetric multiprocessing (SMP) comprises a multiprocessor computer hardware and software architecture in which more than one identical processors are connected to a common or rather shared main memory, and all processors have full access to all

resources like input and output devices, and are managed by a single operating system instance that treats all processors equally, reserving none for special purposes. Nowadays, the majority of the multiprocessor systems use the SMP architecture. While in the multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.

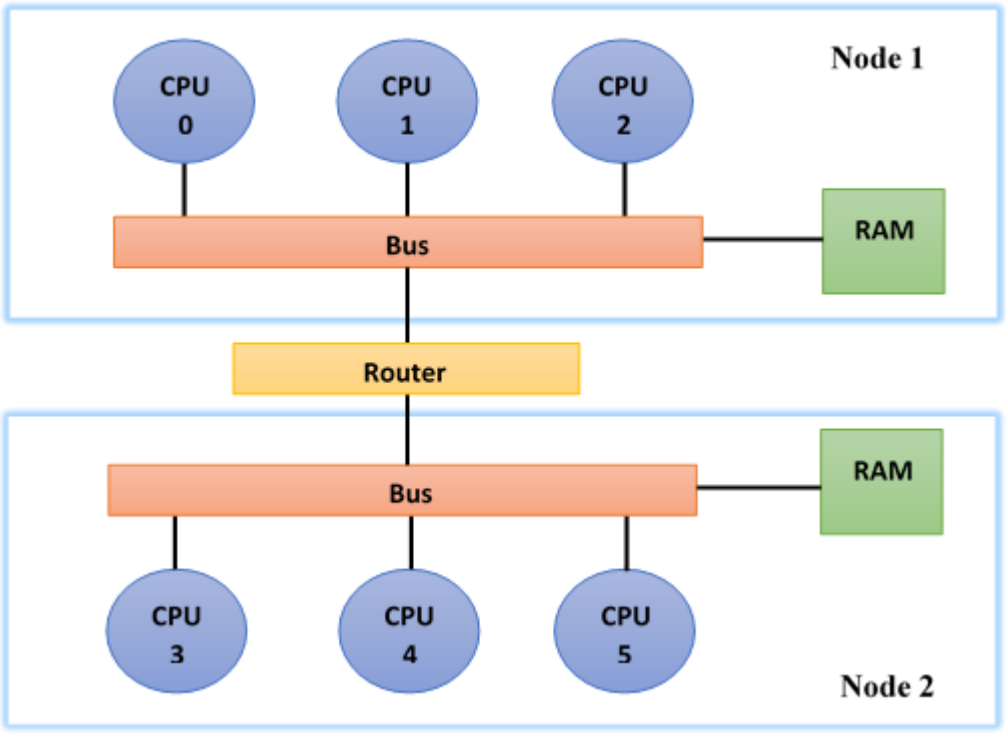

Figure 3.2: NUMA Architecture

NUMA stands for Non-uniform memory access (NUMA) is nothing but one type of design for a computer memory which is used in multiprocessing in which the access time of memory depends on the memory location relative to the processor. In NUMA architecture, a processor can read/write from its local memory faster than non-local memory, i.e., the local memory of other processor or shared memory between processors. The benefits of NUMA are limited to particular workloads, notably on servers where the data is often strongly associated with certain tasks or users.

In high-performance computing generation, NUMA is the future of SMP, but its architecture is clumsier than the Symmetric multiprocessor. Figure 3.2 shows simple NUMA architecture where only two nodes are available in which each node contains more than one processor and they all are sharing one memory. They may have their local memory as well. The complex architectures are also available which can have 4 or 8 nodes. 4 nodes architecture has been considered for this proposed memory allocator, but it is merely easy to scale it up to 8 nodes.

There are different strategies for different size of block in which has been explained in this section.

3.1.1 Multiple strategies for different sizes of blocks (for SMP & NUMA)

As discussed 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 (for SMP & NUMA)

After defining the strategies, now its turn to which policies and mechanisms will be used to implement these strategies.

I. For small blocks, the best-fit policy is used which have been implemented by exact-fit mechanisms to reduce the fragmentation in small sizes of blocks generated by rounding up the request size of the memory block.
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 is used.

3.1.3 Arrangement of blocks (for SMP & NUMA)

DmRT implements the exact-fit mechanism to increase the efficiency of small memory block allocation and to decrease internal fragmentation. It also implements the segregated-fit mechanisms to employ a good-fit and a first-fit policy for searching the nearest segregated size class. Thus it can ignore the requirement of a thorough search. Actually, two types of bitmaps have been used to keep track of unallocated blocks in the implementation of DmRT. Furthermore, this allocator is predictable as it has employed segregated list with bitmap policies and it provides confined execution time.

Among the bitmaps, use of one bitmap is to keep track of small memory blocks, and this bitmap is employed as a two-dimensional array for holding unallocated memory blocks according to the memory block size. In this proposed memory allocator (DmRT), for effective memory allocation, the block size is set apart 4 bytes from 4 bytes to 512 bytes. To check whether a specific size of a memory block is unallocated or not, two mechanisms have been employed. One is in two bitmaps total 64-bits and second is maintaining a pointer array to hold unallocated blocks as shown in Figure 3.3.

