Tampere University

Eetu Auvinen

EMBEDDED DYNAMIC MEMORY ALLOCATOR OPTIMISATION

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Professor Karri Palovuori
Professor Jukka Vanhala
May 2022

ABSTRACT

Eetu Auvinen: Embedded Dynamic Memory Allocator Optimisation
Master of Science Thesis
Tampere University
Embedded systems
May 2022

The objective of this thesis was to improve upon the dynamic memory allocator used in U-Blox GNSS receivers. After initial analysis, the main weakness of the currently used Buddy allocator was determined to be high fragmentation, so lowering this was the focus.

To understand the problem better and find possible alternatives to Buddy, a literature survey was conducted. The survey examined the fundamental building blocks of dynamic memory allocation and how they could be combined for desired results. Using knowledge gained from the survey, a number of allocators were chosen for further examination. Among them, the most promising were two real-time allocators TLSF and a Half Fit variant called O1Heap due to their focus on integrity and more constrained heaps.

After preliminary testing, two more allocators were considered as a response to the observed shortcomings of previously mentioned allocators. First was Half Fit CSTM, which makes slight modifications to the Half Fit algorithm trading constant execution time for lower fragmentation. The second allocator is Half Tree, which is a new allocator created in the scope of this research. It aims to address the weaknesses of the other considered allocators and achieve good results in all required aspects.

A testing framework was designed and implemented to assist in evaluating each allocator's suitability as an alternative for the Buddy allocator. Performance tests measured each allocator's average and worst execution times while fragmentation tests measured the required heap size and other metrics for fragmentation. The tests were run on a real device to provide more accurate results. Testing focused on real allocation sequences (memory traces) gathered from a production device, but synthetic traces were also included. All allocators went through the same tests, and their results were compared to those of the Buddy allocator.

Out of the 4 allocators considered, TLSF and Half Tree were determined to be the most suitable. Half Tree was deemed superior to TLSF because of much better performance while still having low fragmentation. Finally, Half Tree reduces fragmentation to approximately 15% to that of Buddy while only being 0–15% slower in real trace test scenarios. It requires 25% less heap for the same memory traces.

Overall, the thesis objectives have been met. Half Tree allocator provides a significant improvement in memory efficiency over Buddy while maintaining comparable performance.

Keywords: dynamic memory, TLSF, Half Fit, memory allocation

TIIVISTELMÄ

Eetu Auvinen: Sulautetun dynaamisen muistiallokaattorin optimionti
Diplomityö
Tampereen yliopisto
Sulautetut järjestelmät
Toukokuu 2022

Työn tavoitteena oli parantaa U-Bloxin GNSS laitteissa käytettyä muistiallokaattoria. Alustavan analyysin perusteella, nykyisen Buddy-allokaattorin heikkoutena oli korkea fragmentaatio, joten tutkimuksen pääpainona oli tämän alentaminen.

Ongelman ja sen taustojen ymmärtämiseksi, sekä parempien allokaattorien löytämiseksi, tehtiin työn alussa kirjallisuusselvitys. Selvityksen avulla määriteltiin dynaamisen muistinhallinan perusteet ja kuinka niiden avulla voidaan rakentaa halutunlaisia allokaattoreita. Selvityksen osana tarkasteltiin useita erilaisia allokaattoreita ja päädyttiin lopulta valitsemaan 2 allokaattoria tarkempaa tutkimusta varten. Nämä olivat reaaliaika käyttöön suunnitellut ja laajalti tunnetut TLSF ja Half Fit variantti O1Heap.

Alustavien kokeiden perusteella, toteutettiin kaksi uutta allokaattoria vastaamaan kokeissa havaittuihin TLSF:n ja O1Heapin heikkouksiin. Ensimmäisenä Half Fit CSTM, joka tekee pieniä muutoksia Half Fittiin vaihtaen vakioaikaisen suorituksen matalampaan fragmentaatioon. Half Tree puolestaan on uusi allokaattori suunniteltu tätä tutkimusta varten. Sen tavoitteena oli vastata muiden allokaattereiden heikkouksiin ja suoriutua hyvin kaikilla osa-alueilla.

Allokaattoreiden soveltuvuutta Buddyn korvaajaksi arvioitiin kokeellisesti. Kokeiden avulla mitattiin allokaatoreidenn keskimääräistä suorituaikaa, huonointa suoritusaikaa sekä fragmentaatiota ja muistintarvetta. Koejärjestelyt suunniteltiin ja toteutettiin kohdelaitteella tulosten todenmukaisuutta ajatellen. Kokeet keskittyivät laitteelta kerättyihin oikeisiin muistijälkiin, mutta myös synteettisiä muistijälkiä hyödynnettiin. Kaikille 4 allokaattorille suoritettiin samat kokeet ja niiden tuloksia verratiin Buddy -allokaattoriin ja toisiinsa.

Valituista allokaattoreista TLSF ja Half Tree osoittautuivat parhaiten soveltuviksi. TLSF on kuitenkin erittäin hidas verrattuna Buddy -allokaattoriin. Half Tree puolestaan on selvästi parempi, tarjoten edelleen matalan fragmentation, mutta ilman merkittävää kompromissia suorituskyvyssä. Half Treen fragmentaatio on noin 15% Buddy-allokaattorin fragmentaatiosta ja suorituskyky heikkenee vain 0–15% muistijäljestä riippuen. Vaaditun muistin määrässä alempi fragmentaatio tarkoittaa noin 25% parannusta.

Työ täytti sille asetetut tavoitteet. Half Tree allokaattori tarjoaa merkittävän parannuksen muistitehokkuudessa Buddy-allokaattoriin verrattuna, tekemättä suuria uhrauksia suorituskyvyssä.

Avainsanat: muistinhallinta, dynaaminen muisti, TLSF, Half Fit, muistiallokaattori

PREFACE

This thesis was written for U-Blox AG in the spring of 2022. I'd like to thank U-Blox for giving me the opportunity. I would also like to thank the OS team and the U-Blox working environment at large for their support during my work. A special thanks to Alexander Lukichev for continued guidance and valuable feedback helping me improve my work, and also for providing such an interesting topic.

I would also like to thank my thesis supervisors Karri Palovuori and Jukka Vanhala for their support and feedback. I'd also like to thank them and the university in general for the valuable skills I've learned throughout the years.

Lastly, I'd like to thank my friends and family for their support. Finally, I am eternally grateful to Vera, whose guidance, support and love helped me push through the hardest challenges.

Tampere, 13th of May 2022

Eetu Auvinen

TABLE OF CONTENTS

ABBREVIATIONS AND SYMBOLS

DV	Designated Victim, a memory block kept outside the main data structure to promote its reuse
GNSS	Global Navigation Satellite System
FIFO	First-in-first-out, a queue where the oldest item is removed first
ffs	Find first set, finds the most significant bit set to 1
LIFO	Last-in-first-out, a stack where the newest item is removed first
MMU	Memory management unit, a device which transforms virtual addresses to physical addresses
SBBM	Smallest-Biggest Block Metric, a fragmentation measurement method
TLSF	Two-Level Segregated Fit, a general-purpose allocator for use in real-time systems

1. INTRODUCTION

This chapter gives a brief introduction to the subject of dynamic memory allocation and its role in embedded systems. Additionally, it defines the objectives of this thesis and provides an overview of the contents.

1.1 Dynamic memory allocation in embedded systems

Dynamic memory is used to hold objects with unknown size at compile time. Such situations can arise for example from varying amounts of tracked satellites or different lengths of messages placed in buffers. It allows the program to allocate only the necessary amount of memory at any given time from the dynamic memory pool, which is called *heap*. The basic usage of dynamic memory consists of allocating the desired amount of memory with *alloc()* call and once the memory is no longer required, deallocating it with corresponding *free*() function call. This can be further expanded by more specialised memory services like *realloc()* to adjust the size of the allocated block. [1]

The research on dynamic memory usage in computer systems goes back as far as the 1960s. This earlier research up to 1995 was compiled by P.R Wilson and colleagues in their survey which acts as the fundamental basis for most dynamic memory research to this day. [2] While it focuses on the concepts and problems facing dynamic memory in general-purpose computers, those same problems and concepts are relevant in more restricted embedded systems as well.

Traditionally the use of dynamic memory in embedded systems has been avoided due to its perceived inability to match the timing requirements and resource constraints of embedded applications [3][4]. This is partly due to the allocators from before not being as concerned about real-time constraints nor necessarily about being as memory efficient as possible [2][3]. On the other hand, dynamic memory is fundamentally slower and less efficient than static memory [1]. However, as the complexity and unpredictability of embedded applications have increased, so too has the need for efficient dynamic memory management in embedded systems over the last couple of decades [5]. This emerging demand for more efficient and reliable dynamic memory management has resulted in more research on specialized allocators to work in this context with two examples being Half Fit and Two-Level Segregated Fit (TLSF) [3][4].

For dynamic memory allocator to be viable in the real-time embedded context it generally needs to fulfil the following criteria. First, it needs to have a low bounded worst-case execution time. Second, it must be fast enough on average. Finally, it has to use the available memory efficiently, often meaning low fragmentation and implementation overhead. [4] The priority between these criteria will change from application to application. As such, no perfect allocator for every situation exists and the choice of allocator should be done on a case-by-case basis.

Due to the nature of positioning software, the use of dynamic memory is essential. The allocator currently used in U-Blox GNSS receivers is fast on average and has a low bounded worst-case execution. However, it wastes a large amount of memory, averaging around 50% of the requested size being lost to fragmentation.

1.2 Objectives of the thesis

This thesis was written at U-Blox Espoo Oy. U-Blox is a Swiss company specialising in GNSS devices. The main objective of this thesis was to find a way to reduce the amount of wasted memory while maintaining similar or better execution times.

The target devices are U-Blox GNSS receivers, with all the work being done and tested on an M9 standard precision GNSS module. The chip uses an ARM processor without a memory management unit (MMU), which makes dealing with heap fragmentation more critical. A side objective of this thesis was to provide a testing framework for dynamic memory allocation on the target device.

1.3 Thesis structure

This thesis can be broken down into 3 components. These are roughly background theory, test implementation and finally results and analysis. Should the reader be familiar with theory of dynamic memory allocation theory, Chapter 2 and sections of Chapter 3 discussing allocators the reader is already experienced with may be skipped.

Chapter 2 presents the fundamental dynamic memory allocation theory necessary for this thesis. Notably, it introduces the parameters to evaluate an allocator's quality as well as the building blocks used to create memory allocators.

Next, Chapter 3 introduces the allocators considered in this work. It goes over the mechanisms and data structures of 3 pre-established allocators and introduces a new allocator, Half Tree, created in the scope of this research.

