# Dynamic Memory Management for Resource Contrained Next Generation Wireless Sensor Nodes

Student Name: Supratim Das
Roll Number: MT12107

Thesis report submitted in partial fulfillment of the requirements
for the Degree of M.Tech. in Electronics & Communication Engineering with specialization in
VLSI & Embedded Systems

16th May, 2014



**Thesis Committee**
Dr. Amarjeet Singh (Chair)
Dr. Kaushik Saha (Samsung Engineering)
Mr. Amit Kumar (STMicroelectronics India Pvt. Ltd.)

Indraprastha Institute of Information Technology Delhi
New Delhi

This research was fully funded by ST Microelectronics Pvt Ltd.

## Student's Declaration

I hereby declare that the work presented in the report entitled **"Dynamic Memory Management for Resource Constrained Next Generation Wireless Sensor Nodes"** submitted by me for the partial fulfillment of the requirements for the degree of *Masters of Technology* in *Electronics & Communication Engineering in VLSI and Embedded Systems* at Indraprastha Institute of Information Technology, Delhi, is an authentic record of my work carried out under guidance of **Dr. Amarjeet Singh**. Due acknowledgements have been given in the report to all material used. This work has not been submitted anywhere else for the reward of any other degree.

**Supratim Das** **Place & Date: .....**

## Certificate

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

**Dr. Amarjeet Singh**
**IIIT-Delhi**

**Mr. Surinder Pal Singh**
**STMicroelectronics**

**Abstract**

Wireless Sensor Nodes are very low power resource constrained devices with limited radio communications ability. As the technology is evolving at a rapid pace, more and more complex protocols and applications are integrated to these devices. This leads to elevated demands to constrained resources like power, network bandwidth, computational abilities and memory. Often memory proves to be the most constrained among all these resources and an answer to more efficient use of memory is dynamic memory management. Embedded systems developers often prefer a worst case static memory allocation over a dynamic memory allocation for a number of reasons such as unbounded response time, suboptimal usage of RAM space due to fragmentation and certain memory management overhead. Another concern when using a dynamic memory management for an embedded system is that a runtime allocation failure may be catastrophic.

However it is always not possible to design applications with static memory allocations. To alleviate this problem many modern sensor network OS like Contiki, MantisOS, SOS etc. provides dynamic memory management support. Many widely accepted dynamic memory management techniques are not directly applicable for a resource constrained embedded system and so dynamic memory management schemes for resource constrained devices often employ techniques like memory compaction or a customized memory model to fit the use-case. During the span of this research it has been observed that the dynamic memory management API employed in these OS is not compliant with the standard C dynamic memory allocation API. From a software engineering perspective, it calls for a major redesign of an existing code base, and thus leads to unprecedented delays in development and it is generally associated with an unwanted NRE cost.

The primary contribution of this dissertation is to describe a configurable and scalable dynamic memory management scheme, for microcontroller based resource constrained embedded systems with an API compliant with the memory management API of the standard C library. Considering the fact that the proposed dynamic memory manager is targeted only towards severely resource constrained devices, it should eliminate redundant information in the memory descriptors to maximize memory utilization. The second contribution of the work experimentally verifies a hypothesis that was framed during the course of this research, attempting to reduce external fragmentation by localizing similarly sized allocation requests, in a particular region of the managed memory pool. As a final contribution of the work, an extension to the proposed dynamic memory manager is discussed which can potentially include memory compaction, and dynamic rearrangement of allocated blocks leading to more efficient use of the memory.

**Keywords:** Dynamic Memory Management, Constrained Embedded Devices, Wireless Sensor Nodes, Memory Fragmentation

## Acknowledgments

Foremost I would like to express my sincere gratitude to my advisor Dr. Amarjeet Singh for providing excellent guidance and support throughout the span of the project. Without his patience, motivation, critical judgement and outlook this work would never have been successful.

I take this opportunity to thank Prof. R.N. Biswas for his continuous support and his expert advise. A very special thanks IIITD and STMicroelectronics for providing me this opportunity and infrastructure support during the entire span of the research.

I would like thank Dr. Kaushik Saha for offering me the internship at STMicroelectronics and leading me to work in his group on diverse and exciting projects. My sincere thanks goes to Mr. Surinder Pal Singh, Mr. Amit Kumar and Mr. Mridupawan Das for helping me out with all sorts of problems, giving invaluable ideas, excellent guidance and enriching my technical know-how during my stay at STMicroelectronics.

Besides my advisor and mentors I would like to thank my cousin brother Dr. Rajarshi Mitra for his invaluable insights and guidance in all aspects of my life, including this work.

Special thanks to all my friends at IIITD especially Pranay Samanta, Hemanta Mondal and Dipto Sarkar for making me feel right at home, Rohan Sinha for his great stress buster acts and Srikrishna Acharya for being a great friend and project partner.

Although No amount of gratitude is enough I am still immensely grateful towards my family for everything they have done for me throughout my life and last but not the least I would like to thank the very special person in my life Neha Chakrabarty for bearing with my obnoxious unavailability yet supporting me all the way through.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

Infrastructural support for WSN applications in the form of operating systems is becoming increasingly important. It bridges the gap between hardware simplicity and application complexity. One of the important OS design issues which requires lot more attention is memory management in WSN. Memory in current sensor nodes consists of: RAM (for fast data storage), internal flash (for code storage), EEPROM (for data storage), and external flash which is required for data persistence. In a traditional operating system, memory management refers to various techniques used for allocation and de-allocation of memory blocks to different processes and threads. However in embedded systems, a static memory allocation model is preferred over a dynamic memory allocation. But with the emergence of new application domains in WSN which support real-time traffic, dynamically loadable modules [1] and cryptographic security [2] support for dynamic memory is becoming a necessity.

In this chapter, the basic background and concepts behind dynamic memory management is presented first, followed by a description of the related works done in the field of dynamic memory management in embedded systems. The chapter concludes by outlining the primary motivation behind the work that is presented in this dissertation.

## 1.1 Background

Memory management is the act of managing computer memory. The essential requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed. In the C programming language, dynamic memory management is supported via a group a functions in the standard C library, namely malloc, calloc, realloc and free.

### 1.1.1 Rationale

The C programming language manages memory statically, automatically, or dynamically. Static-duration variables are allocated in main memory, usually along with the executable code of the program, and persist for the lifetime of the program; automatic-duration variables are allocated on the stack and come and go as functions are called and return. For static-duration and automatic-duration variables, the size of the allocation is required to be compile-time constant. If the required

size is not known until run-time, then using fixed-size data objects is inadequate.

The lifetime of allocated memory is also a concern. Neither static- nor automatic-duration memory is adequate for all situations. Automatic-allocated data cannot persist across multiple function calls, while static data persists for the life of the program whether it is needed or not. In many situations the programmer requires greater flexibility in managing the lifetime of allocated memory.

These limitations are avoided by using dynamic memory allocation in which memory is more explicitly (but more flexibly) managed, typically, by allocating it from the free store (informally called the "heap"), an area of memory structured for this purpose. In C, the library function malloc is used to allocate a block of memory on the heap. The program accesses this block of memory via a pointer that malloc returns. When the memory is no longer needed, the pointer is passed to free which deallocates the memory so that it can be used for other purposes.

### 1.1.2 Description of Functions

The C dynamic memory allocation functions are defined in stdlib.h header.

| Function | Description |
|---|---|
| malloc | allocates the specified number of bytes |
| calloc | allocates the specified number of bytes and initializes them to zero |
| realloc | increases or decreases the size of the specified block of memory. Reallocates it if needed |
| free | releases the specified block of memory back to the system |

Table 1.1: Description of the standard C dynamic memory management functions

### 1.1.3 Allocator Schemes

Considering the main mechanism used by an allocator, an allocator may fall in one or more of the following categories. Examples of each category are given. In some cases it is difficult to assign an allocator to a category because it uses more than one mechanism.

- **Sequential Fits:** sequential Fits algorithms are the most basic mechanisms. They search sequentially free blocks stored in a singly or doubly linked list. Examples are first-fit, next-fit, and best-fit. First-fit and best-fit are two of the most representative sequential fit allocators, both of the are usually implemented with a doubly linked list.

- **Segregated free lists:** These algorithms use a set of free lists. Each of these lists store free blocks of a particular predefined size or size range. When a free block is released, it is inserted into the list which corresponds to its size. TLSF [3] is an modified segregated free list allocation scheme, with a runtime complexity of $O(1)$ for both allocation and de-allocation operations.

- **Buddy Systems:** Buddy Systems [4] are a particular case of Segregated free lists. Being $H$ the heap size, there are only $\log_2 H$ lists since the heap can only be split in powers of two. This restriction yields efficient splitting and merging operations, but it also causes a high

memory fragmentation. The Binary-buddy allocator is the most representative of the Buddy Systems allocators, which besides has always been considered as a real-time allocator.

- **Bitmapped fits:** Algorithms in this category use a bitmap to find free blocks rapidly without having to perform an exhaustive search. Bitmaps to keep track of empty lists jointly with bitmap processor instructions are used to speedup search operations.
- **Hybrid Allocators:** Hybrid allocators can use different mechanisms to improve certain characteristics (response time, fragmentation, etc.) The most representative is Doug Leas allocator [5], which is a combination of several mechanisms.

### 1.1.4 Classical Implementations

The implementation of memory management depends greatly upon operating system and architecture. Some operating systems supply an allocator for malloc, while others supply functions to control certain regions of data. The same dynamic memory allocator is often used to implement both malloc and the operator new in C++ . Hence, it is referred to below as the allocator rather than malloc.

Historically implementation of the allocator is commonly done using the heap, or data segment. The allocator will usually expand and contract the heap to fulfill allocation requests with the help of an OS system call sbrk.

The heap method suffers from a few inherent flaws, stemming entirely from fragmentation. Like any method of memory allocation, the heap will become fragmented; that is, there will be sections of used and unused memory in the allocated space on the heap. A good allocator will attempt to find an unused area of already allocated memory to use before resorting to expanding the heap. The major problem with this method is that the heap has only two significant attributes: base, or the beginning of the heap in virtual memory space; and length, or its size. The heap requires enough system memory to fill its entire length, and its base can never change. Thus, any large areas of unused memory are wasted. The heap can get "stuck" in this position if a small used segment exists at the end of the heap, which could waste any magnitude of address space, from a few megabytes to a few hundred. The organization of the memory map is shown in figure 1.1 with the heap expanding upwards, and the stack moving the other direction.

Doug Lea has developed dlmalloc ("Doug Lea's Malloc") [5] as a general-purpose allocator, starting in 1987. The GNU C library (glibc) uses ptmalloc, an allocator based on dlmalloc.

Memory on the heap is allocated as "chunks", an 8-byte aligned data structure which contains a header and usable memory. Allocated memory contains an 8 or 16 byte overhead for the size of the chunk and usage flags. Unallocated chunks also store pointers to other free chunks in the usable space area, making the minimum chunk size 24 bytes.

Unallocated memory is grouped into "bins" of similar sizes, implemented by using a double-linked list of chunks (with pointers stored in the unallocated space inside the chunk).

For requests below 256 bytes (a "smallbin" request), a simple two power best fit allocator is used. If there are no free blocks in that bin, a block from the next highest bin is split in two.

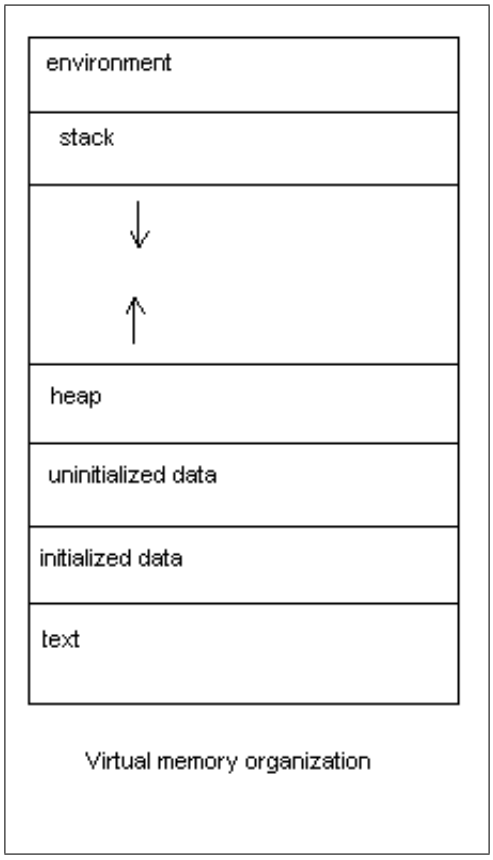


Figure 1.1: Virtual memory organization

For requests of 256 bytes or above but below the mmap threshold, recent versions of dlmalloc use an in-place bitwise trie algorithm. If there is no free space left to satisfy the request, dlmalloc tries to increase the size of the heap, usually via the brk system call.