The second type of bitmap comprises a two-dimensional bitmap array directing to unallocated memory blocks. The primary bitmap which is indexed by i, specifies 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 easiness, the number of ranges in the secondary level is signified as the power of two: $\mathbf{2}^{\boldsymbol{range}}$. For this allocator, the default value of the ***range*** is taken as 6. The variable *range* splits the primary level ranges in an equal number of ranges. For example, if value of *range* is 4 then there are 16 segregated lists inside the provided size ranges, if value of *range* is 5 then there are 32 segregated lists inside the provided size ranges and so on and if value of *range* is 1 then the allocator accomplishes unallocated blocks as powerfully as the binary buddy allocator.

The value of the *range* is crucial to specify the performance of the allocator. Because it is important to decide the minimum size of the memory block. If the value of the *range* is big, then it causes more consumption of memory space for the storing information like extra bits and

pointers. Conversely, If the value of the *range* is too small, then it increases the internal fragmentation accordingly.

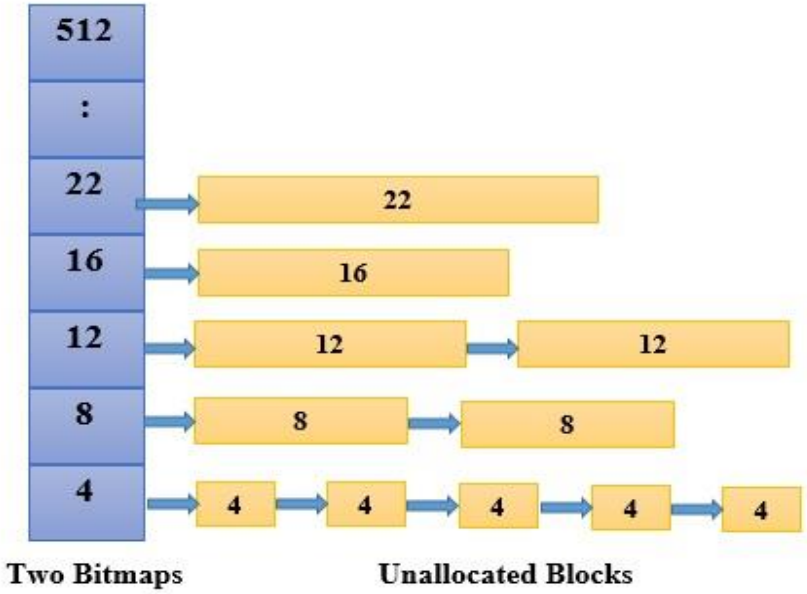

Figure 3.3: DmRT Structure for Small Block Allocation

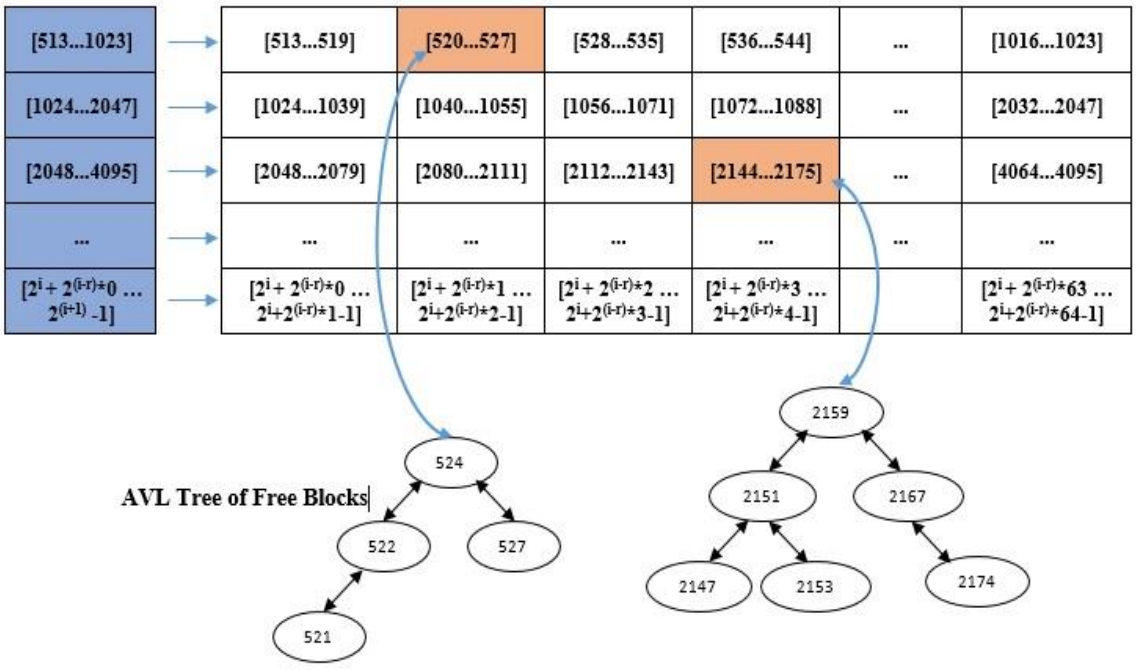

Figure 3.4: DmRT Structure for Normal Block Allocation

Therefore, the index i denotes the existing maximum size of a memory block: $2^{i+1} - 1$, while the number of segregated lists in the provide 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), and the value of index I specify 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 pointers to unallocated memory blocks are kept in two-dimensional pointer array which is known as matrix.

As discussed, for this proposed allocate, the value of *range* is set to 6 by default, each and every component of the array indicates to a list which has unallocated memory blocks of sizes in a range from $2^i + 2^{(i-range)} \times j$ to $2^i + 2^{(i-range)} \times (j+1) - 1$.