Building upon the theory provided in earlier chapters, Chapter 4 discusses the details related to testing. It gives an overview of the test environment and what is being measured as well as how these measurements are conducted. Additionally, the finer details of allocator implementations during testing are specified here.

The results of the testing are presented in Chapter 5. It also provides an analysis of the results and their significance in achieving the objective of this thesis. Lastly, Chapter 5 also provides an overall look into the suitability of considered allocators to replace the currently used one.

Chapter 6 discusses the conclusions and future work for this thesis. After conclusions, a list of references is provided. Lastly, 2 appendixes are provided for pseudocode of Half Tree allocator and additional test results not discussed in Chapter 5.

2. DYNAMIC MEMORY ALLOCATORS

Dynamic memory allocator is a program responsible for managing the heap and servicing allocation and deallocation requests. It is worth noting that usually the allocator can not manage allocated memory blocks since they are being handled by those who requested them. The allocator also can not influence the sequence of allocations and deallocations and has no other option but to service them as they come. The choices an allocation algorithm makes are therefore limited to how it handles the available free memory. As discussed in the previous chapter the goal of an allocator is to serve requests quickly while wasting as little memory as possible. [2] This chapter will define these 2 aspects and discuss design techniques and allocation mechanisms used to help dynamic memory allocators reach that goal.

2.1 Parameters of dynamic memory allocators

The 2 main aspects to determine the quality of a dynamic memory allocator are performance and memory efficiency. In this subchapter, we introduce these fundamental characteristics and how they relate to embedded applications.

2.1.1 Performance

Performance of a dynamic memory allocator in the context of this work refers to the time it takes to allocate or deallocate a memory block. Typical parameters for this are instructions per operation or processor cycles per operation. Of these, the latter is preferred when comparing different allocators [6]. In comparison to static allocation, dynamic memory will always carry a penalty in performance. This is exemplified in embedded systems with real-time requirements, where the most important aspects of performance are more closely related to the memory meeting timing requirements instead of being as fast as it can be. For dynamic memory allocation to be viable in a real-time environment it must both have a bounded worst-case execution time and be fast enough [4].

A way to estimate allocation algorithms' worst-case performance is to analyse its time complexity. The O-notation describes how an algorithm's worst-case execution time increases as the heap grows. For example, if an allocator has a time complexity of O(n), its worst-case execution time scales linearly as the heap size increases. [8] Generally, O(log(n)) is considered good enough in most applications, but the ideal real-time allocator would have a constant O(1) response time [7].

While being a valuable metric to estimate how the execution time might increase with other parameters, the O-notation omits the constant multiplier, for example execution scaling 4n or 10n are both linear and therefore O(n). This means that an algorithm with constant time complexity O(1) could take too long to execute even if it is bounded. [4] It is also worth remembering that in resource constrained devices the amount of memory an allocator can manage is often bound to a relatively low limit, effectively acting as a boundary for the execution time.

For soft real-time systems, like the target device, the worst-case execution time is not as critical. Compared to hard real-time systems, soft real-time systems can handle a small number of requests missing their deadlines. Instead, the focus can be shifted more towards a fast average execution time. This does not mean the worst case can be completely ignored and should be kept within reasonable limits, but in practice most allocators will only be at their worst in very specific conditions. Additionally, in Puaut's research [8] it was shown that analytical worst-case execution time is often far worse than the experienced one, leading to the conclusion that a worst-case measured from the target application can be sufficient for soft real-time applications. [8]

2.1.2 Memory efficiency and fragmentation

Being able to use limited memory more efficiently is one of the main motivators in using dynamic memory. However, it does have its own issues regarding efficient memory use. The main source of wasted memory is fragmentation. Fragmentation relates to inefficient use of memory in the heap. [5] It was defined by Wilson and others as "inability to reuse memory that is free", which has stuck as the standard high-level definition [2].

Embedded systems typically do not have a memory management unit (MMU), making fragmented memory blocks harder to utilise. MMU could enable the use of fragmented memory blocks as if they were continuous, alleviating the issues arising from a fragmented heap. Additionally, assuming the heap size cannot grow during run time means the allocator has no way of servicing an allocation request if the heap is too fragmented and the allocation fails instead. Repeated failures to serve allocation requests will lead to degradation in the quality of the service provided by the system over time. [9]

Fragmentation can be divided into 2 distinct types, internal and external fragmentation. Internal fragmentation occurs when the allocator gives the user a larger block of memory than what was requested. This additional memory cannot be allocated to serve another request and is therefore wasted until the block is deallocated. External fragmentation on the other hand occurs when the allocator does not have a large enough continuous memory block to service a request even though there are enough scattered smaller blocks free to add up to the requested size. [5] This is caused by differences in deallocation times of blocks next to each other and changes in block sizes allocated [2]. Figure 1 shows an example of both internal and external fragmentation in a simple memory structure.

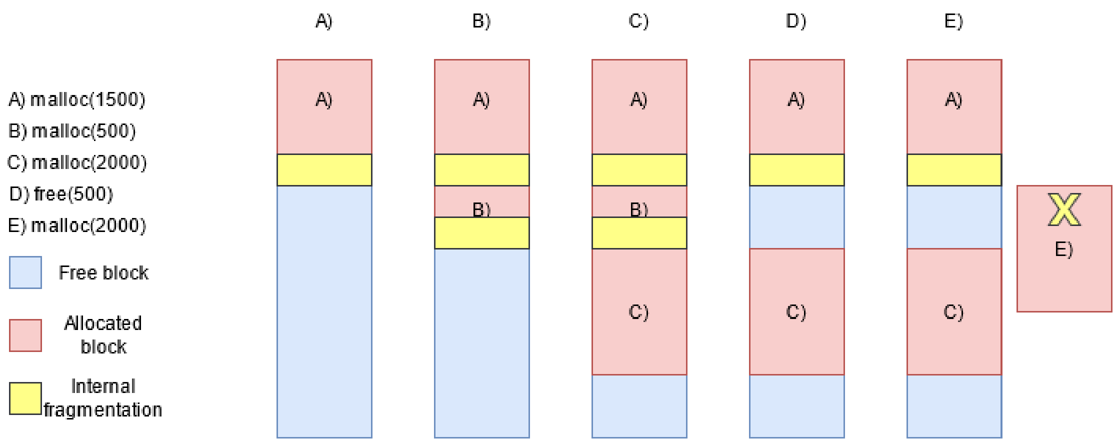

Figure 1 *Example of memory fragmentation in a simple memory.*

The memory in Figure 1 is organised in such a way that it always gives the first available memory block that is large enough, only splits larger memory blocks in multiples of 1000 bytes and has a total size of 6000 bytes. This leads to some memory being wasted due to internal fragmentation for allocations A and B, which is illustrated by a yellow colour. Additionally, allocation E fails even though there is in total 2000 bytes free, but half of that is before C and the other half after it. This is an example of external fragmentation.

The second major source of wasted memory is the implementation overhead of the allocator itself [5][6]. Depending on the design choices this can be in managing the free lists or header and footer fields of allocated blocks. Johnstone and Wilson [10] even argue that this can be considered the most important part of wasted memory. Allocators with low fragmentation but high implementation overhead, possibly due to poor implementation, exist. While having some implementation overhead is unavoidable, it can be optimised. Therefore, another angle to tackle the memory efficiency problem in addition to decreasing fragmentation is making more memory efficient implementations. [10]

Ensuring minimal amount of memory is wasted is essential for embedded systems. Due to the fragmentation problem having more pronounced benefits from optimisation, over performance simply being good enough, it will take a more prominent role in the analysis of design choices and eventually test results. Additionally, the focus of this work will be on the total memory footprint of the allocator which includes both forms of fragmentation and the implementation overhead [5]. While understanding the different sources of memory waste helps us reduce their effects, they are best examined together since often reductions in one aspect may be done at the cost of another [2].

2.2 Policies and design techniques

Wilson and colleagues separate allocator policy from its implementation mechanism. Allocation policy refers to choices regarding the placement and allocation of free blocks and is independent of the mechanism used to implement it. In addition, Wilson and others also define allocation strategy, which can be thought of as picking the suitable policies for the program's allocation pattern. [2] The main policy choices relate to block placement, splitting and coalescing.

There are many policies for choosing which free block to allocate. Knuth introduces 2 frequently used policies for making this decision, first fit and best fit. With first fit policy the allocator will always allocate the first block of memory it finds that is big enough to satisfy the request. In contrast, the best fit policy will go through the available blocks and use the smallest possible block that is big enough to fulfil the request. [11] The principle of best fit is good and often leads to better memory efficiency than the simpler first fit. However, depending on the implementation of each, best fit might be noticeably slower without offering substantial enough benefits to fragmentation. [2][10][11] Another interesting placement policy is good fit, which works similarly to best fit but instead of finding the most suitable block, it finds the first good enough block based on some metric. This policy has been used for example in TLSF and an older version of Doug Lea's allocator, due to its combination of performance and spatial efficiency. [4][12]

Another policy choice regarding the placement of allocations is the ordering of free blocks. There are 3 main ways of handling this which are address ordered, FIFO (first-in-first-out) and LIFO (last-in-first-out). Address ordering effectively does not consider the reuse of blocks and instead goes strictly by their location in memory. This will typically still favour the start of the heap and reuse blocks freed in that region frequently. LIFO policy will add freed blocks to the front of the free list and reuse them before allocating new blocks. FIFO on the other hand adds freed blocks to the back of the free list and will use new blocks before reusing freed ones. [10] Historically, address ordered has been considered as the best policy in terms of fragmentation, but slower than FIFO or LIFO. [2]

The requested allocation sizes can vary wildly depending on the application and the allocator will not always have the exact size block in its free list. In such situations, the allocator can choose to *split* a larger chunk of memory to create a block of the desired size. [2] The split does not have to produce a block of the exact requested size but can instead use some other metric to do the splitting, for example the binary buddy system where each split always halves the larger block [13]. Splitting decreases internal fragmentation but can increase external fragmentation if the leftover free memory is not possible to utilise. To avoid cluttering the memory with unusable small chunks, the allocator can have a *splitting threshold* policy in place. With a splitting threshold, the allocator will only split the block if the leftover will be large enough for it to use. Splitting a block will also always be slower than not splitting it, so a splitting threshold can be used to avoid the loss in performance if the hit in memory efficiency is deemed worth taking. [2]

The inverse operation to splitting is called *coalescing*. Coalescing means merging neighbouring free blocks back into one larger free block. This stops the heap from becoming filled with small blocks unable to serve a large allocation request. However, as with splitting, coalescing also comes with a performance cost and is always slower than not coalescing. This can be balanced with a coalescing policy called *deferred coalescing.* When using deferred coalescing the blocks are not coalesced as soon as they are freed but instead the coalescing is triggered by some pre-determined event. The idea behind this is that the program is likely to reuse the same size blocks often and it is therefore an unnecessary hit in performance to constantly split and coalesce blocks only to repeat the cycle on every allocation. The coalescing can then be done when the program changes its allocation pattern to better service the new sequence. [2] Deferred coalescing is commonly used to improve performance in general-use computer environments, for example in Doug Lea's allocator coalescing is only done when memory runs out [12]. However, it has been shown to increase fragmentation and introduce unpredictability since the request that eventually triggers the coalescing process will take significantly longer than usual to complete [2][9].