For requests above the mmap threshold (a "largebin" request), the memory is always allocated using the mmap system call. The threshold is usually 256 KB. The mmap method averts problems with huge buffers trapping a small allocation at the end after their expiration, but always allocates an entire page of memory, which on many architectures is 4096 bytes in size.

### 1.1.5 Problems associated with dynamic memory management

There are a number of problems associated with dynamic memory management, which often lead to runtime errors and introduction of program bugs if careful considerations are not made while using them. A few common errors associated with dynamic memory management are as follows.

- **Not checking for allocation failures:** Memory allocation is not guaranteed to succeed, and may instead return a null pointer. If there's no check for successful allocation implemented,

this usually leads to a crash of the program, due to the resulting segmentation fault on the null pointer dereference.

- **Memory leaks:** Failure to deallocate memory using free leads to build up of non-reusable memory, which is no longer used by the program. This wastes memory resources and can lead to allocation failures when these resources are exhausted.

- **Logical errors:** All allocations must follow the same pattern: allocation using malloc, usage to store data, deallocation using free. Failures to adhere to this pattern, such as memory usage after a call to free (dangling pointer) or before a call to malloc (wild pointer), calling free twice ("double free"), etc., usually causes a segmentation fault and results in a crash of the program. These errors can be transient and hard to debug for example, freed memory is usually not immediately reclaimed by the OS, and thus dangling pointers may persist for a while and appear to work.

Apart from the above mentioned problems, dynamic memory management implementation also suffer majorly due to fragmentation and unbounded response time while the allocation and deallocation operations. These aspects often lead to suboptimal usage of memory and decreased performance. fragmentation is a phenomenon in which storage space is used inefficiently, reducing capacity or performance and often both. Fragmentation is generally of the two following types:

1. **Internal fragmentation:** Often memory is allocated in integral multiples of blocks of certain size. This leads to more memory being allocated than requested, which causes some of the allocated memory to be wasted. This phenomenon is known as internal fragmentation.

2. **External fragmentation:** External fragmentation arises when free memory is separated into small blocks and is interspersed by allocated memory. The consequence of this leads to a condition when an allocation request fails as a large enough contiguous free space is not found, although the total available space is greater than that of the allocation request size. External fragmentation can be calculated by the following expression.

$$\boxed{ExternalMemoryFragmentation = 1 - \frac{LargestBlockofFreeMemory}{TotalFreeMemory}}$$

## 1.2 Related Work

Dynamic memory management is an established problem in computing, but is often a less touched topic in the domain of resource constrained embedded systems and WSN. Many programmers avoid dynamic allocation in many situations, because of perceived space or time cost. In [6] the author reviews about various aspects of memory management in WSN, problems associated with them and the support for memory management in popular WSN OS like TinyOS, Nano RK, MantisOS, SOS, Contiki, LIMEOS. In their work [7], atienza et al. describes a methodology for designing custom memory manager for multimedia and wireless network applications with a reduced memory footprint. An analysis of memory management approaches in order to characterise the tradeoffs across three semantic domains: space, time and a characterisation of memory usage information such as the lifetime of objects in a realtime system is described in [8] . An elegant dynamic memory management allocation scheme for realtime systems TLSF is proposed in [3], with a allocation and de-allocation complexity of $O(1)$, TLSF guarantees a tightly bounded response time, desirable in

realtime applications. In [9] researchers from LSI and Samsung describe a smart dynamic memory manager for embedded system, that predicts long lived allocations from short lived allocations and allocate them to separate regions of the heap, reducing external fragmentation. In [10, 11], Shalan et al. describes a hardware based solution and required OS support for their proposed SoCDMMU for managing global on chip memory of a MPSoC. While most of these research target relatively high end embedded systems featuring powerful SoCs and featuring RAM and Flash space in MB, in [12] Hong et al. proposed a dynamic memory management for the Atmega128l based MICAZ sensor board running Nano-QPlus sensor OS. Besides this various form of support is also available in sensor OSes like MantisOS [13], SOS [14] and Contiki [1].

## 1.3 Motivation

Although a wide variety of research is done on dynamic memory management for embedded and realtime systems, they often do not scale to very low end and low power resource constrained embedded systems. These embedded systems often feature microcontrollers with very limited RAM and Flash space. Similarly on the other hand, dynamic memory management support in WSN OS has their own set of limitations [12]. To aggravate the problem, it is often observed that the API support for dynamic memory management in sensor OS are not compliant with the dynamic memory management API of the standard C library. To cite a few examples, the managed memory allocator in contiki uses a level of indirection while referencing an allocated block to support it's memory compaction feature. Similarly in the dynamic memory manager proposed in [12] the free function takes as argument the pointer to the allocated block, and the size to be freed, instead of only the pointer to the allocated block as in the standard C library's implementation of free.

These aspects, obscure the view of programming and makes it difficult to port applications across multiple sensor platforms. The motivation of this work, is to develop a platform independent, configurable and lightweight dynamic memory manager for severely resource constrained embedded systems used in WSN applications.

# Chapter 2

# Methodology

Applications running on a low end embedded system are not generally subjected to change too often. All the processes that are supposed to run are known at compile time. Since these systems are targeted for a very specific and well defined purpose, most of its operations are predictable and hence the uncertainty factor can be minimized. The runtime variations are mainly due to variable input data, which is why certain class of applications requires dynamic memory management. Knowledge of the allocation and de-allocation request patterns can really provide valuable insight for designing a custom memory allocator for embedded systems.

In this chapter the procedure of collecting allocation and de-allocation traces from a class of real applications and some random allocation and de-allocation traces is described. This is followed by an analysis of the trace information and finally the chapter is concluded by defining implementation independent API wrappers compatible with the standard C dynamic memory management library.

## 2.1 Selection of applications using Dynamic memory allocation

To conduct an unbiased experimental analysis it was important to select a set of real life applications which can be executed on a microcontroller based embedded system and relied on dynamic memory management. The difficulties in the selection of the applications are threefold. Firstly as it is a general tendency to avoid dynamic memory allocation in embedded system development, majority of the libraries or applications targeted for embedded systems don't employ dynamic memory allocation. Secondly the benchmark applications used for the evaluation of dynamic memory management are too heavy to be deployed on a microcontroller based embedded system. Lastly related research such as [12] does not maintain a transparent test suite as the details of the test suite is very limited and cannot be replicated.

Keeping the above mentioned aspects in mind, the applications were chosen with the following requirements and assumptions:

1. The application must use dynamic memory allocation
2. The memory footprint must be small enough to fit within a microcontroller exhibiting a flash size of 128KB — 256KB and a RAM of size 8KB — 32KB

3. The application may have some use in a resource constrained embedded system.

4. Dynamic memory allocation calls made during reading of a file or writing to a file is not considered as a typical microcontroller based system does not exhibit a filesystem.

5. The code is seen as a black box and no attempt is made to analyze or optimize the code in any fashion.

6. All codes are available from popular open source repositories like googlecode, github or sourceforge, such that the process can be easily replicated.

### 2.1.1 EAP-TLS authenticated security protocol with RSA based PKI

The EAP-TLS is the primary test candidate as the requirement for a dynamic memory allocator surged during the development of this protocol during the analysis and development of a secure network authentication protocol for 802.15.4 based wireless sensor nodes. This project uses AxTLS embedded TLS library for 2.1.1 parsing X.509 certificates and various cryptographic method. This library heavily uses dynamic memory management for parsing X.509 certificates and for the RSA algorithm for digital signature and key exchange. Due to IP rights the entire EAP-TLS code cannot be made open, but axTLS embedded SSL [15] is an opensource library. The download link is axTLS

### 2.1.2 FastLZ

FastLZ [16] is a lossless data compression library designed for real-time compression and decompression. It favors speed over compression ratio. Decompression requires no memory. Decompression algorithm is very simple, and thus extremely fast. Due to its speed, FastLZ is very useful for applications that need to save some space without a sacrifice in performance. FastLZ is very small and self-contained and is inspired by Herman Vogt's LZV and Marc Lehmann's LZF algorithms. The download link is FastLZ.

### 2.1.3 Huffman-textfile-compression

Huffman-Textfile-Compressor [17] is a small text file compressor using the popular Huffman encoding scheme. The compression algorithm is more memory intensive and hence only the decompression algorithm of this application was evaluated for a precompressed text. Dynamic memory allocation is used to manage the Huffman tree during the decoding process. The download link is huffman-textfile-compression.

### 2.1.4 SimpleXML 1.0

Simple XML [18] is a tiny and simple to use XML parser written in C. Simplexml is ideal for limited and fast XML parsing for handheld or low power embedded devices. The download link is SimpleXML 1.0.

## 2.2 Generation of Random allocation and dellocation test set

A random allocation and dellocation test set was generated to observe the effect of the proposed dynamic memory allocation scheme when seemingly it is impossible to find any order in the allo-

cation and de-allocation requests. This serves as a control in the experimental setup when it is not possible to customize the dynamic memory allocator.

Figure 2.1 depicts the flowchart of the process and the required data-structures and macro definitions are described in Listing 2.1.

Listing 2.1: random allocation/de-allocation test set generation data structures

```
/*Configures the maximum number of iterations the program should run*/
#define MAX_LIFETIME 300

/*Configures the maximum number of allocations to be made*/
#define MAX_ALLOCATIONS 250

/*configures the maximum allocation request size*/
#define MAX_ALLOC_REQUEST_SIZE 256

/*configures the minimum allocation request size*/
#define MIN_ALLOC_REQUEST_SIZE 4

/*maintains all allocation/de-allocation requests*/
struct allocation_table{
       void *ptr[MAX_ALLOCATIONS];
       int size[MAX_ALLOCATIONS];
       int index;
};
```

## 2.3 Collection of allocation/de-allocation traces

The linux command ltrace was used to take a trace of the malloc, calloc, realloc and free calls from a running application. ltrace takes as argument a comma separated list of function calls to trace and the executable and dumps the trace of the function call to the stderr. For example if we want to trace the malloc and free calls in an executable named executable_binary, then the following linux command dumps the trace information in a file named allocation_trace_dump.

*$ ltrace e malloc, free ./executable_binary 2 >allocation_trace_dump*

A sample output from the trace is shown as follows:

```
malloc(56)                                                    = 0x0910a008
malloc(3)                                                     = 0x0910a048
free(0x0910a048)                                              = <void>
malloc(14)                                                    = 0x0910a058
free(0x0910a058)                                              = <void>
malloc(14)                                                    = 0x0910a058
free(0x0910a058)                                              = <void>
malloc(19)                                                    = 0x0910a058
```