According to the employment of this allocator, it employs two-dimensional bitmap arrays, which needs a 64-bit variable for the primary bitmap and two-time 64-bit variables for the secondary bitmaps, therefore total 66 variables of 64-bit to specify the unallocated block lists are required. Also in each secondary level range, all available memory blocks arranged in AVL tree inorder to balance the tree structure.

The Primary Level is intended to accomplish the time of execution in a constant manner for allocation of memory blocks. Every segregated list keep the specific size of unallocated memory blocks, and the proposed algorithm 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, and also it is designed using a bitmap, arrays of pointers to unallocated blocks and doubly-linked lists for normal sizes of memory blocks. Sharing a single global heap between more than one threads having a tendency to raise the possibility of lock conflicts. To decrease this, every thread of the application should have a private thread heap.

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

3.1.4 Strategy for selecting Remote Memory

We have proposed one memory allocator which can work on NUMA based architecture for the real-time operating system. Figure 3.5 shows a schematic diagram of NUMA based architecture for RTOS. As shown in the figure, there are total four nodes where each node having two processors, each processor within a node are connected with a bus, and all nodes are connected with shared memory. Also, each processor having their own local (private) memory.

Whenever any processor requires any memory block it will first check into its local memory, if the required memory block is available then it will allocate the same block from the local memory and if not then it will try to access from the shared memory, if the memory block is there in shared memory then it will allocate but if it is not there then it will ask for another processor which is lightly loaded in terms of memory. Now, what is lightly loaded processor? Let's check it.

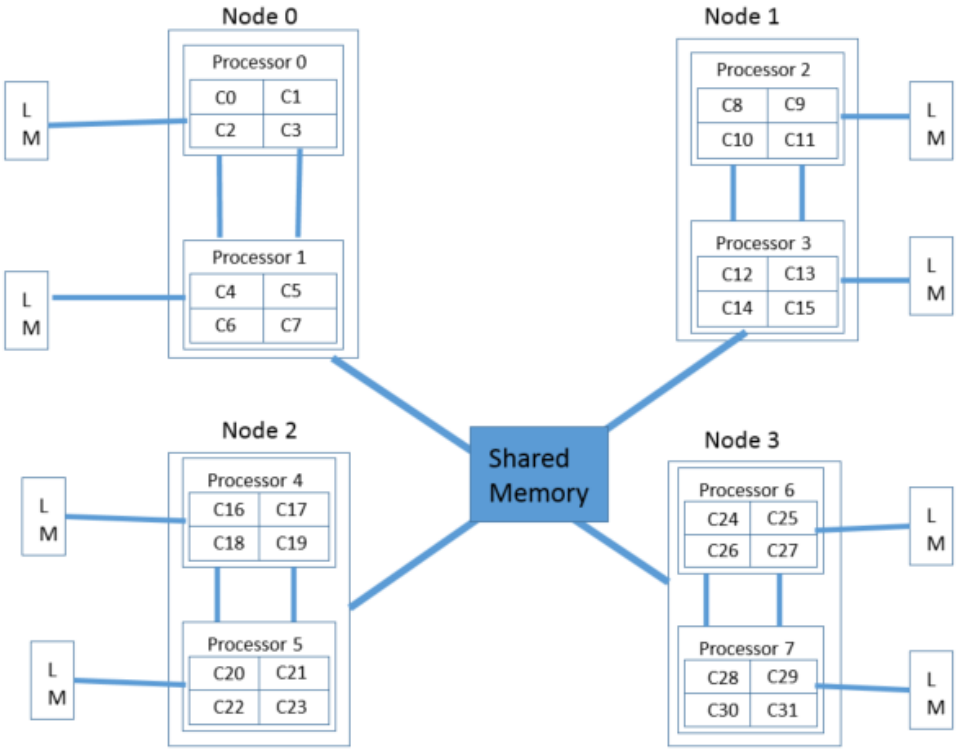

Figure 3.5: Complex NUMA Structure (4 Nodes)

According to memory utilization, each processor will be categorized into four categories.

1) Ideal

2) Heavily Loaded

3) Normal Loaded

4) Lightly Loaded

The first step is to calculate the load average for memory utilization for all processors using following equation [32] [38].

$$Mem_{u\_avg} = \frac{Mem_{u1} + Mem_{u2} + Mem_{u3} + \cdots + Mem_{un}}{n} \quad 3.2$$

The second step is to find the upper and lower threshold value for memory utilization using following equation.

$$T_U = H \times Mem_{u\_avg}$$
$$T_L = L \times Mem_{u\_avg} \quad 3.3$$

Where, T_U = upper limit of threshold,
T_L = lower limit of threshold,
U and L are constants. (U >1 and L< 1)

In the proposed algorithm, U and L are set to be 1.3 and 0.7 respectively which interpret if memory utilization is 30% above the Mem_{u_avg}, it is heavily loaded. And if memory utilization is 70% of the Mem_{u_avg}, it is lightly loaded; otherwise, it is normally loaded.

Hence, Light weight Memory <= 35% of Threshold value

Heavy weight Memory >= 65% of Threshold value

Average Memory node > 35% to < 65%

Ideal Memory < 10% Threshold value

And then select appropriate processor's memory for allocating memory.

Chapter 4
Results & MemSimRT

4.1 MemSimRT

There are so many simulators available to simulate different test cases for scheduling in a real-time operating system like Litmus-RT, Mark3 etc. But till date, no such simulator is available for simulating memory management algorithm for RTOS. So MemSimRT has been designed to simulate various memory allocators for both SMP as well as NUMA architecture based RTOS. Its front end created in C# while back-end developed using python.