By making the right policy choices, the best possible allocator for a given system can be designed. Each choice comes with its tradeoffs which must be carefully considered with the target system in mind. To maximize performance, Unnecessary work done by the allocator is left at a minimum, often at the cost of wasted memory. On the other hand, to make the allocator as memory efficient as possible it should use a best fit or good fit policy with frequent reuse of freed blocks and also do precise splitting and immediately coalesce blocks when possible [6][10].

2.3 Low-level mechanisms

To turn a set of allocation policies into a functional dynamic memory allocator they need to be realised with an allocation mechanism. Different mechanisms in turn use a lot of similar low-level mechanisms in their foundation. This section will introduce and give a brief description of the most used ones. For the sake of brevity, this subchapter will refer to low-level mechanisms as simply mechanisms.

First such mechanism is using *header* fields on allocated blocks. These headers are hidden from the user by the allocator adding the required space for a header field to the requested amount then using the first few bytes for its own needs before handing over the requested amount of memory starting after the header. The header will typically contain the size of the block, information about the neighbouring block or status flags. Header information can be used to make some operations faster, but it adds a potentially significant implementation overhead depending on the size of it and the allocated blocks. [2]

An extension of using a header field called *boundary tags* was conceived by Knuth in 1973 [11]. For this mechanism, an additional *footer* field is added to the end of each memory block. Both the header and the footer contain information about the size of the block and whether it is free or not. The purpose of duplicating information in this manner is to make coalescing easier since a freed block can easily check the footer of the previous block and the header of the next block to determine if coalescing can be done. This adds an even larger implementation overhead than just using headers but makes managing blocks easier. [11] However, this can be optimised since the allocator is not interested in the size of an allocated block because it can not be coalesced. Now, by borrowing a status bit from the next memory block’s header to signal whether the block is free or not, we can only use the full footer field in free blocks and the area reserved for it can be used to fulfil allocation requests [2]

A common way for allocators to keep track of free blocks is using linked lists. This mechanism can be made very memory efficient by using the memory in the free blocks themselves to store the required pointers. This memory is effectively free and adds very little overhead for free block tracking. [11] Since this memory is almost free it is common to use doubly linked lists to make it easy to remove blocks from it as they are coalesced. Using the memory in the free blocks imposes a minimum size to blocks in the system to ensure the needed information can always be stored in the free blocks. [2] Figure 2 shows how a linked list might be implemented.

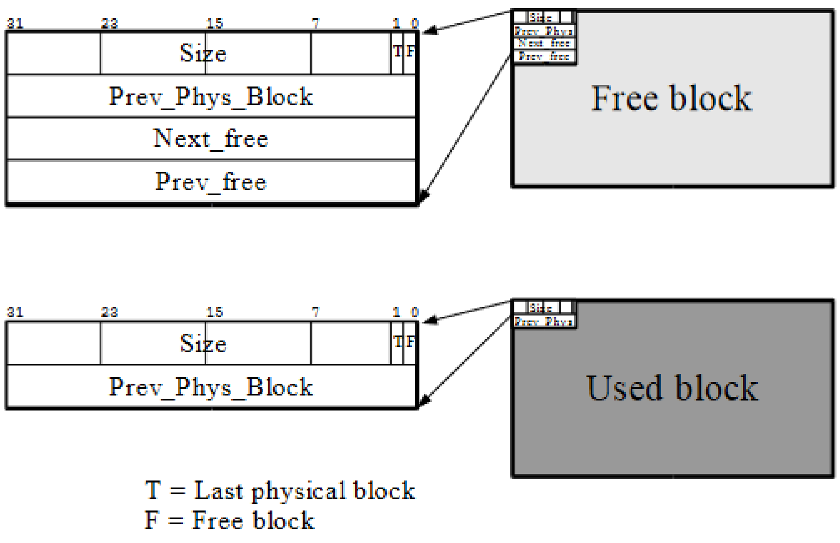

Figure 2 *Block headers for a linked list in TLSF [9].*

As can be seen in Figure 2, the memory needed to upkeep the linked list is not needed once the block is allocated and the space is instead used to satisfy the allocation size. In addition to being an example about linked list implementation, figure 2 also shows the contents of a typical header field discussed earlier. [9]

Another mechanism used to track free blocks and make allocations faster is to use lookup tables. In practice, this means lumping memory blocks of similar size together in their separate lists. [2] Using a lookup table is especially helpful when implementing either a best fit or a good fit policy, since only blocks of roughly the right size are considered [12]. However, if the range of sizes in any of the lists is too large, it can lead to significant internal fragmentation [2].

The last mechanism regarding free block management is giving special treatment to the end of the heap, sometimes referred to as *wilderness preservation* [12]. In less restricted systems, where the heap can grow from one end, giving special attention to preserving a large block near that end can help reduce unnecessary extensions of the region. [2] There are many ways to achieve this but for example Doug Lea's allocator considers the last block much larger than its actual physical size since it can be made so by requesting more memory. Combined with a best fit approach, the last block of the heap will only be used if no other block can satisfy the request. [12] Our target device and most other embedded systems have a fixed heap size making this heuristic less useful. It can still be helpful to strive to preserve one end of the heap for larger allocations, especially since instead of being able to request more memory if a large enough contiguous block is not found, the allocation will fail.

The last low-level mechanism considers giving special treatment to small allocation requests. There is a lot of motivation to do this since programs typically allocate many more small objects than large ones and often the same allocation mechanism is not the best for both types. This often means using a fast but less memory efficient mechanism for small allocations and some other, less wasteful, mechanism for large allocations. [2] Depending on heap size this may not always be beneficial since different allocation mechanisms may lead to more overhead. However, giving special treatment to specific allocation sizes can help reduce for example header overhead.

2.4 Mechanisms

This section will introduce the most important basic allocation mechanisms and how they can implement different policies as well as combine various lower-level mechanisms in the implementations. The outline will largely follow the one used in the survey by Wilson et al. [2] but focus more on the advantages and weaknesses of each one in a more restrictive embedded systems context.

2.4.1 Sequential fits

Sequential Fits are perhaps the simplest allocation mechanism. They are based on having a single list of all blocks in the memory and searching through it to find a block to allocate. This list can be kept in either LIFO, FIFO or address order. Commonly the list is doubly linked and uses boundary tags to help in coalescing. The two most common sequential fit allocators are First Fit and Best Fit. [2][11] Sequential fit mechanisms are not limited to these two allocators, but other variants are not particularly relevant other than for curiosity. Sequential fits generally have poor real-time performance due to unpredictability and both First Fit and Best Fit having a time complexity of O(n) with increasing heap size [7].

Traditionally the most used sequential fit variant is Best Fit. In this variant the allocator will go through the free list and allocate the smallest block that is big enough to fill the request. As such, the ordering policy of the free list matters little for Best Fit. This also means that the average execution time grows rapidly as the number of free blocks increases making performance rather poor for larger memories. This is further made worse by Best Fit tending to increase the number of very small blocks and extending the length of the free list it must go through each time. [2][11] As an upside, Best Fit has been shown to lead to low fragmentation with both real and synthetic memory traces [7][10].

Out of the simplest class of allocators, First Fit is arguably the simplest. As opposed to finding the best block to satisfy the request, First Fit will instead allocate the first one it finds. For this reason, it is often used as a baseline allocator in many tests comparing various allocators [7]. Even though it is very simple in its function and has very bad worst-case execution time and fragmentation, First Fit can be made to be almost as good as Best Fit. Both Knuth and Wilson et al. state that address ordered First Fit can result in similar fragmentation as Best Fit [2][11]. This may be due to it adhering to wilderness preservation by naturally filling up one end of the heap first and tending to roughly order the heap by block size with smaller blocks near the beginning. In tests FIFO ordered first fit has been shown to be almost as good as address ordered. On the same tests, LIFO ordered first fit has been suspect to large fragmentation when faced with unfavourable allocation sequences. [2][10] The average execution time of First Fit is usually much better than Best Fit's, but it has a similar worst-case and can therefore be highly unpredictable [7].

While sequential fit allocators tend to have too poor performance or fragmentation characteristics to be considered viable in an embedded environment, they are still worthwhile to examine for their simplicity. A lot of the more complicated allocators use the same ideas found in sequential fit allocators in their foundation, or even include a similar list structure as a part of the mechanism like in segregated fits.

2.4.2 Segregated fits

Another simple allocation mechanism is the segregated fit, or more generally segregated free list. It uses a lookup table with free lists segregated by size and typically performs a sequential fit-style search on the appropriate list. This is commonly done with best fit or first fit policy. Due to the nature of having segregated free lists based on size the first fit policy becomes especially interesting, since it only considers approximately right sized blocks, it often naturally becomes a type of good fit policy. The ordering of blocks within the free lists can be LIFO, FIFO or especially with best fit policy, ordered by size. [2]

Segregated fits are usually considered a performance optimisation over sequential fits since it is much faster to find a block from the free list with large enough blocks than by going through a single list. If that list is empty, the allocator checks the next one up until it finds a non-empty list. It then takes a block from this list and splits it to desired size and places the leftover chunk in its free list. Conversely, when a block is freed, it may be coalesced with its neighbours taking them out of their respective free lists. Depending on the size ranges of different free lists the allocation is usually a logarithmic operation or O(log(n)) with deallocation being faster and typically constant time O(1). [2] Segregated fit performance can be further optimised to be constant time in both allocation and deallocation, which has made it a popular inclusion in real-time allocators [3][9].

Segregated fits can be divided based on the sizes in their free lists. Firstly, the free lists may be of exact sizes, leading to a potentially huge number of free lists [2][14]. This is often not viable as a solution for the whole memory but can be used effectively for small block sizes to eliminate internal fragmentation and possibly even some overhead as is done for example in Doug Lea’s alloc. Figure 3 shows the structure of segregated free lists in dlmalloc.