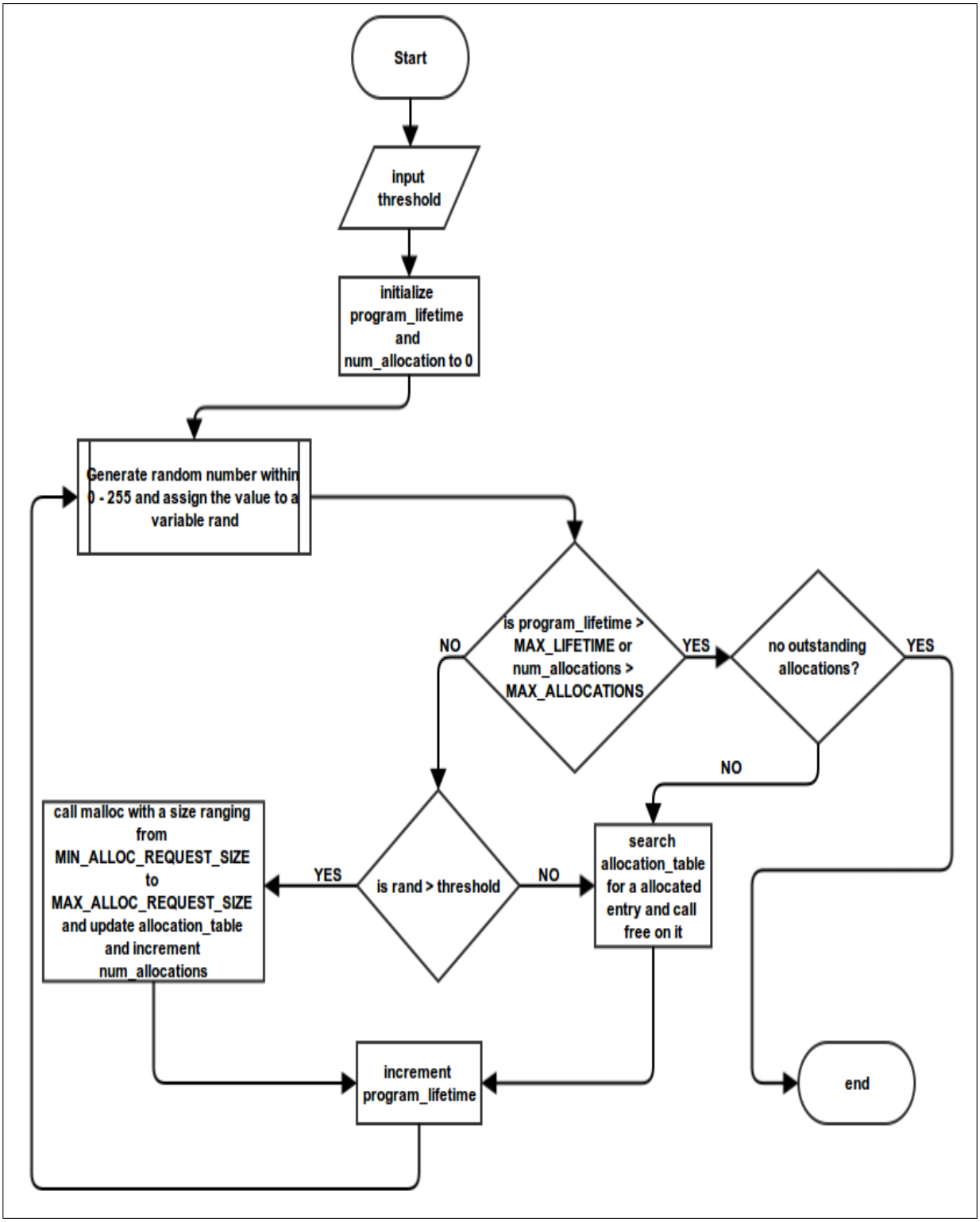


Figure 2.1: flowchart of random allocation/de-allocation test set generation

The obtained allocation trace from the above mentioned step is further processed to obtain the lifetime information corresponding to each allocation. The allocation lifetime is indicative of the duration between an allocation and it's de-allocation. This duration is not measured in units of time, but rather measured in the number of dynamic memory service requests, which can be defined as any call to the dynamic memory management API. For example from the above trace we can conclude that there are total 9 dynamic memory service requests, hence the entire lifetime of the trace is 9.

Listing 2.2 describes the data structures and macros used in the processing of the allocation output trace to generate the allocation size and allocation lifetime and Figure 2.2 describes the flowchart of the same process.

Listing 2.2: data structures and macro definition for processing the memory trace to generate allocation size and allocation lifetime

```
/*configures the maximum number of allocation requests as found in the trace*/
#define MAX_ALLOC 2000

/*maintains the memory requests and lifetime information*/
struct MEM_REQUEST{
       void*  addr;
       uint32_t size;
       uint16_t alive_ctr;
};
static struct MEM_REQUEST mem_request_table[MAX_ALLOC];
```

After this processing step, a two column output is obtained. The first column represents the allocation size and the second column represents the allocation lifetime. A sample output is shown as follows:

```
56                187
3                 1
14                1
14                1
19                159
4                 160
11                155
```

This output is then used for further statistical analysis.

## 2.4 Analysis of allocation/de-allocation traces

In this section the memory allocation and de-allocation traces are analyzed for each application and for the random allocation and de-allocation pattern. The primary purpose here is to identify allocation access patterns, which can be potentially used to design a configurable custom dynamic memory management scheme, which optimizes memory utilization.

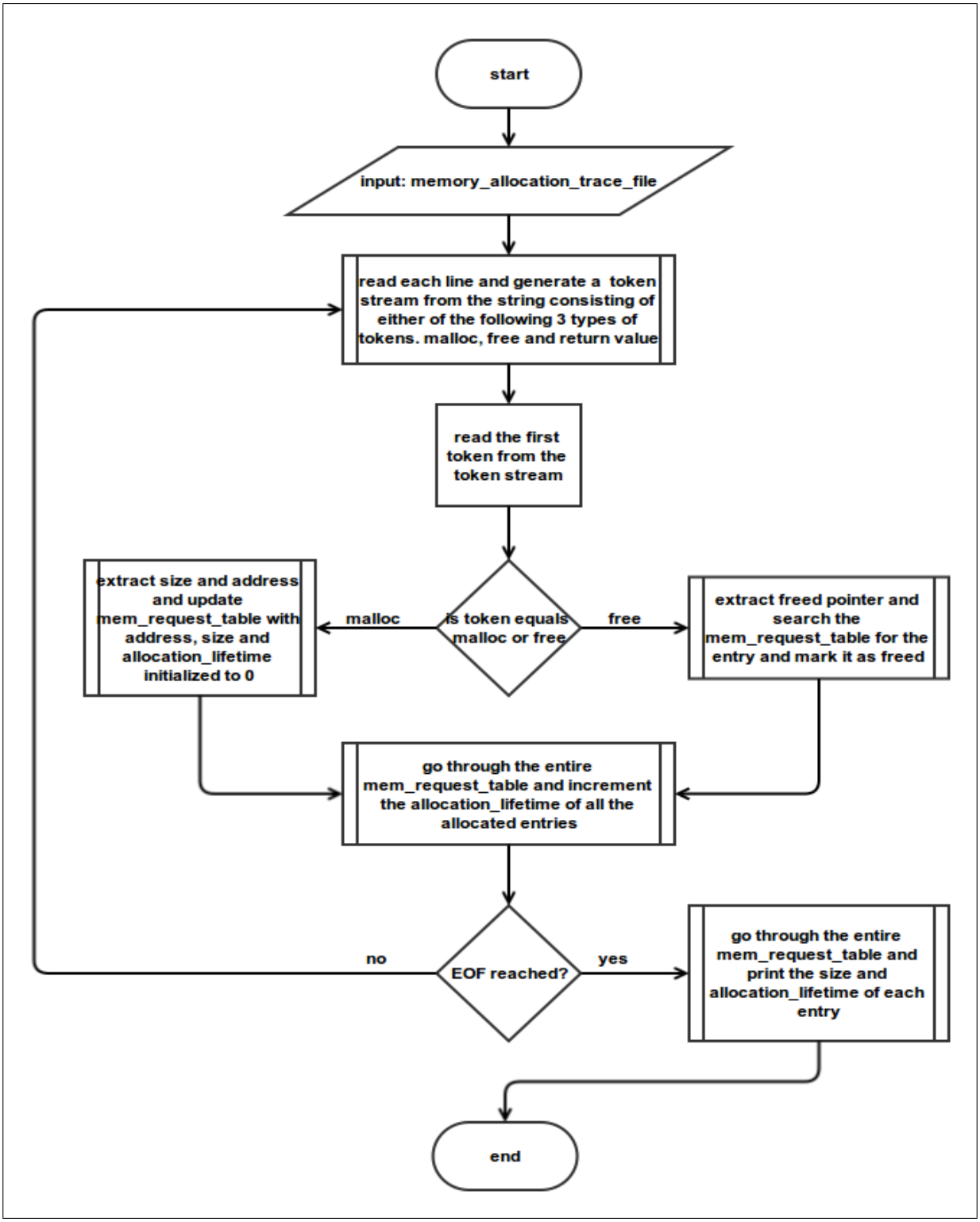


Figure 2.2: flowchart of gathering allocation size and allocation lifetime information from memory trace

### 2.4.1 Trace analysis of AxTLS

Figure 2.3 represents the allocation trace scatter plot from processing of X.509 certificates with 1024 bit RSA. These certificates are of X.509 format, and processing step include parsing the certificates, verifying the certificate with respect to a root authority certificate and then perform a digital signature operation and public key encryption operation. This is the most complex among all the application examples.

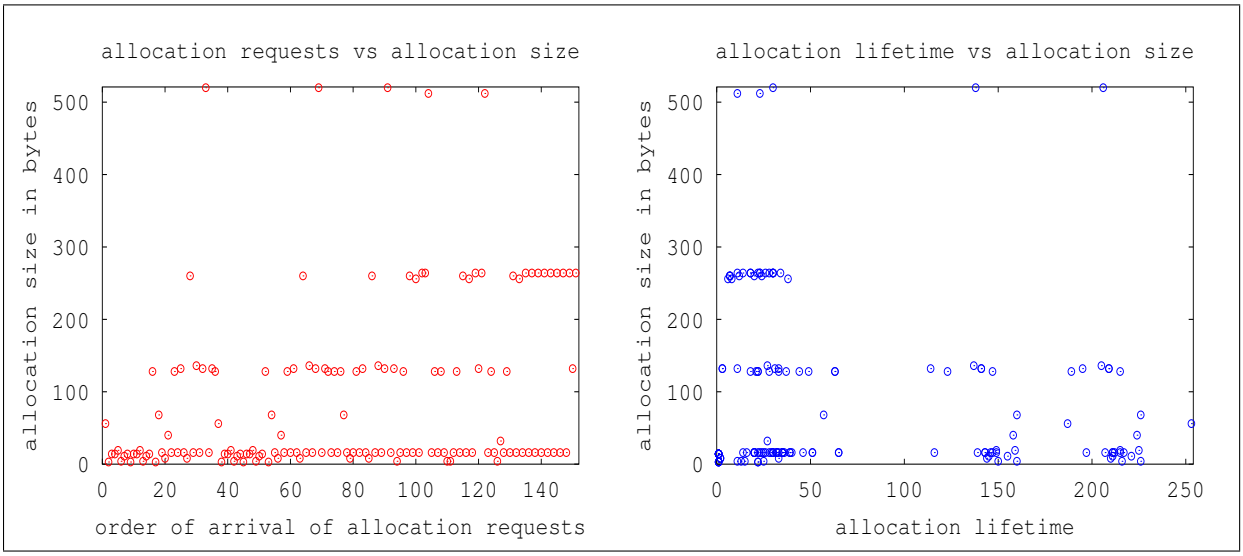


Figure 2.3: allocation scatter plot of AxTLS X.509 RSA certificate processing

From the allocation request vs allocation size scatter plot of figure 2.3 it can be observed that the allocation request sizes is limited to four categories. Most of the requests are below 100 byte, with majority around 32 — 64 bytes, the next is around 128 — 200 bytes. The third category is around 250 bytes and finally there are 5 distinct allocations in the 500 byte region. The allocation lifetime vs allocation request size shows a more or less uniform distribution of allocation request size with respect to their lifetime.

### 2.4.2 Trace analysis of FastLZ

Figure 2.4 represents the allocation trace scatter plot of fastLZ algorithm decompressing a compressed data. The compression is more memory intensive and hence is avoided. From the plot it is very evident that fastLZ has a very simple memory allocation request trace with only 3 allocation requests of 3 sizes. If the plot of allocation request vs allocation size is correlated with allocation lifetime vs allocation size, another aspect of the allocation pattern becomes very apparent, which is that memory is monotonically allocated and then monotonically deallocated. This is a good indication that there can be no issue with fragmentation with this algorithm.