Download MemSimRT using this QRcode:

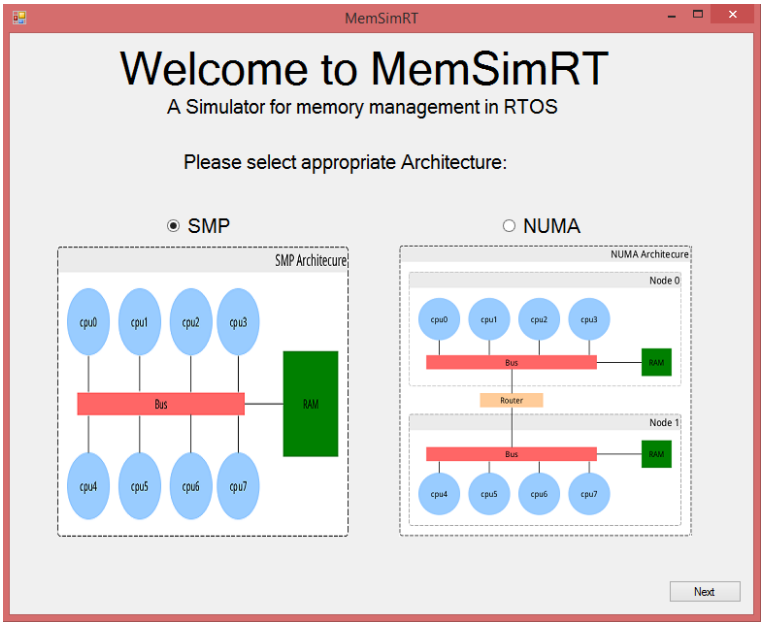

Figure 4.1: The welcome screen of MemSimRT

Figure 5.1 shows the welcome screen of MemSimRT. So it is the Home screen of the simulator. As per our dynamic memory allocator, it has two alternatives. One is SMP, i.e.,

Symmetric multiprocessor and second is NUMA, i.e., Non-uniform memory access based architecture. Basically, in this simulator, NUMA is designed for eight processors, but it can be modified as per requirement by slightly changing the script.

4.2 Results

There are five different test cases.

1. SMP

Case 1: Existing allocators and DmRT allocate from Local Memory

As it is a Symmetric MultiProcessor architecture, all processors will share the same memory which is known as local memory for them. And whenever any request for the memory block is raised then, the memory manager will search and allocate memory block from the same local memory.

2. NUMA

Case 2: Existing from Local and DmRT Follow Local → Shared → Ideal

Existing allocators means Dlmalloc, tcmalloc and TLSF will allocate the memory block from local memory while DmRT will first try to allocate block from local memory; if it fails then it will attempt the same from shared memory and still if it will get failure then it will find ideal memory which has been discussed earlier and then it will allocate block from it. As DmRT tries to find memory block from three different types of memory, its execution time will be more than the other allocators, but it provides consistent execution time. And also it will satisfy a maximum number of the request as well as it will have less fragmentation due to proposed allocator structure.

Case 3: Existing allocators from Local and DmRT from Ideal

In this case, all existing allocator will allocate memory block from Local memory only. While DmRT first finds the idle memory and then it will allocate memory block from it. Here, existing allocators allocating blocks only from local memory that's why it can have less number

of request satisfaction while DmRT will have a maximum number of request satisfaction. Also, other parameters will be best due to its structure.

Case 4: Existing allocators and DmRT both from Ideal

In this case, existing allocators and DmRT both will first find idle memory and then allocate a block from it. As existing allocators and DmRT, both allocate memory from ideal memory, execution time will be moreover same, but still, DmRT will have a maximum number of request satisfaction and less fragmentation.

Case 5: Existing allocators and DmRT follow Local → Shared → Ideal

In this case, existing allocators and DmRT both will first try to allocate memory block from local; if they fail then they will try to allocate the same block from shared memory and still got the failure then they will find idle memory and try to allocate same memory block from it. Though both existing allocators and DmRT follow the same path from allocating memory, proposed allocator defeats all of them in each parameter.

In each case, there are three different test categories have been selected.

a. Best case, i.e. test has been taken for 100 memory blocks request.
b. Average case, i.e. test has been taken for 1000 memory blocks request.
c. Worst case, i.e. test has been taken for 2000 memory blocks request.

There are three main parameters are considered for the results.

Parameter 1: Execution time. It should be consistent and minimum.

Parameter 2: Fragmentation. It should be as low as possible.

Parameter 3: Number of requests satisfied: It should be as high as possible.

Here, total four memory management algorithms have been compared.

a. Dlmalloc b. tcmalloc c. TLSF d. DmRT

All tests have been done on MemSimRT.