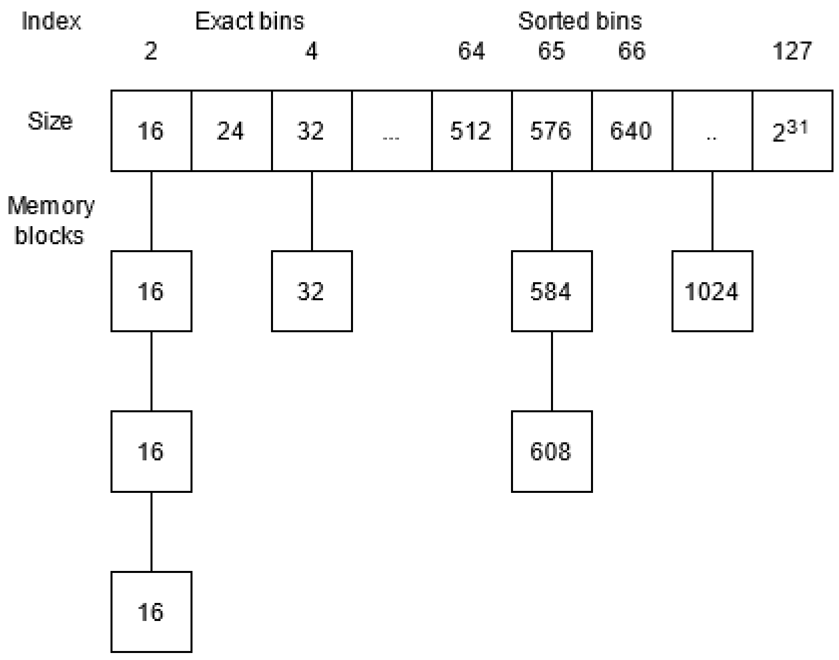

Figure 3 *Doug Lea’s allocator segregated free lists. Adapted from original in [12].*

In Figure 3 the first bins up to 512-byte blocks are done with exact sizes. From there another type of free list is introduced, one with a range of sizes. The first one of these lists contains all block sizes between 512–575 bytes and in dlmalloc’s case is sorted by size to make it closer to best fit. [12] The third and final type of free list is similar to exact size but uses rounded up sizes instead. This can for example mean that each block size is a power of two and requests are rounded up to the closest one. The main benefit of this variant is cutting down the amount of free lists and list searches needed, but at the cost of potentially severe internal fragmentation. [2]

Segregated fits are interesting for embedded and real-time systems because they can be made both memory efficient and sufficiently fast in performance. They may however have high implementation overhead if the list structures are not considered carefully.

2.4.3 Buddy system

Buddy systems are a special subcategory of segregated fits originally proposed by Knowlton [13]. What makes buddy systems different from other segregated fits is their limited splitting and coalescing. The memory starts as one large chunk, which is always split in two parts, which are called *buddies*, until the desired allocation size is reached. A buddy system does not allow general coalescing, but instead a block may only be coalesced with its buddy. These limitations mean all possible block sizes are known and can be held in their separate free lists. [2] Figure 4 illustrates splitting and coalescing in a buddy system.

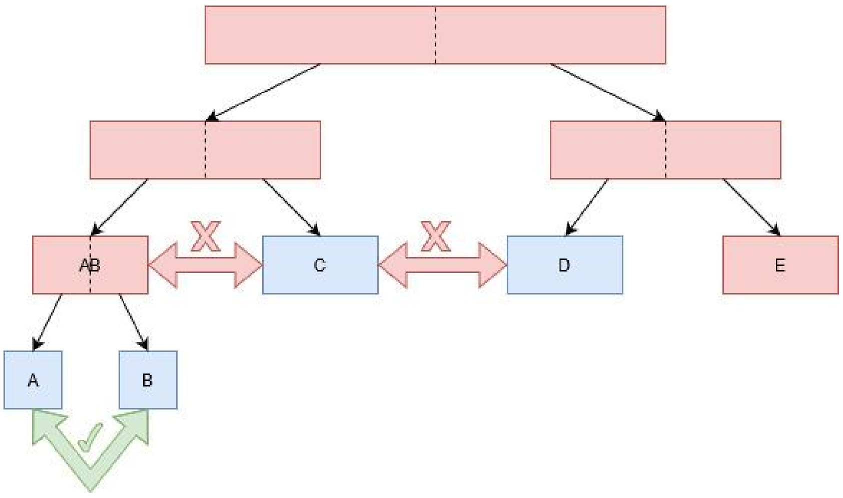

Figure 4 *Buddy system splitting and coalescing.*

In the above example, the larger block is always split in half to create the buddies. A and B can be coalesced since they are both free and are buddies. AB and C can not be coalesced since AB is still considered in use until its parts are coalesced. C and D is a case unique to buddy systems. They are neighbours in physical memory, and both are free, yet they are not buddies and therefore can not be coalesced.

The main benefit of using a buddy system is its performance. As discussed in the previous chapter, segregated fits have been of interest in real-time and embedded context, and historically buddy systems have been considered most suitable for these use cases [7]. Both splitting and coalescing operations are quite fast on average and with a limited number of levels the worst case is not catastrophically slow. This is in part due to each block knowing the size of its buddy and being able to find its header fast and partly due to the roughness of the fit resulting in internal fragmentation. The main weakness of buddy systems tends to be this higher than usual internal fragmentation. [2] As an upside, they suffer from relatively little external fragmentation, barring niche rare cases such as the one present in Figure 4 with blocks C and D being unable to be used for a larger request [11].

The simplest and most popular buddy system is the binary buddy, where each block size is a power of two and they are always split in half [2][7]. Binary buddy suffers greatly from internal fragmentation with larger block sizes where for example a request of 4100 bytes would have to be allocated 8192 bytes, wasting almost 50% of the allocation to internal fragmentation [2]. To combat binary buddy's internal fragmentation issues other variants of buddy systems have been explored.

One well-known variant is the Fibonacci buddy which splits blocks in Fibonacci series instead of half, for example splitting 13 to blocks of 5 and 8. [2] Fibonacci buddy should have less internal fragmentation than binary buddy, but it has been shown to be slightly slower [8]. Weighted buddy is more complex version that allows splitting a block in two different ways. These ways are power of two and 3 times power of two. This allows for more size classes than regular binary buddy. Lastly, there is the double buddy, which means using two buddy systems at the same time. This can be achieved for example by using a regular binary buddy to control one block and a binary buddy with some offset controlling another block. [2] All of these buddy system variations can have their advantages over binary buddy in terms of internal fragmentation but are not as universally applicable and require careful planning to work in their target environment. With synthetic trace testing binary buddy had the highest internal fragmentation, and with external fragmentation taken into account it still had higher fragmentation than Fibonacci and double buddy on average. [15]

Buddy systems are a viable mechanism for an embedded dynamic memory allocator. In fact, the allocator currently in use on the target device is a version of binary buddy that will be explored in more detail in the next chapter. They offer good enough performance for soft real-time use with an acceptable worst-case performance but suffer greatly from fragmentation.

2.4.4 Indexed Fit

Indexed fits are another variation of sequential fits. Where sequential fits used a linear linked list, indexed fits use a more complex indexing structure such as a binary tree. Typically, these specialized structures are used in an attempt to provide a better fit than sequential First Fit and a faster fit than sequential Best Fit. Indexed fits are the least defined of the mechanics introduced here but being aware of other indexing structures may prove valuable. [2]

One popular example of an indexed fit allocator is Stephenson's Fast Fits which uses a Cartesian tree sorted by size and address. From the tree, Stephenson proposes two different options for finding a block to allocate, which are Leftmost Fit and Better Fit. Leftmost fit functions similarly as first fit since it takes the first one by address on the left side. Better fit on the other hand goes through the tree choosing the best suitable block until both descendants are too small to satisfy the allocation. Both methods offer on average logarithmic worst-case execution but are truly O(n), should the tree get terribly unbalanced. [2][16] In Stephenson's testing, Fast Fits were not beneficial for small heaps managing small amounts of allocated blocks at once. But if the number of allocations was raised, Fast Fits were able to outperform sequential fit algorithms by a large margin while requiring similar implementation overhead. However, Leftmost Fit is naturally susceptible to becoming unbalanced since it favours one side of the tree, and Better Fit is not immune to it either. Should this happen the performance of Fast Fits would become even worse than sequential fits. [16]

Indexed fits are a broad category of different allocation algorithms that often work well on some allocation sequences and poorly on others. They are of interest to explore and try out but can have some notable pitfalls in worst-case performance, which means the indexing structure must be chosen carefully with worst-case in mind. Additionally, they can prove useful as a part of a multimechanic allocator, where a small tree size can be more easily ensured.

2.4.5 Bitmapped fit

The last basic mechanism discussed here is the bitmapped fit. In their simplest form they have a bitmap with one bit flag for each fixed size area in the heap that signifies whether it is free or allocated. The allocator will then perform bitmap searches to find a long enough streak of free areas and allocate that section to satisfy a request. In this basic format the bitmapped allocator performs slowly, but it can be made faster by using larger lookup tables to scan through the data. [2]

One advantage of using bitmaps is the low overhead per block of memory since only 1 bit of storage is required for each area in memory. For small block sizes, which are the majority of allocations in most programs, this can be a major reduction in wasted memory. They are however quite slow unless the searching algorithm can be heavily optimised and therefore not ideal for embedded systems. [2] But as with some of the previous mechanics, bitmaps can prove to be a useful tool in conjunction with other basic mechanics and for example, the real-time allocators Half-Fit and TLSF both use bitmaps to make indexing faster through the use of first free set (ffs) bit-scan operation [3][4].

3. GENERAL-PURPOSE ALLOCATORS

To find the dynamic memory allocator best suited for the application, a number of general-purpose allocators were considered. This chapter will go over the functionality of the most viable candidates. For the purposes of this thesis an allocator’s functionality includes how it implements allocation and deallocation of memory blocks as well as the data structures it uses to manage free blocks. Finally, a new allocator called Half Tree is introduced.

In addition to the allocators examined in this chapter, many others were looked into but dismissed before a more thorough examination. These include, but are not limited to, tcmalloc [17], mimalloc [18], umm_malloc [19], jemalloc [20], Fast Fits (AVL) [16], CAMA [21] and eheap [22] and tertiary buddy [23]. Dlmalloc was also determined to not be suitable as is, but it was used for inspiration in the Half Tree allocator.