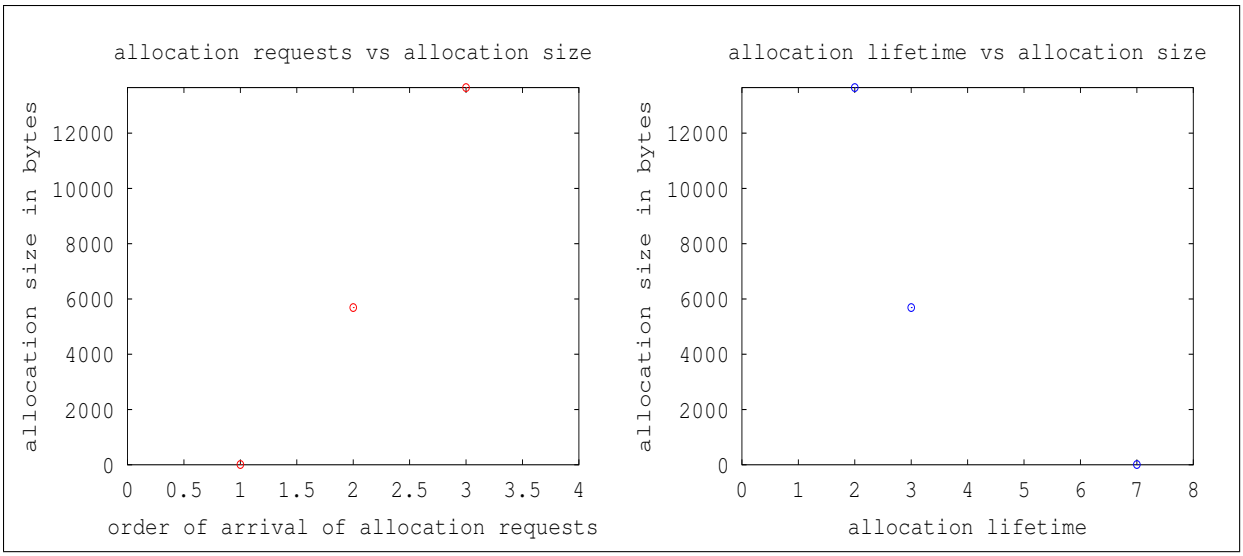


Figure 2.4: allocation scatter plot of FastLZ performing decompression

### 2.4.3 Trace analysis of Huffman-textfile-compression

Figure 2.5 represents the allocation trace scatter plot of huffman-textfile-decompression algorithm. From the figure it is very clear that the allocation size requests are either of 8 byte, 16 byte or 33 byte. The allocation lifetime vs allocation size plot suggests that the 33 byte allocation is very short lived, which is probably because it is continually allocated and deallocated from allocation request 300 onward. The allocation requests of 8 byte and 16 byte are stable allocations with a long lifetime.

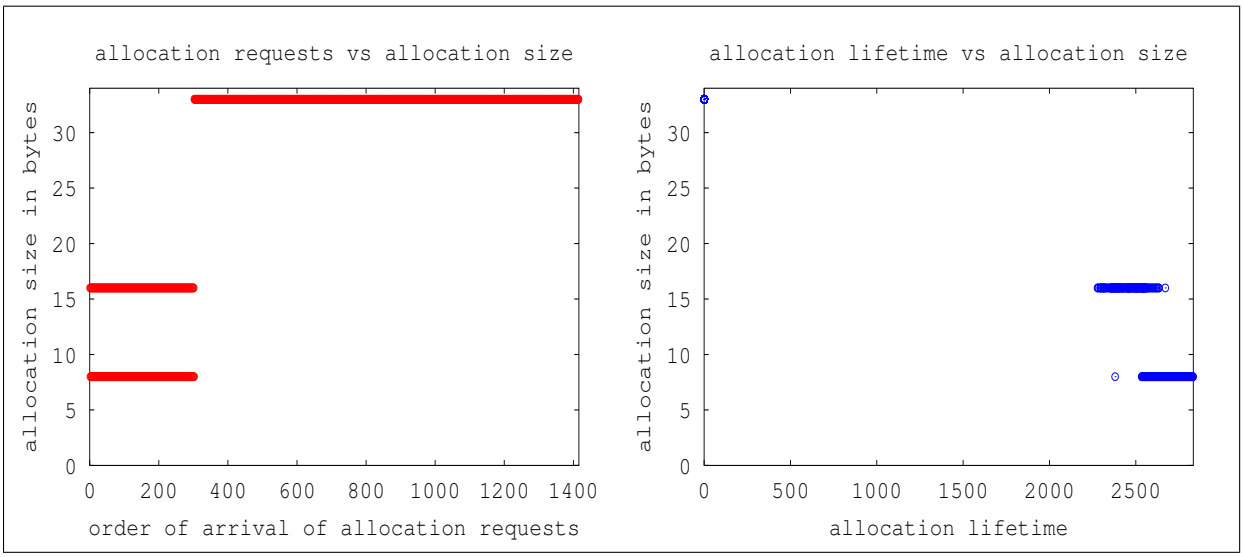


Figure 2.5: allocation scatter plot of huffman based decompression

### 2.4.4 Trace analysis of SimpleXML 1.0

Figure 2.6 represents the allocation trace scatter plot of SimpleXML 1.0. From the allocation request vs allocation size scatter plot it can be concluded that most of the allocation request are small and below 32 bytes. There is 1 allocation above 500 byte. The allocation lifetime vs allocation size plot shows that all allocations are uniformly spread across the entire lifetime axis, however the 500 byte allocation holds a large lifetime.

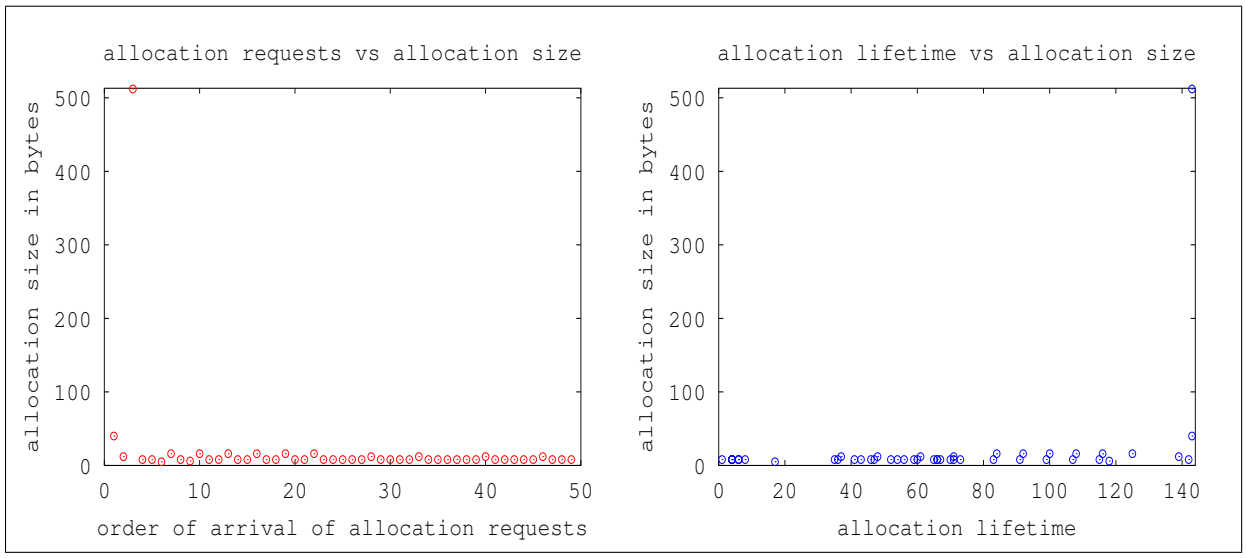


Figure 2.6: allocation scatter plot of SimpleXML 1.0 parsing an XML followed by some query

### 2.4.5 Trace analysis of some random allocation and de-allocations

Figures 2.7, 2.8 and 2.9 shows allocation request scatter plots of the synthetic random allocation trace discussed in 2.2. From the allocation request vs allocation size scatter plot nothing viable is observable, as the distribution is random. However the allocation lifetime vs allocation size displays subtle features across the 3 traces. In Figure 2.7 the allocation lifetime is neither too large or too low, and the distribution is even. Figure 2.8 has a large allocation lifetime for most of its allocation and Figure 2.9 has a low allocation lifetime across most of it's allocations.

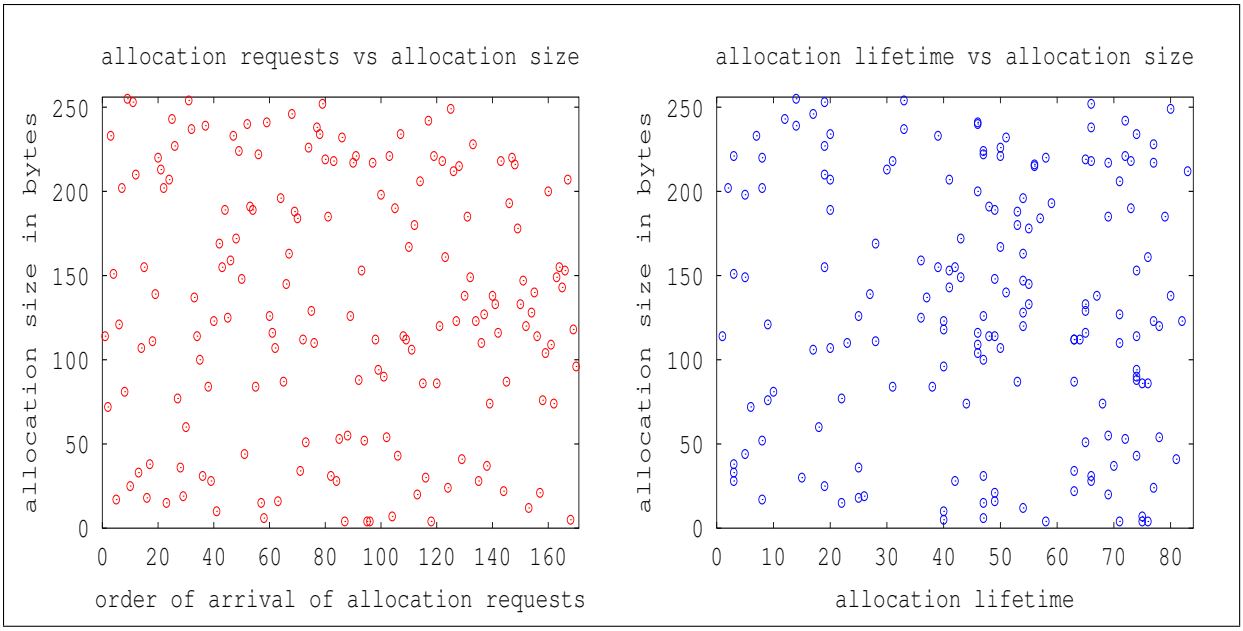


Figure 2.7: allocation scatter plot of random allocation/de-allocation trace taken with threshold value of 50 see flowchart in 2.1

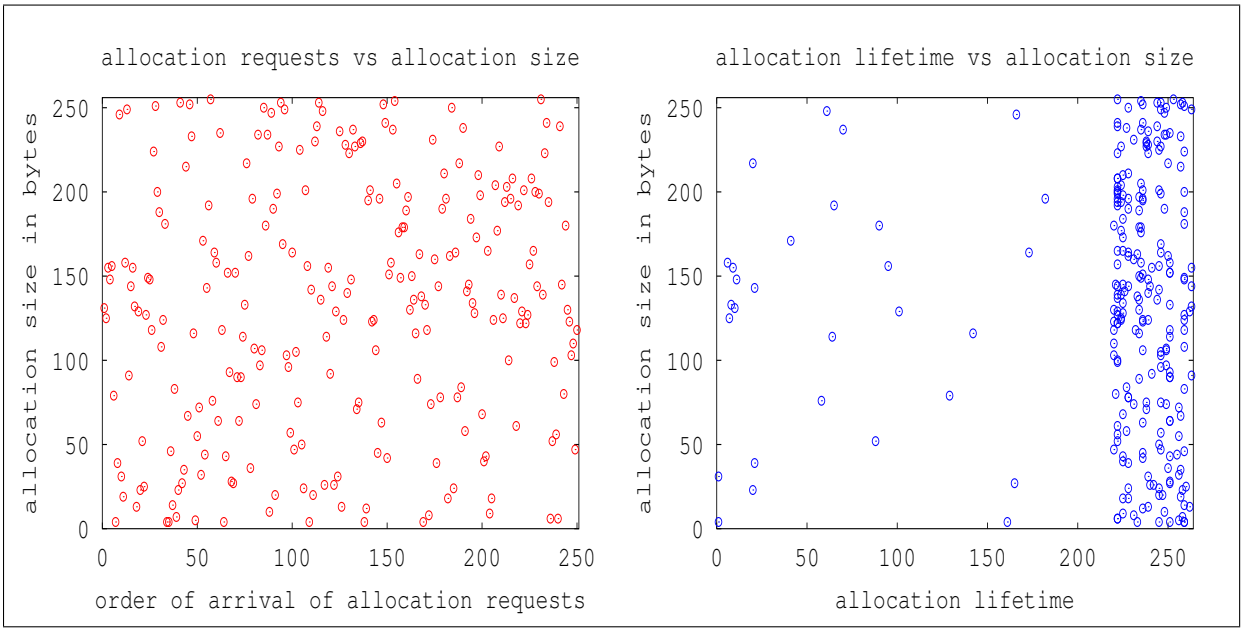


Figure 2.8: allocation scatter plot of random allocation/de-allocation trace taken with threshold value of 120 see flowchart in 2.1