Table 4.1: Results of All Allocators in All Test cases

Parameter		Execution Time				Fragmentation				Request Satisfied			
Test No.	**Cases**	**Dlmalloc**	**Tcmalloc**	**TLSF**	**DmRT**	**Dlmalloc**	**tcmalloc**	**TLSF**	**DmRT**	**Dlmalloc**	**Tcmalloc**	**TLSF**	**DmRT**
Case 1	**Best Case**	287.8581	330.3003	268.598	234.6128	43.6472	29.684	22.4791	17.5031	56.6156	62.5883	81.5737	87.6169
	Average Case	1904.826	2890.503	1461.272	1067.995	52.3926	35.157	27.0205	22.0902	45.458	57.4617	74.9894	83.109
	Worst Case	3204.577	4352.133	2153.912	1847.152	60.4389	43.6719	32.0433	26.9948	34.903	52.783	70.9776	77.3786
Case 2	**Best Case**	326.2426	410.8068	290.2026	374.3901	43.5037	36.6849	21.5874	10.5141	57.5068	62.3301	76.6088	94.6614
	Average Case	2013.324	2988.474	1522.335	2303.212	52.5491	44.4944	29.5867	15.6241	45.2403	54.2546	65.6095	87.4181
	Worst Case	3202.561	4348.651	2049.025	3361.854	61.4645	43.3702	36.5828	20.5572	35.5009	50.4204	61.5822	83.7309
Case 3	**Best Case**	338.0589	420.5202	296.4418	245.4583	45.2763	33.6326	24.0237	15.4697	57.7885	64.1904	76.9458	89.9526
	Average Case	2054.716	2995.241	1539.964	1115.835	53.8153	44.9067	30.2955	19.5226	46.4232	54.5453	65.9168	82.8598
	Worst Case	3283.047	4288.159	2064.704	1785.676	60.4389	43.6719	32.0433	26.9948	34.903	52.783	70.9776	77.3786
Case 4	**Best Case**	374.8572	444.5905	319.6948	249.566	35.2178	26.3902	19.2872	14.5781	67.5358	72.1999	83.1096	89.1851
	Average Case	2110.58	3240.527	1530.277	1149.484	43.0248	35.5497	26.3408	19.7854	55.5939	64.722	73.3219	81.1776
	Worst Case	3277.428	4467.102	2172.658	1860.321	52.6015	43.6602	35.9747	25.1212	45.1788	53.4233	65.3064	74.1789
Case 5	**Best Case**	528.5204	636.5573	480.8449	385.8492	31.4948	23.8764	17.5356	10.4119	71.7431	78.1612	86.8777	93.7361
	Average Case	2928.034	4166.439	2541.517	2252.019	38.895	31.6255	22.913	15.0111	62.0389	70.6678	79.9134	87.5092
	Worst Case	4231.555	5107.635	3684.495	3367.702	47.3835	38.2598	30.8472	19.1314	52.5833	61.8889	74.2487	83.9386

As shown in the table, DmRT performs better concerning all other allocators (Dlmalloc, tcmalloc and TLSF) in all cases. Only in Case 2 where all existing allocators allocate memory from Local memory only and DmRT allocates from Local, Shared and Ideal in which execution time is more than Dlmalloc and TLSF but in same case number of request satisfied is maximum than others as well fragmentation is minimum than other allocators.