3.1 Currently used allocator

At the time of writing the target device uses a form of binary buddy allocator to manage dynamic memory. The key difference from regular binary buddy, where each block size is a power of two, is that each block is instead a power of two multiplied by 12. This size scheme is used to align it with the free block header size. The smallest block size is set at $2^2 * 12 = 48$ bytes and there are a total of 9 size levels in use, meaning that the largest block size is 12288 bytes. The allocator manages blocks of different levels in their separate free lists to make finding appropriately sized free blocks faster. Entering blocks to the free lists is done with LIFO policy.

The allocation sequence in a binary buddy allocator is described with a flowchart in Figure 5.

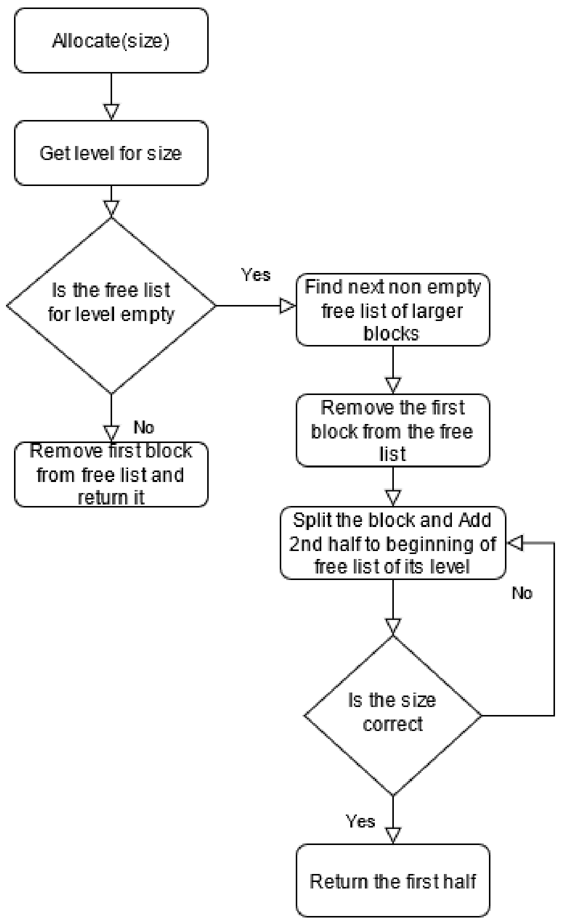
Allocate(size)
Get level for size
Is the free list for level empty
Yes
No
Find next non empty free list of larger blocks
Remove the first block from the free list
Split the block and Add 2nd half to beginning of free list of its level
No
Is the size correct
Yes
Return the first half
Remove first block from free list and return it

Figure 5 *Allocation sequence in binary buddy.*

The allocation in Figure 5 is quite simple and fast. First, the requested size is rounded up and transformed to the nearest level that is large enough for it. Second, If the free list holding blocks of this size is not empty, the first block in the list is removed and returned to the user. Alternatively, if the free list is empty, the next non-empty free list is found and the first block from there is removed. This block is then split in half until it matches the requested size. The first half is given to the user and the second halves are added to the front of the free lists of their respective sizes.

Since the number of levels is relatively low, the allocation is quite fast even in the worst case. The worst-case performance for allocation happens when there are only free blocks of the largest size and request needs the smallest block, resulting in maximum number of split operations. As an additional benefit to performance, if the next request is of the same size, the ‘buddy’ of the previous allocation will be ready to service the request in the free list.

The deallocation sequence is described with a flowchart in Figure 6.

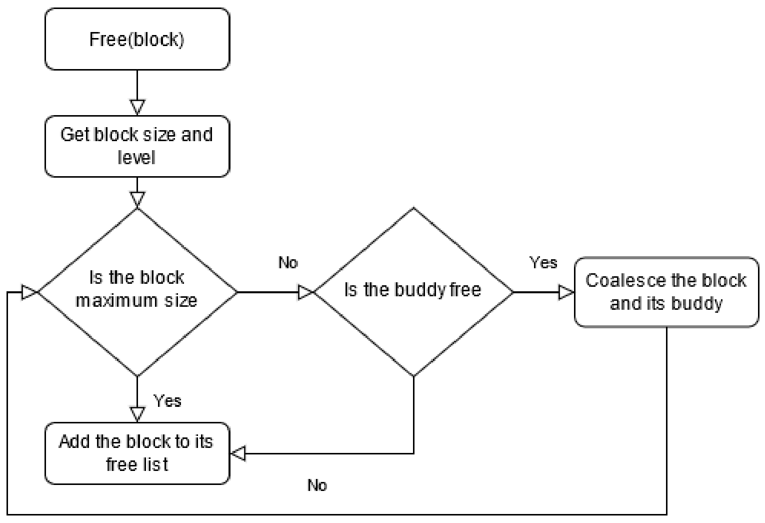
Free(block)
Get block size and level
Is the block maximum size
No
Is the buddy free
Yes
Coalesce the block and its buddy
Yes
Add the block to its free list
No

***Figure 6** Deallocation in binary buddy.*

The deallocation sequence is similar to the allocation sequence. It starts with finding the corresponding size level of the block to be freed. A block of maximum size can no longer have a buddy, so it is added to the free list as is. However, if the block has a free buddy, it is coalesced with it. This cycle is repeated until the block is either maximum size or the buddy is allocated and therefore cannot be coalesced.

The free operation is also quite fast on average. The block and its buddy are of the same size, so finding the header of the buddy is trivial. The worst-case for deallocation happens when a block of minimum size is freed, and all its buddies are also free. In this situation, the deallocation sequence consists of the maximum number of coalescing operations which is bound by the number of levels used in the allocator.

The implementation overhead for this type of buddy system is low. Each block header only needs to be 4 bytes to store all necessary information. Using boundary tags is also not required as the buddy system doesn’t support general coalescing. The free lists require one 4-byte pointer each in the heap’s management structure, but the rest of the list can be stored within free blocks themselves as described in chapter 3. The current allocator supports using multiple independent heaps which will multiply this heap header overhead for each new instance.

3.2 TLSF

Two-Level Segregated Fit (TLSF) is a general-purpose allocator designed for real-time systems devised by Masmano and colleagues [9]. Both allocation and deallocation are O(1) time operations. This is achieved with a combination of segregated fit using size ranges and bitmaps for searching. Additionally, TLSF aims to minimise fragmentation by implementing immediate general coalescing and a good fit policy. [4][9] It has been shown to produce low fragmentation while being competitive in performance in tests from multiple sources [7][24][25].

As the name implies, this allocator uses a two-level array to store many free lists of various sizes. The first level has block sizes arranged in powers of two. This level uses an offset to handle all blocks below a certain size outside of the main TLSF structure. In practice this can mean that blocks of size below 2^7 are considered small blocks and placed in a separate segregated list, then the first level of TLSF will have a free list for sizes $2^7 – 2^8$, $2^8 – 2^9$ ... $2^{30} – 2^{31}$. The location on this level is referred to as the block's first-level index (FLI). Each of these size ranges contains a second level of TLSF, which splits the sizes within each range further. The second level granularity is user configurable with recommendations being 16 or 32 size groups with corresponding second-level indexes (SLI). The second level does size range splitting evenly. [9][26][27] Figure 7 shows the TLSF data structure.

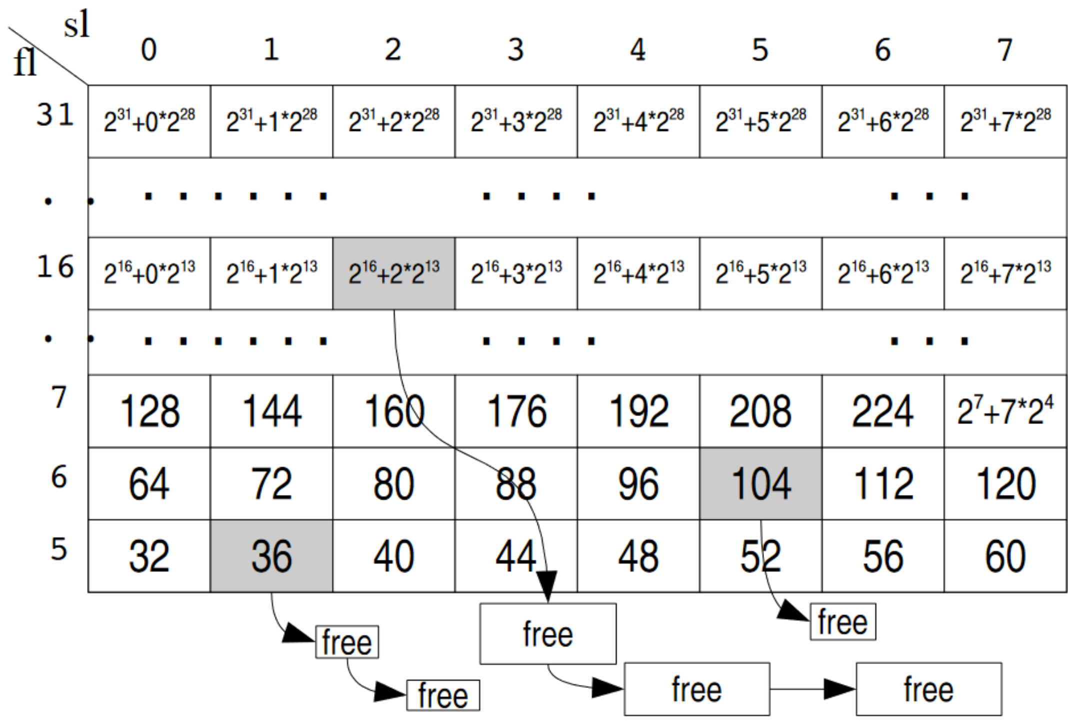

***Figure 7** TLSF data structure [27].*

In Figure 7 the TLSF data structure is shown with the first level vertically and the second level horizontally. The structure in Figure 7 only uses 8 second level indexes for demonstration purposes. [28] This does however illustrate a requirement for a minimum block size that can be managed in the TLSF structure. If there were 16 SLI in use instead, then FLI 5 would try to split the size ranges with only 2 bytes in between. On a 4-byte aligned system this would render half of the free lists useless. [1][28]

Figure 7 also exposes one major weakness of TLSF which is implementation overhead. Each of the free lists require a 4-byte pointer in heap management, which with FLI = 30, SLI = 16 and minimum size offset being 6 results in a total of 1536 bytes to manage the free lists. Additionally, TLSF requires one bitmap for the first level and a bitmap for each second level, further increasing the overhead. [28] The per block overhead is not as bad, with the original implementation needing 8-byte headers per used block as shown earlier in Figure 2 [9][26]. However, later implementations of TLSF principles, for example Matthew Conte's TLSF, are done with a 4-byte header to reduce overhead [29].