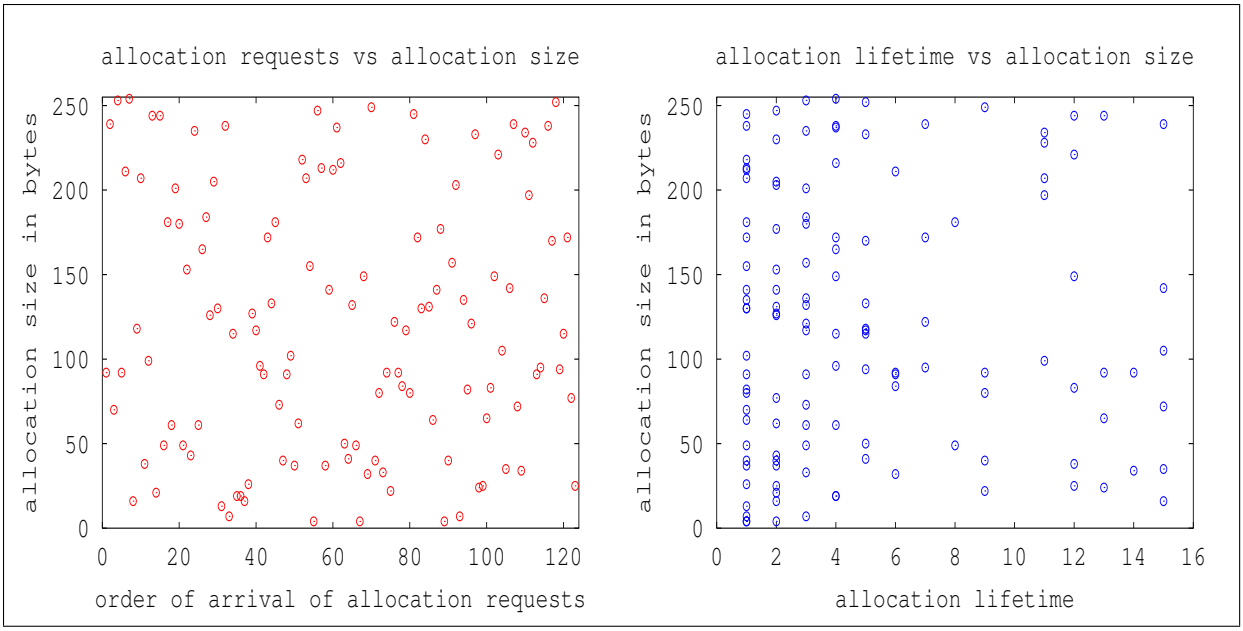


Figure 2.9: allocation scatter plot of random allocation/de-allocation trace taken with threshold value of 20 see flowchart in 2.1

## 2.5 Hypothesis

The scatter plots of the allocation traces of the real applications reveals that the allocation request size are not random, but rather is restricted to a few levels. Mostly with the applications that were analyzed the number of levels is generally 3 — 4. This is an interesting observation, and with the objective of reducing external fragmentation the following can be hypothesized.

*If most of memory allocation request are restricted to a few category of size, then external fragmentation may be reduced using a banked memory layout, where the entire memory is divided into a few fixed sized sections, and similar sized memory allocation requests are serviced in a bank assigned to that category with the highest priority. This in effect will club like sized blocks together thus reducing a chance of external fragmentation.*

## 2.6 Requirements of API for ease in portability

Since most embedded systems programming is done in C, the dynamic memory management API in the standard C library is chosen as the standard interface. Table 2.1 tabulates the function name and the type signatures of the standard C library dynamic memory management API and the function name and type signatures required to be supported by the dynamic memory allocation schemes proposed in this dissertation.

| **Standard C library API** | **Proposed API** |
|---|---|
| void* malloc(size_t size) | void* st_malloc(size_t size) |
| void* calloc(size_t size) | void* st_calloc(size_t size) |
| void* realloc(void* old_ptr, size_t size) | void* st_realloc(void* old_ptr, size_t size) |
| void free(void* ptr) | void st_free(void* ptr) |

Table 2.1: dynamic memory management API type signatures

The requirements to the API described in this section allows for code portability as it requires minimal changes to an existing code base using standard C library dynamic memory allocation API to use the embedded dynamic memory allocation mechanisms described in this work.

# Chapter 3

# Design & Implementation

In this chapter, 3 different dynamic memory management schemes are discussed. These schemes are progressively developed by identifying limitations in each design and subsequent addition of features in the next. The design of the data structures in all of the scheme revolves around the following two design objectives.

1. The data structures used should be such that they should support an API which is compliant with the dynamic memory management API of the standard C library
2. The data structures should be small enough so that the overhead of maintaining metadata can be kept at minimum.

In order to verify the hypothesis framed in 2.5, the proposed schemes can be configured to use multiple memory banks, each of a specific size and serving allocation request within a particular size domain.

## 3.1 Architecture 1

This architecture is based on a bitmapped memory allocation scheme. The basic concept is very simple and has also been used widely before [19, 20]. The memory is divided into small blocks of a particular size, and every bit in the bitmap identifies one such block. The number of bits in the bitmap equals the number of blocks of the memory and the total amount of memory is given by the product of the block size and the number of blocks. If a bit is set to 1 then it is assumed that it's corresponding block is allocated and if it is 0 then the corresponding block is free.

However in the proposed implementation, in order to have a standard C library compliant API, two bitmaps are used instead of one. These two bitmaps are known as the block status bitmap and the End marker bitmap respectively. Figure 3.1 represents these two bitmaps, where the block size is 8 bytes, hence the described 64 bit bitmap manages 512 bytes of memory.

The block status bitmap just maintains the current allocation status of it's corresponding block. The end marker bitmap tracks how many blocks are allocated on a allocation request and is used during the free operation to determine the number of serial blocks required to be freed. Details on the data structure and the algorithm of the allocation and de-allocation operations are described in the next section.

### 3.1.1 Data Structures and algorithms

Figure 3.1 illustrates the 64 bit wide block status bitmap and the end marker bitmap. In the proposed scheme the block size is chosen to be 8 bytes hence a 64 bit bitmap can manage a 512 byte of memory.

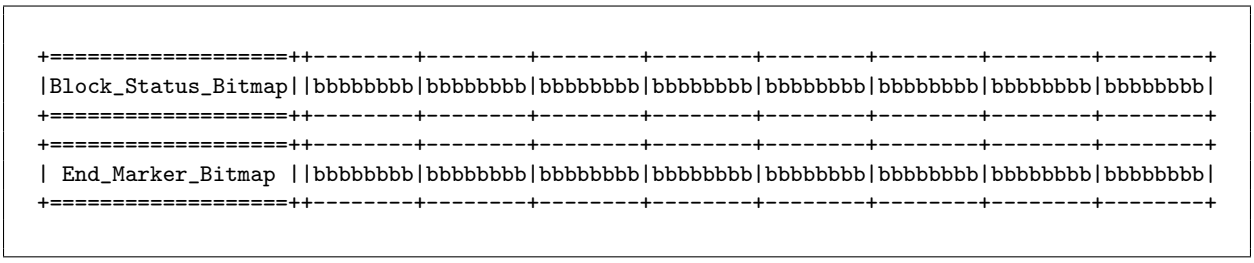


Figure 3.1: Bitmap structure for Managing a 512 byte memory frame

A 512 byte of managed memory block is known as a memory frame, and it is the smallest manageable unit. The rest of the memory is configured in integral multiples of 512 byte frames. For ease of operation a single memory request is limited to a maximum size of 512 bytes.

The allocation procedure is described in Algorithm 1 and the de-allocation procedure is described in Algorithm 2. This scheme has a constant allocation overhead of 2 bytes for maintaining 64 bytes of data with a block size of 8 bytes, besides a per allocation overhead of 4 bytes for the pointer that stores the return address.

The worst case runtime complexity of the allocation and the de-allocation algorithm is $O(n)$ since it requires a linear search of the bitmap to find suitable block for allocation, as well as a linear search to locate the end marker bitmap during the de-allocation process. Figure 3.2 illustrates the state of the bitmap after 3 successive allocation requests of 18 bytes, 24 byte and 8 byte and Figure 3.3 illustrates the state of the bitmap after a de-allocation request for the 24 byte block.

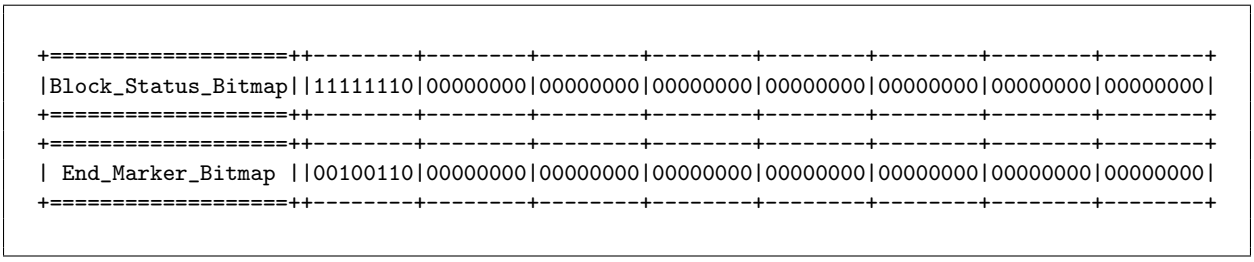


Figure 3.2: State of bitmap after allocation requests of 18 byte, 24 byte and 8 byte

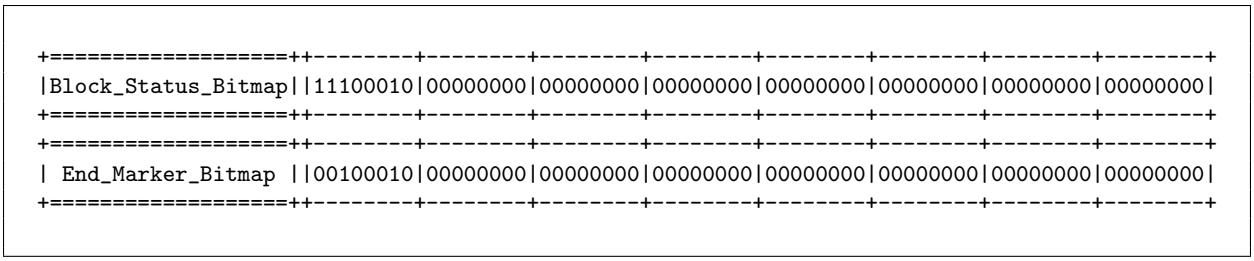


Figure 3.3: State of bitmap after the de-allocation of the 24 byte block

**Input**: An integer *req_size* bytes of allocation request
**Output**: Pointer to the allocated block of memory or NULL if allocation fails
$allocation\_size \leftarrow req\_size$;
**if** $allocation\_size > MEMORY\_FRAME\_SIZE$ **then**
    **return** *NULL*;
**end**
Select proper memory bank based on *allocation_size*;
$num\_banks\_visited \leftarrow 0$;
$N \leftarrow \lceil size/BLOCK\_SIZE \rceil$;
**while** $num\_banks\_visited < NUM\_BANKS$ **do**
    $n \leftarrow 0$;
    **foreach** *bit in the bitmap* **do**
        **if** *bit is set* **then**
            $n \leftarrow n + 1$;
            **if** $n = N$ **then**
                set all the $n$ status bits;
                set the $n_{th}$ end marker bit;
                **return** *start address of the allocated memory block*;
            **end**
        **else**
            $n \leftarrow 0$;
        **end**
    **end**
    Select the next memory bank;
    $num\_banks\_visited \leftarrow num\_banks\_visited + 1$;
**end**
**return** *NULL*;

**Algorithm 1:** The bitmapped allocation algorithm of Architecture 1

**Input**: A Pointer *ptr* to the allocated block of memory
**Result**: Free the allocated memory pointed by *ptr*
$bitmap\_index \leftarrow \lfloor (ptr - memory\_bank\_base\_address)/BLOCK\_SIZE \rfloor - 1$;
**repeat**
    $bitmap\_index \leftarrow bitmap\_index + 1$;
    $block\_status\_bitmap[bitmap\_index] \leftarrow 0$;
**until** $end\_marker\_bitmap[bitmap\_index] \neq 1$;
$end\_marker\_bitmap[bitmap\_index] \leftarrow 0$;

**Algorithm 2:** The bitmapped de-allocation algorithm of Architecture 1

### 3.1.2 Advantages & Disadvantages

The advantages of this scheme are as follows:

1. Fixed metadata overhead for managing the entire pool
2. Simple scheme hence easy to implement

The disadvantages of this scheme are as follows:

1. Large block size of 8 byte leads to internal fragmentation and reducing the block size increases the metadata overhead
2. Configuring the memory manager with different parameters is not easy.
3. Large runtime, as linearly searching the bitmap is computationally expensive
4. The upper limit on the size of an allocation request to the frame size is a limiting factor.