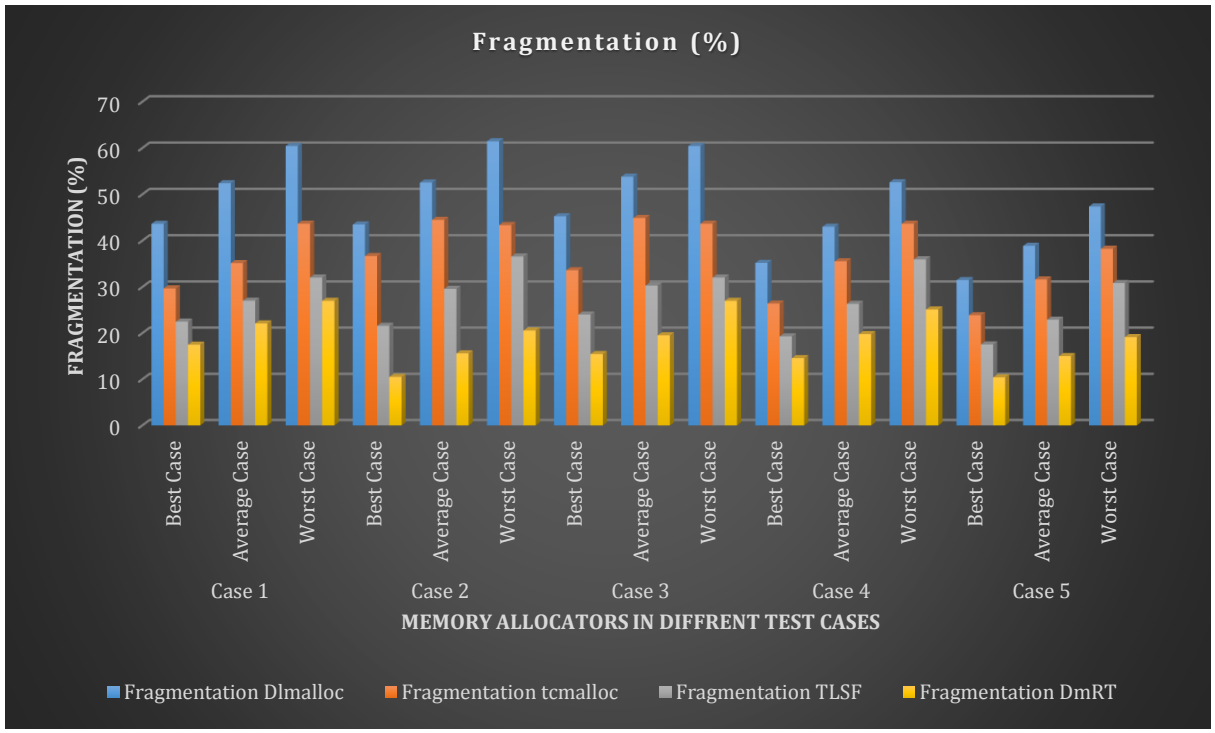

Figure 4.2: Fragmentation in % of all Allocators in All Test cases

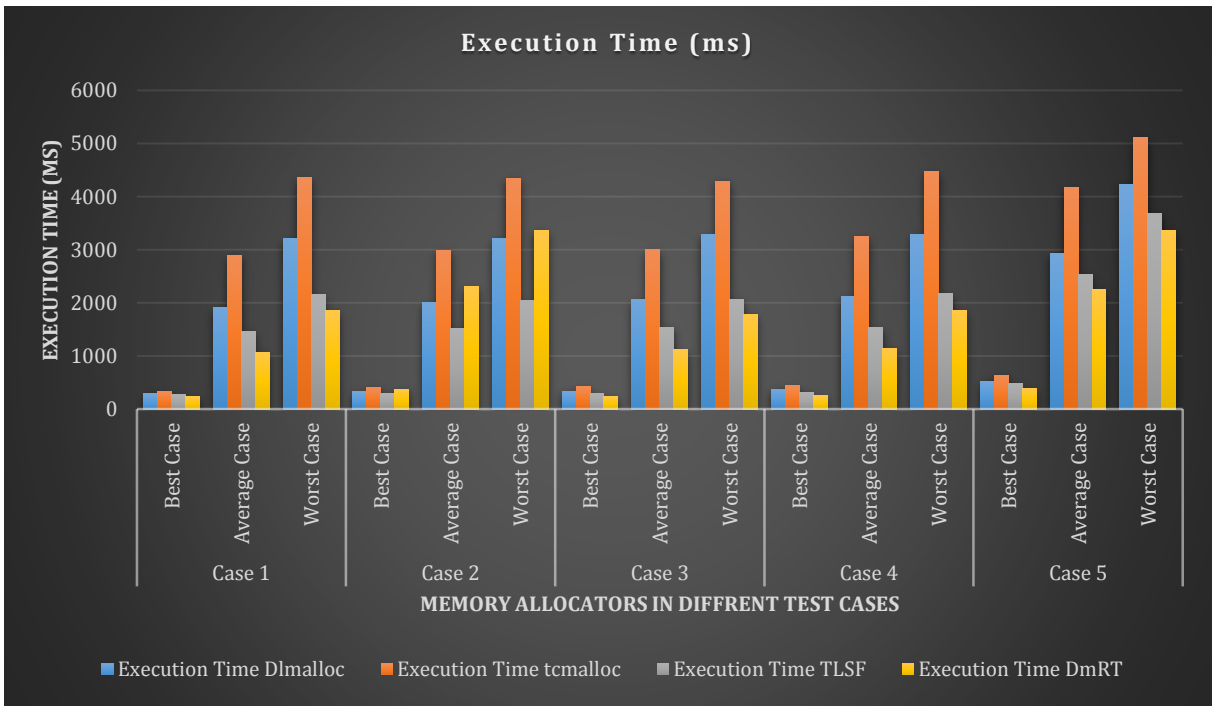

Figure 4.3: Execution time in (ms) of all Allocators in All Test cases

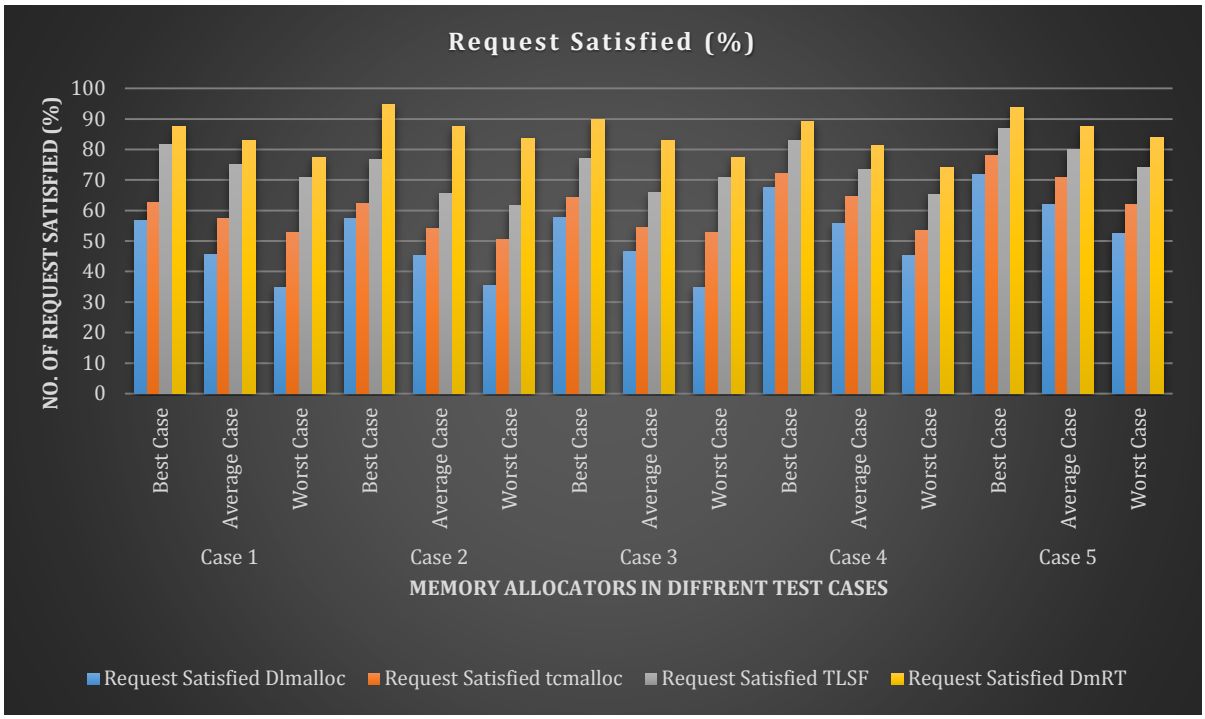

Figure 4.4: No. of Request Satisfied in % of all Allocators in All Test cases

References

[1]. Bays, C. (1977). A comparison of next-fit, first-fit, and best-fit. Commun. ACM, 20(3): (pp. 191–192).

[2]. Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. (2000). Hoard: a scalable memory allocator for multithreaded applications. SIGPLAN Not., 35(11): (pp. 117–128).

[3]. Christian Del Rosso. (2005). Dynamic Memory Management for Software Product Family Architectures in Embedded Real-Time Systems. Fifth Working {IEEE} / {IFIP} Conference on Software Architecture (pp. 211-212)

[4]. Dipti Diwase, Shraddha Shah, Tushar Diwase and Priya Rathod. (2012). Survey Report on Memory Allocation Strategies for Real-time Operating System in Context with Embedded Devices. International Journal of Engineering Research and Applications, Vol. 2, Issue 3, (pp.1151-1156).

[5]. Edge, J. (2009). Perfcounters added to the mainline. http://lwn.net/Articles/336542/.

[6]. Ferreira, T., Matias, R., Macedo, A., and Araujo, L. (2011). An experimental study on memory allocators in multicore and multithreaded applications. In Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on, (pp. 92 –98).

[7]. FSF, F. s. f. (2012a). Glibc, the gnu c library. ”http://www.gnu.org/software/libc/libc.html”.

[8]. FSF, F. s. f. (2012b). The gnu c++ library manual. “http://gcc.gnu.org/onlinedocs/libstdc++”.

[9]. Gergov, J. (1996). Approximation algorithms for dynamic storage allocation. In Algorithms — ESA ’96, volume 1136, pages 52–61. Springer Berlin / Heidelberg.

[10]. Gloger, W. (2006). ptmalloc2. ”http://www.malloc.de/en/”.

[11]. Hans-Georg Eßer. (2011) Combining memory management and filesystems in an operating systems course. Proceedings of the 16th Annual {SIGCSE} Conference on Innovation and Technology in Computer Science Education, Darmstadt, Germany.

[12]. Hasan, Y. and Chang, M. (2005). A study of best-fit memory allocators. Computer Languages, Systems & Structures, 31(1): (pp. 35 – 48).