Allocation in TLSF starts by calculating the corresponding FLI and SLI for the requested size. This is done efficiently with processor instruction ffs to find FLI and bits following that determine the SLI. With these, the first non-empty list that holds only blocks larger than request is found, and its first block is removed. If the block is large enough to be split, the second half FLI and SLI are calculated, and it is placed in the appropriate free list. [28]

Deallocation starts by attempting to coalesce the block with its neighbours. Previous neighbour is found using boundary tags and the next neighbour can be found using the size of the freed block. Once coalescing is done, indexes for the new block are calculated, it is added to the right free list and the bitmaps are updated. [28]

From these descriptions, it is easy to find the worst-case performance for TLSF. Allocation does no list searches but instead finds the right free list with a bit scan operation and does one splitting operation. The worst-case for deallocation sees the block coalesce with both of its neighbours and adds it to a free list. None of these operations depend on the number of free blocks being managed, making TLSF fast in its worst-case. However, it also means TLSF will often perform at a level close to its worst-case if there is enough memory to split and memory gets freed frequently leading to coalescing operations.

3.3 Half Fit

Half Fit is a real-time dynamic memory allocator proposed by Ogasawara in 1995 to provide an alternative to buddy systems [3]. It has not gained much popularity as a viable allocator since it suffers from similar problems as binary buddy and is often considered strictly inferior to TLSF [21][25]. However, there are also words of encouragement from Masmano et al. "Half-fit is superior in all respects to Binary-buddy. Half-fit is faster than Binary-buddy and handles memory more efficiently" [6]. Since the aim of this thesis is to improve upon a variant of binary buddy, Half Fit is worth at least looking into.

Half Fit works on the same fundamentals as TLSF. It uses segregated free lists with size ranges divided in powers of two much like the first level of TLSF. The allocator then upkeeps a bitmap corresponding to each size range to quickly tell if it is empty or not. [3] Due to its simplicity, Half Fit is faster than TLSF and loses much less memory to implementation overhead [7].

Half Fit suffers from wasted memory that cannot be attributed to traditional external or internal fragmentation called *incomplete memory use* by Ogasawara. This problem arises from the way free lists are managed in combination with the allocation algorithm. The allocation algorithm only checks for free lists where each block is large enough to satisfy the request. In the worst case, this can lead to a situation where there are several blocks large enough to satisfy the request, but since their size is effectively rounded down to closest power of two in the free list, the allocation algorithm cannot find them. Due to this issue, Half Fit has unbounded memory use in the worst-case. [3] Incomplete memory use can also lead to an increase in fragmentation when the allocator is unable to reuse blocks as effectively. This has led to Half Fit performing poorly in most fragmentation tests [6][7].

One way to resolve the issue of incomplete memory use is to replace it with internal fragmentation. This is achieved by rounding up each request to a power of two. Now the allocator will not split blocks into sizes between powers of two which was causing suitably large blocks to not be found. The added internal fragmentation leads to a poor average fragmentation but avoids the potentially catastrophic unbounded worst-case. [21] This solution makes Half Fit more suitable for high integrity and hard real-time applications and has been successfully used for example in Kirienko's O1Heap allocator [30]. While good in terms of predictability, rounding up the requests to powers of two means this allocator suffers from similar fragmentation profile as binary buddy.

Another way to avoid incomplete memory use is to add a mechanism to the allocation algorithm to check the free list that can have blocks of the right size. At its simplest form this can mean going through the whole free list, or stopping early if a block is found, before taking a block from the free list of larger blocks. Such a solution would sacrifice constant execution time during allocations for much better memory efficiency. This could be made to execute in bounded time by only allowing the algorithm to check a specific number of blocks in the free list before giving up and taking the larger one. Another possibility would be having the free lists be sorted or indexed in a manner which allows faster searching. This type of solution has been considered for TLSF and shown to provide slightly better performance and memory efficiency [31].

Half Fit as originally proposed by Ogasawara is not suitable for our needs. It suffers from unbounded memory use making it too unreliable. The suggested trading of incomplete memory use for more internal fragmentation is also not promising. The rounding up of requests leads to it having comparable fragmentation to binary buddy. However, the lastly discussed improvements using some mechanism to search through the smaller block free list has some merit to it. The heap sizes managed by our allocator will be small, so even bounding the search to a low number of blocks is likely to be able to go through most of the free list.

3.4 Half Tree

Half Tree is a new dynamic memory allocator designed in the scope of this research. It is built upon the foundations of Half Fit and aims to solve the problems with incomplete memory use while preserving good enough performance and offering a low-bounded worst-case execution time. In addition to Half Fit, the allocator draws inspiration from dlmalloc. Dlmalloc was not considered as a suitable allocator by itself, due to it being aimed more at general-purpose computers with larger, expandable heaps. However, due to its popularity and influence it was examined thoroughly. Half Tree implements a similar binary tree structure for free lists and a slightly varied *designated victim* (DV) system being used in dlmalloc.

Half Tree implements a binary tree structure for free lists in order to make searching them for a large enough block faster. More specifically the tree structure is a bitwise trie where each level looks at the next bit in a memory block’s size to decide whether it should be placed in the left or right branch. Figure 8 has an example of how a bitwise trie works in Half Tree. The real allocator uses sizes aligned to 4 bytes but for illustration purposes smaller sizes are used in Figure 8.

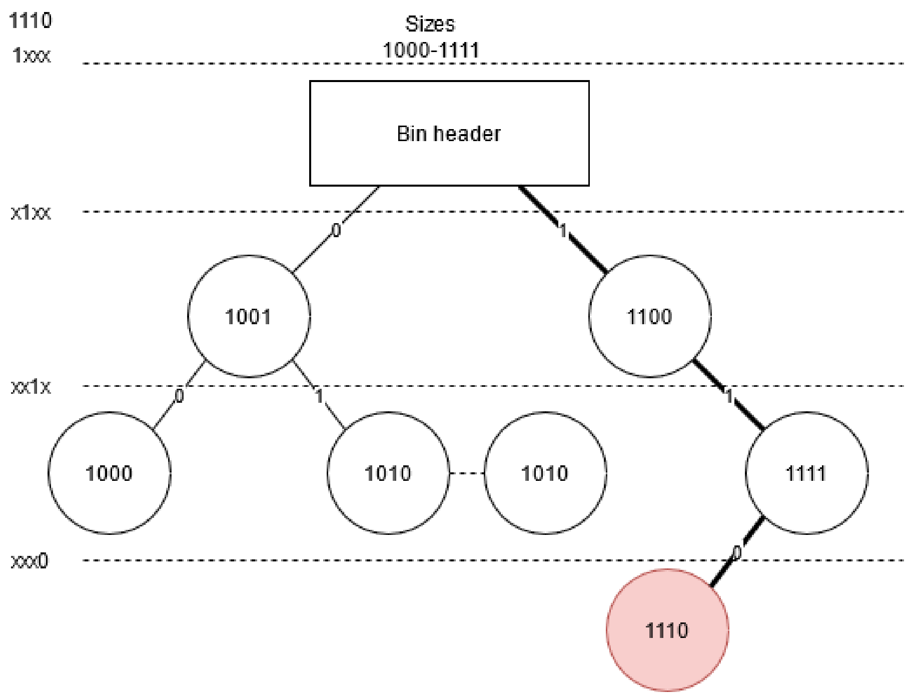

Figure 8 *Inserting a block in bitwise trie in Half Tree.*

In Figure 8 a block of size 0b1110 is being inserted into the trie. The bin where a memory block is to be placed is determined in the same way as in Half Fit using a rounded down base-2-logarithm. This is done efficiently with ffs instruction finding the most significant bit set to 1. The next level examines only the bit following the first set bit to determine if the block goes to the left or right. The level after this examines the bit after that and so forth. One additional feature of the data structure is seen in Figure 8 with blocks of size 0b1010. If the tree already contains a block of the same size as the one being added, it is not placed in the main trie but instead added to a linked list of same sized blocks. [32]

A common problem with various tree structures is that reordering the tree will take a considerable amount of time, or the tree will have an effectively sequential worst case if the balance is not upkept [16]. The structure of bitwise trie means the tree in any bin has at most as many levels as there are bits in the size of the memory, leading to a low-bounded worst-case. The bitwise trie used in Half Tree does not need to be balanced to ensure the tree does not grow uncontrollably due to this inherent maximum number of levels. Additionally, the total amount of memory required to fill the levels grows exponentially as the number of potential levels increases.

Removing a block and restoring the tree is a simple and relatively fast operation in Half Tree. There are 3 basic states the block being removed can be in, each leading to slightly different remove and restore procedures. First, the block can have others of the same size linked to it as is the case in Figure 8 0b1010. To remove and replace the block in the tree simply make sure one of them retains the tree state and the other can be removed without affecting the structure. Second, the block can be a leaf node in which case it can simply be removed without further action, example 0b1000. The last possibility is the block being one of the internal nodes, example 0b1001. A node can be replaced with any of the blocks under it without breaking trie ordering. [32] In Half Tree the leftmost leaf in the block's subtree is used to replace the block to promote a better fit for next allocation.

The last part of the data structure is the designated victim. In dlmalloc, DV is the leftover part of a split operation which is not yet placed into the main data structure but instead held on to in case it can be used again soon after [32]. Half Tree extends this functionality to also be able to hold a recently freed block. Both allocators implement a system to replace small DVs, determined by comparing to a *small DV threshold* value, to increase the likelihood of reuse [32]. Dlmalloc implements a more fundamental wilderness preservation system which also prevents the last chunk of memory from becoming a designated victim after a split [32]. This functionality is mimicked in Half Tree by not allowing unnecessarily large blocks to become the DV. This limit can be tuned to suit the target application's allocation sizes.

Allocation in Half Tree works in 3 steps. First, use the DV if it is large enough. If this fails, find the bin index where a block of the requested size would be in. This bin is then searched through, with priority always being on the right branch, until a large enough block is found, or a leaf is reached without finding one. Should this search also fail, use ffs to find the next suitable bin with the help of a bitmap similarly to Half Fit and take the first block from that bin, with priority on the left branch. In each of the possible cases, a large enough block is split, and the remainder is placed in a tree or becomes the new DV if it meets the criteria.

Half Tree supports general coalescing by using boundary tags. When a block is deallocated, it is first coalesced with its free neighbours. Special care is given to the situation where the DV is one of these neighbours, but it will be coalesced nonetheless. If the DV was coalesced, or there was no DV to begin with, the freed block becomes the new DV. Alternatively, the freed block may be used to replace a small DV or be placed directly into the main data structure.