## 3.2 Architecture 2

The bitmapped memory management scheme, as discussed in section 3.1, presents certain design flaws, which renders it impractical for a number of cases. One of the major design flaws is the condition where it fails to allocate memory when the request size exceeds the frame size, even when there is enough memory to service the request. Other problems includes limited reconfigurability, internal fragmentation and large runtime.

The architecture discussed in this section attempts to mitigate the above mentioned problems, with a new design which maintains the entire memory as a linked list of allocated and free blocks. This architecture attempts to maintain the metadata in the managed memory pool itself and is more configurable than it's previous counterpart.

### 3.2.1 Data Structures and algorithms

Figure 3.4 illustrates the layout of managed memory along with the memory descriptor header. The memory descriptor header is only 2 bytes in size, so this scheme will incur a memory management overhead of 2 bytes for storing the memory descriptors besides the 4 byte pointer used to store the return address of the allocated block of memory.

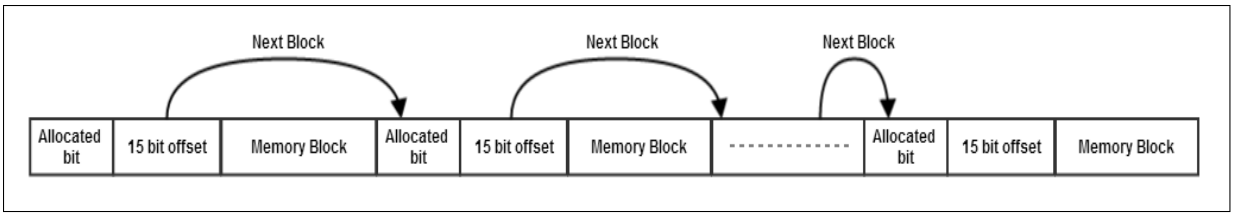


Figure 3.4: Memory layout and Memory descriptor representing 1 allocation bit and 15 bit for offset

The memory descriptor has 2 parts:

1. **Allocated Bit:** The allocated bit determines whether a block is allocated or free. if the allocated bit is set then it signifies that the block is allocated or else the block is free.
2. **15 bit Offset:** This field determines the size of the allocated memory in terms of number of minimum allocation units. If the minimum allocation unit is set to 1 there is no internal fragmentation.

The choice of the 15 bit wide offset is based on a very practical consideration that, microcontrollers often have a RAM space in the range of 1KB — 100KB. Out of this the entire RAM is never used for dynamic memory purposes, as RAM is also accounted for stack space and other static data in the bss section. With a 15 bit wide offset and a block size of 1 byte, each descriptor has an upper limit to account for $2^{15}$ bytes or 32KB of memory, which is within practical boundaries. Besides more memory can be easily accounted for by increasing the block size.

Initially there is only one large free block, in the linked list. The head of the linked list is given by the base address of the managed memory block. When a allocation request arrives, the linked list is traversed in order to search a large enough free block to service the request. The first-fit scheme is used during the search process, and contiguous smaller free blocks are coalesced to form a larger free block. Similarly when a block is found which is larger than what is requested, the free block is split to form a allocated block and a smaller free block. This concept is widely used in buddy systems [4]. The de-allocation process is much simpler. Given the pointer to an allocated block of memory the de-allocation process simply resets the allocted bit in the memory descriptor of the memory block. The allocation process is described in Algorithm 3 and the de-allocation process is described in Algorithm 4.

The worst case runtime complexity of the allocation algorithm is $O(n)$, since the allocation depends on a linear search of the linked list. Although the performance of this algorithm will be still better than it's bitmapped counterpart as the bitmapped version required many bitwise operations during the linear search. The de-allocation operation however has a worst case complexity of $O(1)$ as it comprises a single reset operation on the allocated bit.

```
Input: An integer req_size bytes of allocation request
Output: Pointer to the allocated block of memory or NULL if allocation fails
allocation_size ← req_size;
if allocation_size > LARGEST_MEMORY_BANK_SIZE then
    return NULL;
end
Select proper memory bank based on allocation_size;
num_banks_visited ← 0;
while num_banks_visited < NUM_BANKS do
    allocated ← false; size_index ← 0;
    mem_descriptor ← startaddress(CURRENT_MEMORY_BANK);
    while !allocated and size_index < CURRENT_BANK_SIZE do
        if mem_descriptor.allocated_bit is set then
            size_index ← size_index + SIZE(mem_descriptor) + 2;
            mem_descriptor ← NEXT_DESCRIPTOR(mem_descriptor);
        else
            curr_mem_descriptor ← mem_descriptor;
            mem_descriptor ← NEXT_DESCRIPTOR(mem_descriptor);
            while mem_descriptor.allocated_bit is clear and
            size_index < CURRENT_BANK_SIZE do
                curr_mem_descriptor ← MERGE(curr_mem_descriptor, mem_descriptor);
                mem_descriptor ← NEXT_DESCRIPTOR(curr_mem_descriptor);
            end
            if SIZE(curr_mem_descriptor) ≥ allocation_size then
                if SIZE(curr_mem_descriptor) > allocation_size then
                    curr_mem_descriptor ← SPLIT(curr_mem_descriptor, allocation_size);
                end
                curr_mem_descriptor.allocated_bit ← true;
                allocated ← true;
                mem_descriptor ← curr_mem_descriptor;
            else
                size_index ← size_index + SIZE(mem_descriptor) + 2;
                mem_descriptor ← NEXT_DESCRIPTOR(mem_descriptor);
            end
        end
    end
    if !allocated then
        Select the next memory bank;
        num_banks_visited ← num_banks_visited + 1;
    else
        return mem_descriptor + 2;
    end
end
return NULL;
```

**Algorithm 3:** The allocation algorithm of Architecture 2

**Input**: A Pointer *ptr* to the allocated block of memory
**Result**: Free the allocated memory pointed by *ptr*
$mem\_descriptor \leftarrow ptr - 2$;
$mem\_descriptor.allocated\_bit \leftarrow false$;
**Algorithm 4:** The de-allocation algorithm of Architecture 2

### 3.2.2 Advantages & Disadvantages

The advantages of this scheme are as follows:

1. Highly configurable
2. Supports a block size of 1 byte hence no internal fragmentation
3. Low memory management overhead
4. Better response time and the complexity of free operation is $O(1)$.

The disadvantages of this scheme are:

1. Suffers from external fragmentation.
2. Does not supports memory compaction.

## 3.3 Architecture 3

The dynamic memory management scheme that will be discussed in this section is pretty much similar to the scheme discussed in section 3.2. This scheme attempts to add memory compaction feature and hence eliminate the external fragmentation problem. The design of this scheme is partly inspired from the managed memory allocator in Contiki [1], which too employs memory compaction.

The basic problem when employing memory compaction is that, as the contents of the memory is moved around the references to the allocated blocks in the application program becomes invalid. Thus it is required to update these references whenever a compaction operation is performed. The managed memory allocator API in contiki resolves this problem by introducing a level of indirection and before referencing an allocated memory block, the referencing pointer should be updated using a macro.

The scheme that will be described in this section attempts to automate this process of updating the pointer references. In order to integrate this feature, few modifications must be added to the API requirements as defined in Table 2.1. These modifications are not radical and although absolute compliance with the standard C API is violated, they still can be related closely and interchanged with ease. Table 3.1 tabulates modifications to the original proposed API. The main difference is that in all the allocation functions, return by value is replaced by return by address. The prime idea is that, the memory descriptor will have one extra field to store the reference of the pointer which will in term store the reference of the allocated block in the application program.

This will enable the allocator to update the pointer references automatically during the defragmentation process.

| Proposed API | Modified API |
|---|---|
| void* st_malloc(size_t size) | void st_malloc(void** ref, size_t size) |
| void* st_calloc(size_t size) | void st_calloc(void** ref, size_t size) |
| void* st_realloc(void* old_ptr, size_t size) | void st_realloc(void** ref, void* old_ptr, size_t size) |
| void free(void* ptr) | void st_free(void* ptr) |

Table 3.1: modified dynamic memory management API type signatures

### 3.3.1 Data Structures and algorithms

Figure 3.5 represents the layout of the memory and the memory descriptor used in this scheme. The memory descriptor comprises two fields of 16 bit. The first field represents the size of the allocated block in bytes. With 16 bits, each allocation can be of maximum $2^{16}$ bytes or 64KB. The second field holds the offset of the pointer to the pointer which holds the reference of the allocated memory block in the application code from the base RAM address. This eliminates the storage of redundant information.

During the defragmentation process the second field is added to the RAM base address to obtain the absolute address of the pointer holding the allocation reference, and all invalid references can be updated automatically without involving any level of indirection. Information regarding the RAM base address is often found in the microcontroller datasheet or the same can be gathered from the linker script. Shown below in listing 3.1, is an example of a portion of linker script defining the memory map of a stm32w microcontroller.

Listing 3.1: linker script of stm32w showing the memory map (source: contiki)

```
MEMORY
{
  RAM_region (xrw) :       ORIGIN = 0x20000000,         LENGTH = 16K
  ROM_region (xr) :        ORIGIN = 0x08000000,         LENGTH = 256K
  FIB_region (ra) :        ORIGIN = 0x08040000,         LENGTH = 2K
}
```

Algorithm 5 shows the allocation procedure and Algorithm 6 shows the de-allocation procedure in this scheme of memory management. Since this scheme uses memory compaction, no banking scheme is employed in this case. The worst case runtime complexity of allocation algorithm is $O(1)$ since, allocation does not include any search, and is always done from free_memory_offset if enough memory is available. The de-allocation algorithm however has a complexity of $O(n)$ since, the defragmentation operation requires pushing memory down as well as a scan through the allocated blocks in order to update the references.

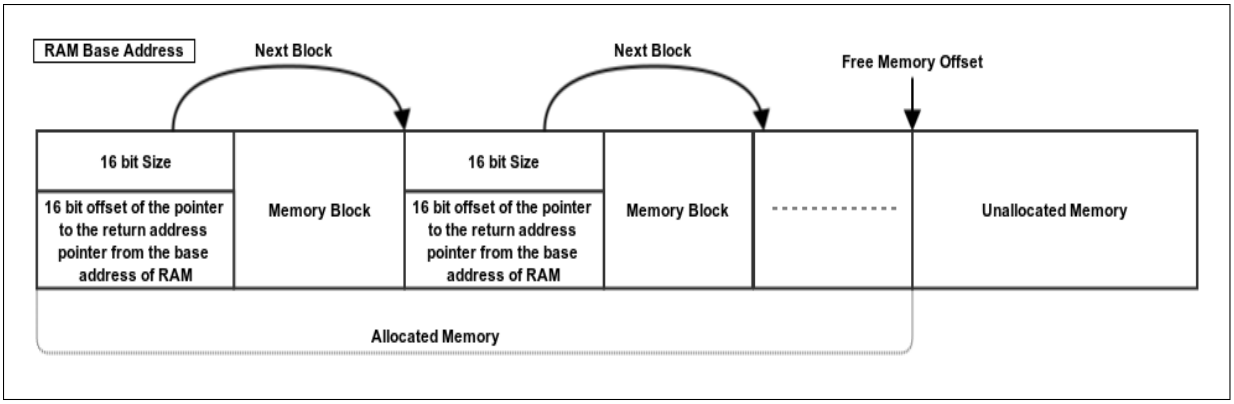


Figure 3.5: Memory layout and Memory descriptor featuring an additional 16 bit field which can be used for the addition of memory compaction feature

**Input**: An integer $req\_size$ bytes of allocation request, A reference of the pointer $ref$ which will hold the start address of the allocated block
**Result**: The pointer referenced by $ref$ is assigned the pointer to the allocated block of memory or NULL if allocation fails
$allocation\_size \leftarrow req\_size$;
$ref\_offset \leftarrow ref - RAM\_BASE\_ADDRESS$;
**if** $allocation\_size > available\_memory$ **then**
  $DEREFERENCE(ref) \leftarrow NULL$;
**end**
$mem\_descriptor \leftarrow free\_memory\_offset$;
$mem\_descriptor.size \leftarrow allocation\_size$;
$mem\_descriptor.ref \leftarrow ref\_offset$;
$DEREFERENCE(ref) \leftarrow free\_memory\_offset + 4$;
$available\_memory \leftarrow available\_memory - (allocation\_size + 4)$;
$free\_memory\_offset \leftarrow free\_memory\_offset + (allocation\_size + 4)$;

**Algorithm 5:** The allocation algorithm of Architecture 3

**Input**: A Pointer $ptr$ to the allocated block of memory
**Result**: Free the allocated memory pointed by $ptr$ and defragment the memory
$mem\_descriptor \leftarrow ptr - 4$;
$available\_memory \leftarrow availble\_memory + mem\_descriptor.size$;
$current_offset = ptr - 4$; $src = ptr + mem\_descriptor.size$;
$len = free\_memory\_offset - (ptr + mem\_descriptor.size)$;
$free\_memory\_offset \leftarrow free\_memory\_offset - (mem\_descriptor.size + 4)$;
MEM_MOVE($current\_offset$,$src$,$len$);
**foreach** *allocated block in current_offset to free_memory_offset* **do**
  update the allocation references
**end**

**Algorithm 6:** The de-allocation algorithm of Architecture 3

### 3.3.2 Advantages & Disadvantages

Advantages of this scheme are as follows:

1. Has Zero internal fragmentation.
2. Has Zero external fragmentation.
3. Low overhead for storing the memory descriptors.