[13]. Hasan, Y., Chen, W.-M., Chang, J. M., and Gharaibeh, B. M. (2010). Upper bounds for dynamic memory allocation. IEEE Trans. Comput., 59(4): (pp. 468–477).

[14]. Hewlett-Packard Corporation (2012). HP Pro-Liant DL980 G7 server with HP PREMA Architecture PREMA Architecture. Technical Whitepaper.

[15]. Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation (2011). Advanced configuration and power interface specification.

[16]. Hirschberg, D. S. (1973). A class of dynamic memory allocation algorithms. Commun. ACM, 16(10): (pp. 615–618).

[17]. Jane W. S. Liu. (2000). “Real-time System”, 1st Edition published by Person Education.

[18]. Johnstone, M. S. and Wilson, P. R. (1998). The memory fragmentation problem: solved? SIGPLAN Not., 34(3): (pp. 26–36).

[19]. Knuth, D. (1997). The art of computer programming: Fundamental Algorithms, volume 1. addison-Wesley, 2 edition.

[20]. Lea, D. (1996). A memory allocator. ”http://g.oswego.edu/dl/html/malloc.html”. Unix/Mail December, 1996.

[21]. Lei Liu, Mengyao Xie, Hao Yang. (2017). Memos: Revisiting Hybrid Memory Management in Modern Operating System. CoRR abs/1703.07725

[22]. Lei Liu, Yong Li, Chen Ding, Hao Yang, Chengyong Wu. (2016). Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random? IEEE Trans. Computers 65(6): (pp. 1921-1935)

[23]. Linus Torvalds, e. (2011). Source codes of linux kernel v3.0.4. ”http://lxr. linux.no/linux+v3.0.4/”.

[24]. Marchand, A., Balbastre, P., Ripoll, I., Masmano, M., and Crespo, A. (2007). Memory resource management for real-time systems. In Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference, (pp. 201 – 210).

[25]. Masmano (2012). The lastest version of TLSF source. http://wks.gii.upv.es/tlsf/files/src/TLSF-2.4.6.tbz2.

[26]. Masmano, M., Ripoll, I., and Crespo, A. (2003). Dynamic storage allocation for real-time embedded systems. Proc. of Real-Time System Simposium WIP.

[27]. Masmano, M., Ripoll, I., Balbastre, P., and Crespo, A. (2008a). A constant-time dynamic storage allocator for real-time systems. Real-Time Systems, 40(2): (pp. 149–179).

[28]. Masmano, M., Ripoll, I., Real, J., Crespo, A., and Wellings, A. (2008b). Implementation of a constant-time dynamic storage allocator. Software: Practice and Experience, 38(10): (pp. 995–1026).