Major weakness of Half Tree is its large minimum block size. Including all the required pointers to create the tree structure, along with size information and the footer, the smallest block Half Tree can manage is 24 bytes. The header of a used block can be as small as 4 bytes but if allocation requests are small enough, the minimum size can lead to large amounts of internal fragmentation. This however does not lead to issues on the target device, since it does not allocate many blocks smaller than 24 bytes. Furthermore, the minimum block size is still vastly improved from binary buddy’s 48 bytes.

Half Tree is a new memory allocator with a design goal of being more memory efficient than Half Fit and faster than TLSF. It implements a good fit policy, albeit much further from best fit than for example TLSF. The free lists are organized with a loose FIFO policy since older blocks are more likely to be found first. Pseudocode for Half Tree can be found in Appendix A.

4. TESTING TECHNIQUES AND METHODOLOGIES

To evaluate the allocators’ capabilities in the target application, we put them through various tests. This chapter will go over everything related to the testing setup starting with the platform used to run them. Following this, we define the parameters for quantifying performance and fragmentation. Next, the memory traces are introduced, and test implementation and measurement methods are discussed. Lastly, allocator implementation specifics used in the tests are defined.

4.1 The testing platform

All tests were done on a U-Blox M9 standard precision GNSS chip. To avoid regular operation of the chip interfering with results, a special build was used to excludes most of the firmware and operate as close as possible to hardware only. This is a more constrained environment than using a general-purpose PC, but still allows for extensive enough tests to be ran.

This decision was made for two main reasons. Firstly, by performing the testing on the target device the results will be more representative of the real use case. It will use the same hardware, same instruction set architecture and the same libraries and software framework. Secondly, a side objective of this work was to provide a dynamic memory benchmark to use in the future for the same chip. By implementing the testing framework within the confines of the firmware, it makes achieving this goal easy.

4.2 Define measurement parameters

Measuring performance is straightforward. As discussed earlier in chapter 2, the most interesting aspects are average allocation and deallocation time as well as the worst execution times for each. For these tests we do not directly use processor cycles or instructions per operation but instead use a separate hardware timer counter on the chip. It acts much akin to clock cycles and allows for easy comparisons between allocators. When the text refers to cycles, it is these counter cycles it is referring to.

Quantifying fragmentation is a much more complicated issue. The most widely used method was introduced by Johnstone et al. as being the ratio between the maximum amount of memory used by allocator and the maximum amount requested by the user, or *live memory* [10]. This can be expressed with formula

$$\frac{used\ memory - live\ memory}{live\ memory} * 100\% \qquad (1)$$

Figure 9 shows a live memory trace compared to memory used by the allocator.

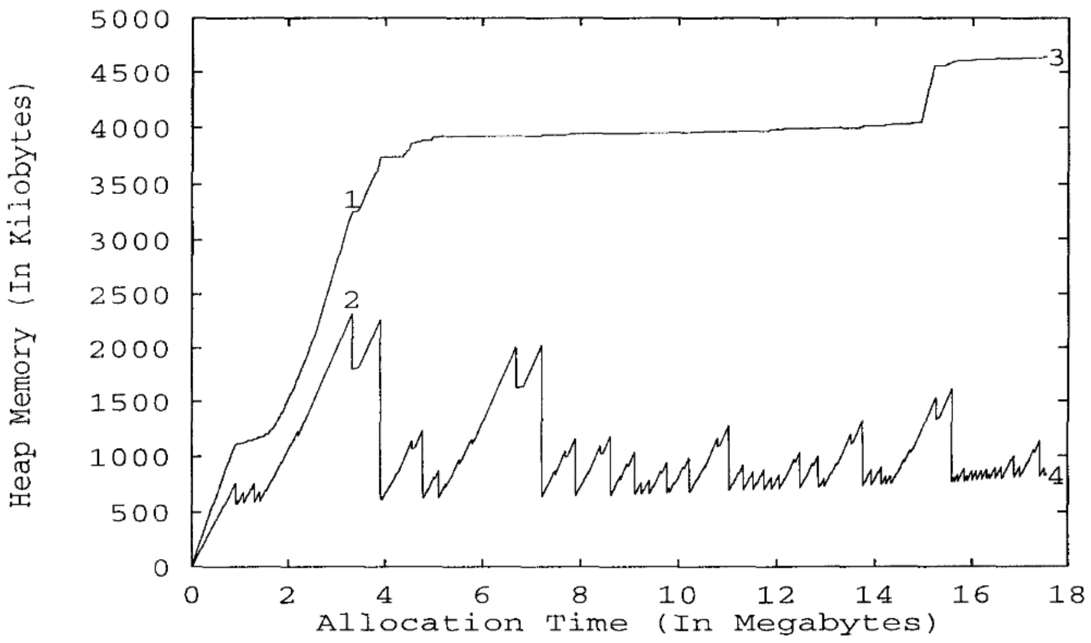

***Figure 9** Measurements of fragmentation. The lower line plots live memory while the upper line represents used memory [10].*

From Figure 9 points 2 and 3 mark the maximum live memory and maximum used memory. While this has established itself as the most used method of measuring fragmentation, Johnstone and colleagues also proposed 3 other methods. From Figure 9 those would be ratios between points 1 and 2, points 3 and 4 or taking the average ratio over run time. [10] Each of these have their merits and pitfalls, but for this work we will be focusing on the one discussed first. It appears to be the most balanced and given its popularity, will align the results better with research in this field. The main weakness of this method as stated by Johnstone et al. is, that it can give a deceptively low result if the point of maximum used memory has low live memory [10]. However, it should still give a fair enough result to compare allocators between each other and can be combined with other parameters to account for its weaknesses.

One interesting measure of fragmentation we use is *Cost Metric* introduced by Rosso. Cost Metric is measured in bytes and means the smallest possible heap size that can service all incoming requests. This is specifically designed for constrained devices and provides a good number for comparison, since it includes all aspects of memory waste from implementation overhead to different types of fragmentation. [33] As well as finding the Cost Metric itself, it will be used to run other fragmentation tests on the minimum heap size.

Finally, another parameter used in testing is the *Smallest-Biggest Block Metric* (SBBM) introduced by Rosso. This metric tracks the biggest block available for the allocator at any given time and the result is the smallest of these over the duration of the test run. SBBM can give more insight into how efficiently the memory was utilised and how much more memory could have been requested without failure at any point in the allocation sequence. This is especially interesting when combined with a small heap from Cost Metric. [33]

4.3 Memory traces

Memory trace refers to a specific sequence of allocation and deallocation requests. Throughout research on dynamic memory allocation a variety of different types of traces have been used in testing. Earlier work focused heavily on synthetic, mostly randomised, traces which were not great at replicating real program behaviour. In the survey by Wilson et al. they strongly advised against the use of randomized synthetic traces and encouraged a shift towards more accurate synthetic traces or better yet, real traces from real programs. [2] This has resulted in more recent work largely favouring the use of traces from real programs, typically things like compilers, scripting language interpreters or simulation and optimisation tools [10][31].

The tests in this thesis use a combination of synthetic and real traces. The synthetic traces are mostly naive, rather simplistic, traces designed for a specific purpose. Their inclusion is motivated by the forementioned benchmark for future use as well as to potentially find what type of requests the allocators might be struggling with. The main comparisons should be made with the number of real traces, gathered from device run time. These traces provide a result closer to the real use case, or at least a snippet of it.

The inspiration for types of synthetic traces to use was from Masmano et al. [7]. For these tests, synthetic traces are generated using pseudorandom numbers from *rand_r()*. The sequence from rand_r() is reset with a known seed before each test it is used in to make the tests fair for each allocator and each individual test run. Table 1 shows the traces and their characteristics. Time to live is given in number of allocation operations before the current one will be deallocated.

Table 1 Synthetic traces.

Trace	Size range (bytes)	Time to live (in allocs)
Small	8 – 376	0 – 35
Large	2000 – 12000	0 – 3
Random	8 – 6136	0 – 10
Typical	48 – 2000	0 – 15
Worst	varies	varies

The first two traces in Table 1 are mostly self-explanatory. They use only relatively small or large blocks. Similarly, the random trace uses a wide range of allocation sizes and a longer time to live than the large trace. For random and large traces, the time to live is limited by the heap size, since we do not want them to fail allocations even with the worst allocator. The typical trace resembles the allocation sequence from device run time with a bias towards smaller allocation sizes. Lastly, the worst trace is different from allocator to allocator and attempts to create a sequence leading to worst performance for that allocator. This trace is not used for fragmentation testing.

From small, large and random traces it can be seen that the upper limit for block sizes aligns with the sizes used in Buddy allocator. For large trace this is a hard limit since the implementation of Buddy cannot handle larger blocks. For the others, these limits were chosen to avoid Buddy performing worse than average. When looking for an improvement, we want Buddy to perform at least as well as on average but if it performs better, that is not harmful.

The real traces were compiled by adding instrumentation code to the firmware and instigating desired states. The first trace was taken during stable run time and tracks approximately a minute of allocations and deallocations (3453 allocations). This is called the **stable trace**. Next trace, the **boot trace**, includes the first 400 allocations from the device being rebooted. The last two traces were taken from the device while putting strain on dynamic memory. First one is **short stress trace** (463 allocations) and focuses more closely on the largest allocations happening on target device. Second trace, named simply **stress trace** (2000 allocations), is a longer trace with more features enabled.

4.4 Test infrastructure and implementation

The constrained testing platform requires some compromises and workarounds from the implementation. One of these is the heap size used for the tests. Performance tests and main fragmentation tests use a single heap of 86064 bytes. This size was chosen to result in Buddy allocator having only blocks of maximum size initially and to be large enough for meaningful testing. Fragmentation tests are also run with the minimum heap size found from Cost Metric to see how extremely constrained heap size might affect the fragmentation. Before each test, the allocator is returned to its initial state with an empty heap.

Performance testing is done simply by checking the value of the timer counter before and after an operation, adding the difference to a running total, and finally calculating the average. Additionally, the slowest allocation and deallocation are kept track of. Number of allocations and deallocations for synthetic trace performance tests was set at 2500. This was determined to be large enough to include enough variations in sequences due to their simplicity.

To measure fragmentation, we need to know how much memory the allocator uses for the test run as well as how much memory is requested. The test keeps track of the highest amount of live memory it has requested as well as keeping track of the biggest block available after each allocation, for SBBM. For each block allocated it checks the start and end addresses. It is worth noting that the block might be larger than requested and the real size needs to be checked from the allocator before calculating end address. This check includes header and footer overhead in the block size, meaning they will contribute to the total measured fragmentation. By calculating the difference between the highest and lowest addresses touched by the allocator, we can determine the total memory used. Finally, formula (1) is used to calculate the total fragmentation.