The disadvantages of this scheme are as follows:

1. Not 100% compliant with the dynamic memory management API of the standard C library
2. The pointer used for referencing the allocated memory must be declared with the volatile keyword, as their value may be changed any time, by an external entity.
3. The pointer used for referencing the allocated memory must be passed by reference to another function or part of the program if that is ever required. Pass by value of these pointers should be strictly prohibited as it may lead to invalid memory access.

# Chapter 4

# Results & Discussion

In this chapter details about the experimental results and analysis is presented. The hardware platform used for testing and profiling is introduced. This is followed by results, which is divided in two parts, the first presents the results on the impact of memory banking on external fragmentation and the second part deals with the comparative analysis of the proposed memory manager with two other opensource dynamic memory managers. The results are followed by an analytic discussion based on the results. The chapter concludes with the inferences drawn from the results and discussions.

## 4.1 Hardware platform

Architecture 1 and Architecture 2 discussed in chapter 3 is implemented and tested on the following platforms:

1. STM32W based MBXXX board. Figure 4.1. Relevant specification of the STM32W is as follows:
    - 256KB of Flash
    - 16KB of RAM
    - ARM Cortex M3 clocked @ 24MHz
2. Microblaze 32 bit softcore processor based system, synthesized on a Xilinx FPGA. Relevant configuration of the softcore processor is as follows:
    - 32KB of Flash
    - 32KB of RAM
    - Microblaze clocked @ 100MHz
3. x86 processor based general purpose PC

Architecture 3 was just conceptualized at the time of writing this report and hence is not implemented or tested, and the results being discussed is based on architecture 2. Profiling information was collected from PC using x86 timestamp counter(RDTSC), and from STM32W using the standard ARM SysTick timer. Although the results were similar, the profiling done on STM32W bare

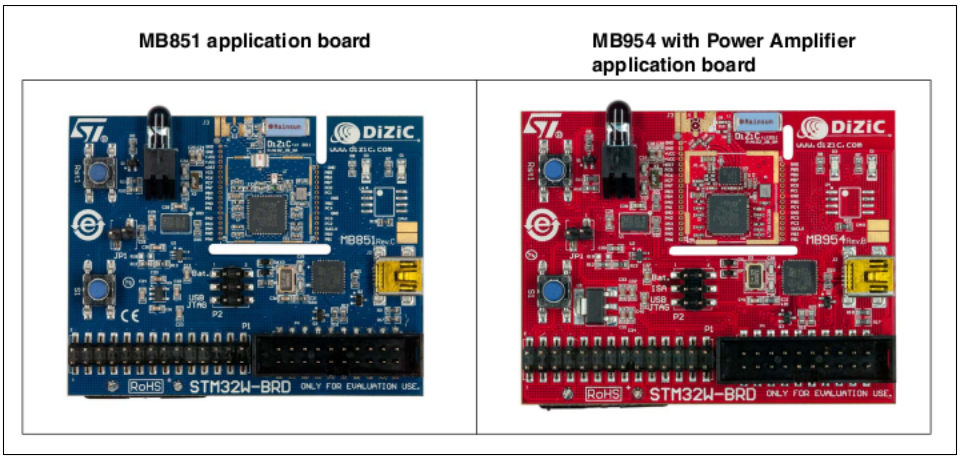


Figure 4.1: STM32W based MBXXX boards by Dizic

metal code was more accurate and consistent than that collected using a PC. The minute inconsistency observed while profiling on the PC was primarily because of the scheduling activity performed by the OS. Thus all results and information in the following section is collected using a MB851 board.

## 4.2 Comparision of average and maximum external fragmentation with and without memory banking

Table 4.1 compares the average external fragmentation and table 4.2 compares the maximum external fragmentation for single banked, double banked and triple banked approach, in order to verify the hypothesis formulated in section 2.5. The plots for the same are shown in Figure 4.2 and 4.3.

| | **st_memmgr with 1 Bank** | **st_memmgr with 2 Bank** | **st_memmgr with 3 Bank** |
|---|---|---|---|
| **AxTLS** | 0.041404 | 0.363028 | 0.552681 |
| **FastLZ** | 0.000 | 0.097475 | 0.097475 |
| **Huffman-textfile-compression** | 0.000 | 0.187293 | 0.550673 |
| **SimpleXML** | 0.00079 | 0.366154 | 0.594156 |
| **Random 1** | 0.019752 | 0.455656 | 0.601536 |
| **Random 2** | 0.025097 | 0.249801 | 0.390301 |
| **Random 3** | 0.00187 | 0.4012887 | 0.431280 |

Table 4.1: Average External Fragmentation

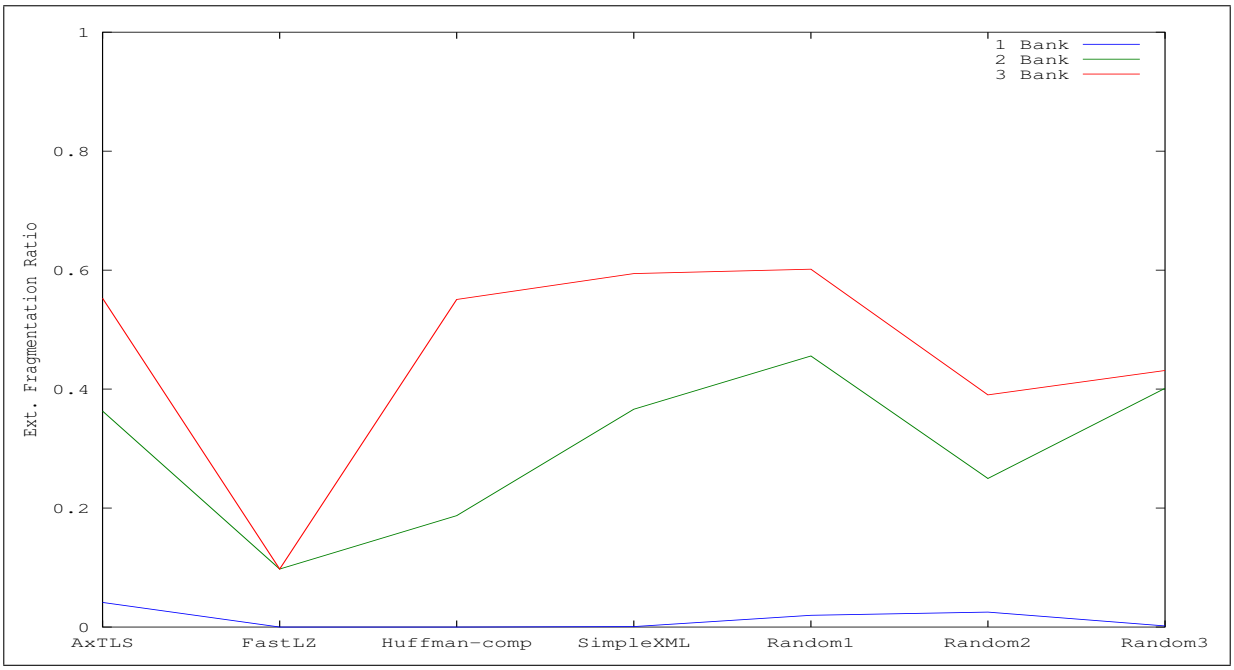


Figure 4.2: comparision of average external fragmentation using 1,2 and 3 memory banks

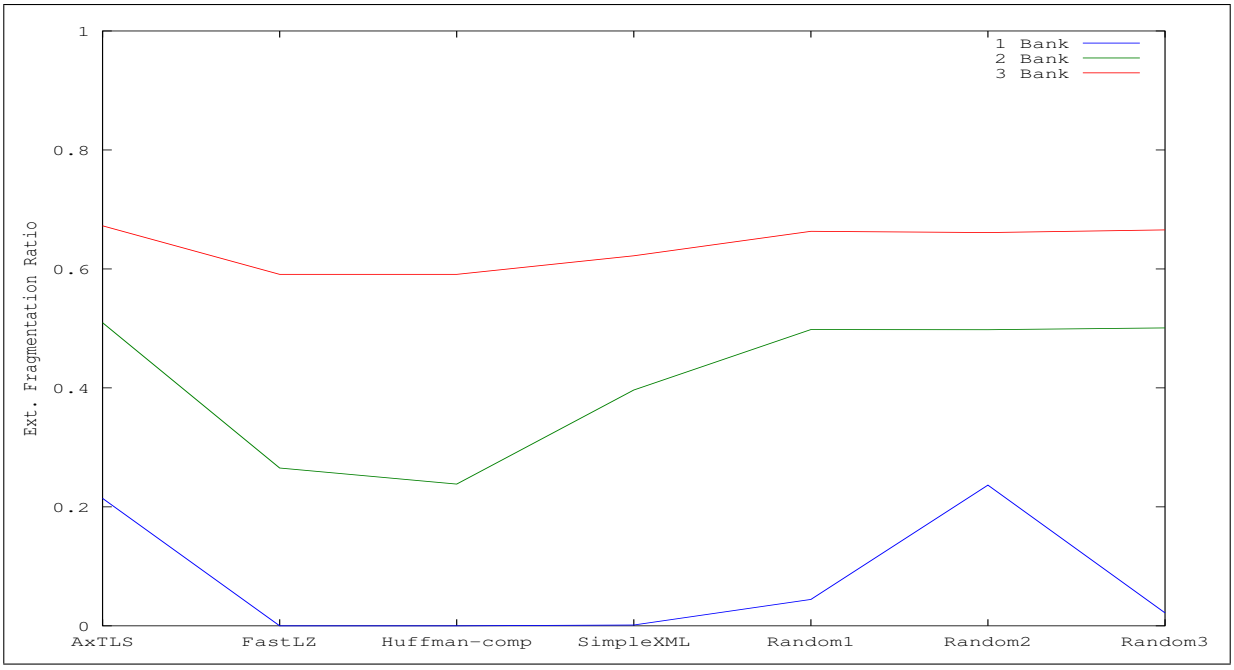


Figure 4.3: comparision of maximum external fragmentation using 1,2 and 3 memory banks

| | st_memmgr with 1 Bank | st_memmgr with 2 Bank | st_memmgr with 3 Bank |
|---|---|---|---|
| **AxTLS** | 0.214336 | 0.509547 | 0.672419 |
| **FastLZ** | 0.000 | 0.265339 | 0.590832 |
| **Huffman-textfile-compression** | 0.000 | 0.238283 | 0.590832 |
| **SimpleXML** | 0.00138 | 0.396369 | 0.622029 |
| **Random 1** | 0.044414 | 0.498105 | 0.663269 |
| **Random 2** | 0.236559 | 0.497801 | 0.661023 |
| **Random 3** | 0.021722 | 0.500823 | 0.665595 |

Table 4.2: Maximum External Fragmentation

## 4.3 Performance comparision with heapLib and memmgr

This section compares the performance of the proposed dynamic memory manager described in architecture 2 along with the following two open source dynamic memory managers:

1. **heapLib 1.1:** a simple, lite and easy to use heap management library for embedded systems [21].
2. **memmgr:** a fixed-pool memory allocator [22].

The performance evaluation is done based on the following two metrics:

1. **cost metric:** The minimum memory required to successfully execute a application as described in [23]
2. **average execution time:** the average execution time taken by the allocation and deallocation operation taken over the entire run of the application.