[29]. Ogasawara, T. (1995). An algorithm with constant execution time for dynamic storage allocation. In RTCSA '95: Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications, pages 21–25, Washington, DC, USA. IEEE Computer Society.

[30]. Ogasawara, T. (2009). Numa-aware memory manager with dominant-threadbased copying gc. SIGPLAN Not., 44(10): (pp. 377–390).

[31]. Page, I. and Hagins, J. (1986). Improving the performance of buddy systems. Computers, IEEE Transactions on, C-35(5): (pp. 441 –447).

[32]. Paul Werstein, Hailing Situ, Zhiyi Huang. (2006). "Load Balancing in a Cluster Computer", Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'06).

[33]. Puaut, I. (2002). Real-Time Performance of Dynamic Memory Allocation Algorithms. In ECRTS '02: Proceedings of the 14th Euromicro Conference on Real-Time Systems, (pp. 41–49), Washington, DC, USA. IEEE Computer Society.

[34]. Puaut, I. and Hardy, D. (2007). Predictable paging in real-time systems: A compiler approach. In Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on, (pp. 169 –178).

[35]. Robart L. Budzinski, Edward S. Davidson. (1981). A Comparison of Dynamic and Static Virtual Memory Allocation Algorithms" IEEE Transactions on software Engineering, Vol. SE-7, NO. 1.

[36]. Sanjay Ghemawat, P. M. (2010). Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[37]. Seyeon Kim. (2013). Node-oriented dynamic memory management for real-time systems on ccNUMA architecture systems. University of York, UK.

[38]. Vatsal Shah, Kanu Patel. (2012). Load Balancing algorithm by Process Migration in Distributed Operating System. International Journal of Computer Science and Information Technology & Security (IJCSITS), ISSN: 2249-9555, Vol. 2, No.6.

[39]. V Shah, A Shah. (2017). Critical Analysis for Memory Management Algorithm for NUMA based Real-time Operating System. IEEE Xplore.

[40]. V Shah, A Shah. (2018). Proposed Memory Allocation Algorithm for NUMA based Soft Real-time Operating System. International Conference On Emerging Technologies In Data Mining And Information Security (IEMIS 2018)

[41]. Vatsal Shah, Apurva Shah. (2016). An Analysis and Review on Memory Management Algorithms for Real-time Operating System. International Journal of Computer Science and Information Security (IJCSIS), Vol. 14, No. 5.

[42]. Vee, V.-Y. and Hsu, W.-J. (1999). A scalable and efficient storage allocator on shared memory multiprocessors. In Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks, ISPAN '99, Washington, DC, USA. IEEE Computer Society.

[43]. Wellings, A. J., Malik, A. H., Audsley, N. C., and Burns, A. (2010). Ada and cc-numa architectures what can be achieved with ada 2005? Ada Lett., 30(1): (pp. 125–134).

[44]. Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. (1995b). Dynamic Storage Allocation: A Survey and Critical Review. In IWMM '95: Proceedings of the International Workshop on Memory Management, (pp. 1–116), London, UK. Springer-Verlag.

[45]. Wilson, P., Johnstone, M., Neely, M., and Boles, D. (1995a). Memory allocation policies reconsidered. Technical report, Technical report, University of Texas at Austin Department of Computer Sciences.

[46]. XiaoHui Sun, JinLin Wang, xiao chan. (2007). "An Improvement of TLSF Algorithm".

[47]. Youngki Chung, Ramakrishna M, Jisung Kim and Woohyong Lee. (2008). Smart Dynamic Memory Allocator for embedded systems. Proceedings of 23rd International Symposium on Computer and Information Sciences, ISCIS '08.

[48]. Zorn, B. and Grunwald, D. (1992). Empirical measurements of six allocation-intensive c programs. SIGPLAN Not., 27(12): (pp .71–80).

[49]. Zorn, B. and Grunwald, D. (1994). Evaluating models of memory allocation. ACM Trans. Model. Comput. Simul., 4(1): (pp. 107–131).

Publications

1. Vatsalkumar H. Shah, Dr. Apurva Shah, (May, 2016). ***"An Analysis and Review on Memory Management Algorithms for Real-time Operating System"*** *published in International Journal of Computer Science and Information Security, Volume 14, Issue 5, (pp. 236-240)* (Web of Science Thomson Reuters, Scopus, DOAJ)

2. Vatsalkumar H. Shah, Dr. Apurva Shah, (December, 2017). ***"Critical Analysis for Memory Management Algorithm for NUMA based Real-time Operating System".*** *In Proceedings of IEEE Conference, 2017 (International Conference on Intelligent Sustainable Systems 2017.).* (pp. 323-327). (INSPEC, Scopus Indexed)

3. Vatsalkumar H. Shah, Dr. Apurva Shah, (February, 2018). ***"Proposed Memory Allocation Algorithm for NUMA based Soft Real-time Operating System".*** As a book chapter in *Advances in Intelligent Systems and Computing (AISC), Springer Series*. (ISI, DBLP, EI-Compendex, SCOPUS) *(To be published)*

4. Vatsalkumar H. Shah, Dr. Apurva Shah, (June, 2018). ***"Memory Allocator for SMP & NUMA based Soft Real-time Operating System".*** As a book chapter in *Advances in Intelligent Systems and Computing (AISC), Springer Series*. (ISI, DBLP, EI-Compendex, SCOPUS) *(To be published)*