### 4.3.1 Results based on cost metric

Table 4.3 tabulates the minimum memory in units of 256 byte blocks, required to execute one of the listed memory traces without failing.

| | AxTLS | SimpleXML | Huffman-Compression | Random 1 | Random 2 |
|---|---|---|---|---|---|
| **heapLib** | 7424 | 3072 | 12544 | 6144 | 5376 |
| **memmgr** | 7680 | 3584 | 16896 | 6400 | 5632 |
| **st_memmgr** | 7168 | 2816 | 11520 | 5888 | 5120 |

Table 4.3: Cost Metric in integral multiple of 256 bytes

### 4.3.2 Results based on average execution time

Table 4.4 tabulates the average execution time of the malloc and free function on a managed memory pool of 10KB which is taken over multiple runs of the functions on various memory traces.

| | heapLib | memmgr | st_memmgr with 1 Bank | st_memmgr with 2 Bank | st_memmgr with 3 Bank |
|---|---|---|---|---|---|
| **malloc** | 1626 | 214 | 1222 | 817 | 773 |
| **free** | 2485 | 138 | 27 | 27 | 27 |

Table 4.4: Average Execution time in CPU cycles for 10K of managed space on taken over multiple runs

### 4.3.3 Results based on ROM footprint

Table 4.5 compares the ROM footprint in bytes of each of the memory managers, when compiled with the -O3 optimization flag, and all debugging and profiling codes disabled.

| heapLib | memmgr | st_memmgr |
|---|---|---|
| 2788 | 1680 | 2360 |

Table 4.5: ROM footprint in bytes (compiled with -O3 optimization flag using arm-none-eabi-gcc-4.8.1)

## 4.4 Discussion

From the results in section 4.2 it can be observed that introducing multiple memory banks actually has an adverse effect on external fragmentation. This invalidates the hypothesis formulated in 2.5, that localizing similar sized memory allocation using separate memory banks can help reducing the problem of external fragmentation.

The results indicate a flaw in the hypothesis, which is due to the assumption that if one memory request cannot be serviced in a memory bank then the next memory bank in priority should be considered. In very simple terms, dividing an entire contiguous block of memory into multiple memory banks implicitly involves a level of fragmentation. This idea is illustrated in figure 4.4, where a 7KB managed memory pool is divided into 3 memory banks of sizes 2 KB, 3KB and 2KB respectively. Bank 0 has 2 free fragments of 100 bytes and 250 bytes, Bank 1 has one free fragment of 300 bytes and Bank 2 has a free fragment of 100 bytes. Now since memory banks are used, the free fragments of 250 bytes from Bank 0 and the 300 bytes from Bank 1 cannot be considered as a single large 550 byte free fragments. So while calculating the external fragmentation the largest serviceable block will be 300 bytes, and the total free space will be 750 bytes. Hence the external fragmentation will be given as follows:

$$\boxed{1 - \frac{300}{750} = 0.6}$$

On the other hand if there were no memory banks, then the largest serviceable block would be 550 bytes, hence the external fragmentation in that case will be given as follows.

$$\boxed{1 - \frac{550}{750} = 0.266}$$

Despite the invalidation of the hypothesis, the comparative results involving the minimum memory metric tabulated in table 4.3 shows that the proposed memory allocator performs better than that of heapLib and memmgr. The results confirms that the proposed memory manager can service the

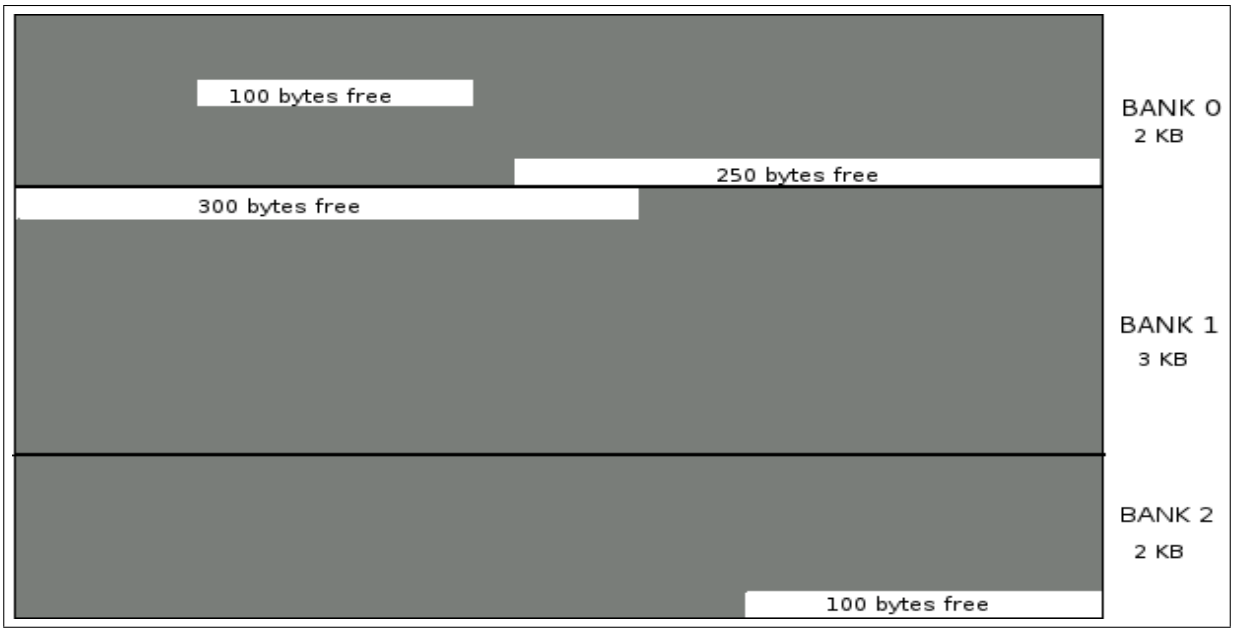


Figure 4.4: illustration of implicit external fragmentation due to memory banking

allocation requests of a program using a managed memory which is 256 — 1024 bytes lesser than that of heapLib and about 512 — 5376 bytes less than that of memmgr.

Results in average execution time as given in table 4.4 indicates that the allocation process is fastest for memmgr. The proposed memory manager's allocation process is slower than that of memmgr, but faster than that of heapLib. The execution time of free however is best in the proposed memory manager. The effect of introducing memory banks on execution time has also been studied and it has been observed that, there is a slight improvement in the execution time of the allocation process, as the search space is limited when memory banks are introduced.

Table 4.5 shows that in terms of ROM footprint memmgr has the lowest memory footprint with about 680 bytes less than that of the proposed memory allocator. It is important to make note of the point that the proposed memory allocator still has conditions and checks on deciding memory banks, which as the results suggests can be disabled, which will eventually lead to a smaller code footprint.

## 4.5 Inference

From the results and discussion it can be inferred that while designing memory allocators for constrained devices, one of the most important aspects that should be brought to the design consideration is the size of the memory descriptors used for maintaining allocation and de-allocation information. The proposed memory manager uses a 2 byte memory descriptor when compared to heapLib's 5 byte memory descriptor or memmgr's 8 byte memory descriptors. Consider a program that makes 160 allocation requests and the size of the managed memory pool is 5KB. The amount of overhead incurred by the memory descriptors in each case is as follows:

1. **st_memmgr:** 2 * 160 = 320 bytes leading to a 6.25% overhead.
2. **heapLib:** 5 * 160 = 800 bytes leading to a 15.6% overhead.
3. **memmgr:** 8 * 160 = 1280 bytes leading to a 25% overhead.

Another important inference that can be drawn from figure 4.3 is that for majority of real applications as well as random allocation traces the maximum external fragmentation lies between 0 — 0.2 with a single contiguous memory pool. Partitioning the memory into banks, has an adverse effect on external fragmentation and should be avoided, although it decreases execution time to certain extent.

# Chapter 5

# Conclusion

## 5.1 Conclusion

In this dissertation the problem of dynamic memory management in constrained embedded systems is revisited. The motivation for this work emerged as a requirement during a project where a viable replacement was required for the dynamic memory management in standard C library. Although various embedded dynamic memory management API are present, they are not compliant with the dynamic memory management of the standard C library. Using one of these non-compliant libraries would have led to refactoring a complex code base, which has a high chance of introducing bugs and also increase the development time. Other opensource embedded dynamic memory managers failed to service all requests within a limited memory.

As a first contribution, a detailed methodology is described for gathering allocation and de-allocation traces from real applications. Using this approach valuable insight about the allocation and de-allocation requests made by an application can be gathered. This methodology can also be used for generating reliable datasets for testing purposes.

The second contribution of this work presents a hypothesis based on the statistical analysis of various real application which is later invalidated through experimental results. But based on the observations a more concrete design strategy is established which can improve the efficiency of a dynamic memory manager in terms of optimum use of the memory.

As a final contribution three complete memory manager design are presented. Majority of the results are based on the second design as it is an improvement over the first one. The third design is an extension of the second design but deviates slightly in terms of compliance with the dynamic memory API of the standard C library. The third design includes memory compaction which will result in 0 external fragmentation.

## 5.2 Future Work

The research that has been described in this dissertation can be further explored, by finding better datastructures that may reduce the search complexity of $O(n)$ incurred during the allocation operation. Since most embedded systems perform realtime operation, a quick and bounded response is

often desirable. Moreover reducing the search complexity will also lead to lower power consumption, another desireable aspect for resource constrained embedded systems used in sensor network applications.

Finally it is worth exploring the performance of the third memory management scheme proposed in this dissertation and do a comparative analysis with the managed memory allocator in Contiki OS which also implements memory compaction, with an API very different from the standard C library dynamic memory management API. Although the third memory management scheme deviates from the standard C dynamic memory management API, it can still be closely related.

# References


[1] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[2] Don Sturek. Zigbee ip stack overview. *ZigBee Alliance*, 2009.

[3] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. Tlsf: A new dynamic memory allocator for real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 79–88. IEEE, 2004.

[4] James L Peterson and Theodore A Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[5] Doug Lea. A Memory Allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`, 2000.

[6] Manjiri Pathak. An approach to memory management in wireless sensor networks. August 08 2013.

[7] David Atienza, Stylianos Mamagkakis, Francky Catthoor, Jose M Mendias, and Dimitris Soudris. Dynamic memory management design methodology for reduced memory footprint in multimedia and wireless network applications. In *Proceedings of the conference on Design, automation and test in Europe-Volume 1*, page 10532. IEEE Computer Society, 2004.

[8] Andrew Borg, Andy Wellings, Christopher Gill, and Ron K Cytron. Real-time memory management: Life and times. In *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 11–pp. IEEE, 2006.

[9] M Ramakrishna, Jisung Kim, Woohyong Lee, and Youngki Chung. Smart dynamic memory allocator for embedded systems. In *Computer and Information Sciences, 2008. ISCIS'08. 23rd International Symposium on*, pages 1–6. IEEE, 2008.

[10] Mohamed Shalan and Vincent J Mooney. A dynamic memory management unit for embedded real-time system-on-a-chip. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, volume 17, pages 180–186, 2000.

[11] Mohamed Shalan and Vincent J Mooney III. Hardware support for real-time embedded multiprocessor system-on-a-chip memory management. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 79–84. ACM, 2002.

[12] Hong Min, Sangho Yi, Yookun Cho, and Jiman Hong. An efficient dynamic memory allocator for sensor operating systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1159–1164. ACM, 2007.

[13] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications*, 10(4):563–579, 2005.

[14] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176. ACM, 2005.

[15] Cameron Rich. axTLS embedded SSL. `http://axtls.sourceforge.net/index.htm`, 2011.

[16] Ariya Hidayat. FastLZ. `http://fastlz.org/`, 2007.

[17] Shailesh Ghimiray. huffman-textfile-compression. `https://code.google.com/p/huffman-textfile-compression/`, 2011.

[18] Bruno Essmann. SimpleXML 1.0. `http://simplexml.sourceforge.net/index.html`, 2002.

[19] Ross McIlroy, Peter Dickman, and Joe Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 31–40, New York, NY, USA, 2008. ACM.

[20] George Albert Buzsaki and Douglas James McMahon. Dynamic memory allocation in a computer using a bit map index, July 21 1998. US Patent 5,784,699.

[21] Paolo Pariani. heapLib 1.1. `http://sourceforge.net/projects/heaplib/`, 2012.

[22] Eli Bendersky. memmgr. `http://eli.thegreenplace.net/2008/10/17/memmgr-a-fixed-pool-memory-allocator/`, 2008.

[23] Christian Del Rosso. The method, the tools and rationales for assessing dynamic memory efficiency in embedded real-time systems in practice. In *Software Engineering Advances, International Conference on*, pages 56–56. IEEE, 2006.