This implementation of fragmentation measurement works well in most cases, since the allocators under test attempt to keep the used memory compacted to one end. However, there are some cases where this can prove problematic for the limited block size Buddy implementation. Since it uses a LIFO policy, it uses memory in the heap starting from the last block added. In a case where just barely more than one maximum size block of memory is needed to service all requests this leads to a high measured fragmentation. Figure 10 shows one such scenario.

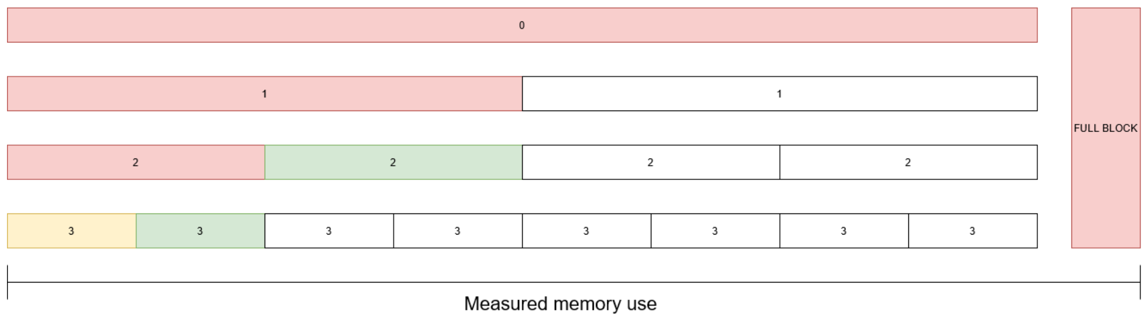

***Figure 10** Example of disproportional fragmentation in Buddy allocator. Red blocks are in use, green ones are free and yellow is the block that will be allocated.*

In Figure 10 there is no more memory left in the previously used full block on the right, so the next request for a level 3 block is taken from the next maximum size block on the left. Now, the allocator will split the new block down to size and allocate the furthest away block from the last full block. This unveils a weakness in fragmentation measurement where the real fragmentation could be very low, assuming there is no internal fragmentation, but the result would still be very high. This high result for fragmentation is not proportional to the harmfulness, or its likelihood of manifesting as failed allocations, of it since the memory within each maximum sized block is mostly independent of each other.

There are also other possible scenarios where the Buddy allocator will show higher fragmentation than it is affected by. One example experienced during testing, for the curious it can be most clearly seen in synthetic typical trace in Appendix B, would be switching over to using the second half of a block. Depending on the sequence, it is possible that Buddy starts allocation from the left side, but once it fills up, switches over to the right side and ends up using that for the rest of the run while the left side is freed up. This leads to the measurement considering the start address to be on the left side, while the end address is at the end of right side. At most the allocator used barely over half of the memory in the block, but due to the side switch, used memory is measured as a full block being in use. This is again not overly harmful since the memory is still as compact as before, but the resulting fragmentation measurement will be much higher.

Luckily, the problems with Buddy and fragmentation measurements are alleviated by the second half of the fragmentation tests. The problems stem from having a larger than necessary heap size and are conveniently solved by finding the Cost Metric and rerunning the test with a minimum size heap. Looking at the situation in Figure 10, with minimum sized heap it would be one maximum size block, to the right, and one block of level 3, to the left. Figure 11 shows the same allocation as Figure 10, but with minimum heap size instead.

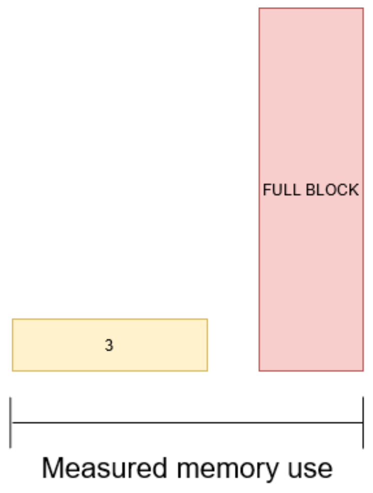

Figure 11 *Same situation as in Figure 10 but with minimum heap size.*

Now in the situation shown in Figure 11 the smaller block would be immediately next to the full block resulting in fragmentation measurement working as intended. This means the fragmentation results from Buddy with a large heap should be taken with a grain of salt, especially if they differ greatly from results with Cost Metric.

As stated above, all fragmentation tests are run with both large and minimum size heaps. Originally, as described by Rosso, Cost Metric is found by shrinking the heap size until an allocation fails [33]. In the interest of speeding up the tests and ensuring the heap size is truly the smallest possible the direction is reversed in our testing. The allocator will start with a heap that is too small for the sequence and increment it by 8 bytes until no failures occur. Due to the nature of dynamic memory allocation, it is possible that there are heap sizes which fail between Cost Metrics found from different directions. All other fragmentation tests are then repeated on this small heap.

4.5 Allocator implementations

The *Binary Buddy* allocator was used mostly as it is in the firmware. However, some features had to be removed to make it work on a more barebones build, but the core functionality remains the same. Additionally, any unnecessary state tracking code was removed since it would not be included in the other allocators either. As stated above, multiple heaps were not used in the testing, but support for them was left in and is also included in all other allocators built upon the same framework as Buddy. This introduces a slight performance penalty over not having it but is fair between the allocators that do have it.

To provide a point of reference and act as a baseline, two simple sequential allocators were implemented. These two are *First Fit* and *Best Fit*. Neither of them can be considered as viable allocators for our use-case due to their unbound worst-case performance. The motivation for First Fit's inclusion is to be the most simple and basic allocator possible. Best Fit on the other hand is considered very good in terms of fragmentation [7][10]. This makes it a valuable baseline for good fragmentation. Both allocators use LIFO policy and are implemented in the same framework as Buddy allocator with all the same features.

There are two interesting open source implementations of *TLSF* and due to the ease of porting them, both are included in the tests. The first one is the original implementation of TLSF 2.0 by its creators [26]. The second implementation is a popular, more recent version made by Conte [29]. One big difference immediately noticeable is Conte's TLSF has a 4-byte header whereas the original uses 8 bytes. Another notable difference is that Conte's version includes support for using multiple heaps while the original does not. Both versions use asserts for error checking, which were disabled in the test runs.

Each version of TLSF received some minor changes both to make them compatible with the test interface and better suited for the target application. A set of interfacing functions which call various TLSF functions were added to both implementations, possibly making them very slightly slower. Additionally, first level indexes of both were brought down to 18 since the heap size will not be large enough to require using the full 32 indexes. Second level indexes were also reduced to 8 for Conte's and 16 for the original. The original had some issues when attempting to run it with just 8 SLI so the larger number was used. This reduction in bins greatly reduces implementation overhead while not having significant impact on fragmentation when working with relatively small blocks. It is worth noting that since TLSF is not using the same framework as binary Buddy, it might be slightly faster or slower when made to work with that frame.

Half Fit as it was originally proposed by Ogasawara was not implemented due to issues with incomplete memory use leading to it not being reliable enough [3]. Kirienko's *O1Heap* allocator has recently been successfully implemented in a project and is open source [30][34]. It is a Half Fit variant that rounds all requests up to powers of 2 to provide high predictability and both bounded worst-case performance and fragmentation. It has been designed as a "predictably bad allocator" which makes it unlikely to be the best candidate for systems that don't value predictability over everything. [30] For our testing it acts as a substitute for the base Half Fit implementation to be used for comparison with other variants based on the same fundamentals. The open source implementation was modified to interface with our tests and to not upkeep diagnostics. Additionally, all asserts were disabled for the testing.

Next Half Fit variant, referred to as *Half Fit CSTM* from now on, implements a simple sequential search of the free list which might have a block of a large enough size. It does not include a threshold where it gives up the search and takes the large block found by base Half Fit operation, but this is trivial to implement if desired. It was mostly built from scratch to work in the same framework as the current allocator but uses some helper functions and takes guidelines from O1Heap. Additionally, the header size was shrunk down to 8 bytes from O1Heap's 16 bytes by implementing boundary tags. It also limits the number of bins to 16, with an offset of 4, making the maximum block size it can handle ~1 MB.

Finally, *Half Tree* allocator was built from the ground up, barring a handful of Half Fit helper functions from O1Heap, according to the design described in Section 3.4. It also works with the same framework as the Buddy allocator and its structure is based on the pseudocode from appendix A. The number of bins and the same offset as with Half Fit CSTM were used in testing. For the test, small designated victim threshold was set to 150 bytes and large DV boundary was set to 2800 bytes.

5. TEST RESULTS AND ANALYSIS

This chapter will go over the results of our testing. First, performance and fragmentation results are introduced and analysed separately. Second, the allocators are evaluated based on the results as a whole. The analysis will focus on real trace results, but a brief overview of synthetic trace results is provided. For the inclined, full results from synthetic traces can be found in Appendix B.

5.1 Performance results

Table 2 through Table 5 show performance in terms of execution time from real traces in order boot, stable, stress -short and stress trace. The column *average operation* takes the average from allocation and deallocation to provide a single number for comparison. This is further elaborated by comparing the change in average operation with the binary Buddy allocator.

Table 2 *Performance results in boot trace. Units are in timer cycles.*

	Average		**Worst**		**Average operation**	
	Alloc	**Free**	**Alloc**	**Free**	**Operation**	**Change**
Buddy	84	130	275	369	107	0,00%
First Fit	75	129	97	180	102	-4,67%
Best Fit	96	127	150	180	111,5	4,21%
TLSF (Cont.)	259	193	262	292	226	111,21%
TLSF (Masm.)	173	167	188	245	170	58,88%
O1Heap	78	71	80	101	74,5	-30,37%
Half Fit CSTM	108	136	117	176	122	14,02%
Half Tree	90	162	157	253	126	17,76%

Table 3 *Performance results in stable trace. Units are in timer cycles.*

	Average		**Worst**		**Average operation**	
	Alloc	**Free**	**Alloc**	**Free**	**Operation**	**Change**
Buddy	90	145	346	382	122	0,00%
First Fit	73	121	96	172	97	-20,49%
Best Fit	87	127	122	182	107	-12,30%
TLSF (Cont.)	259	199	262	292	229	87,70%
TLSF (Masm.)	175	170	188	245	172,5	41,39%
O1Heap	78	72	80	101	75	-38,52%
Half Fit CSTM	108	136	117	176	122	0,00%
Half Tree	106	135	151	243	120,5	-1,23%

Table 4 *Performance results in stress trace -short. Units are in timer cycles.*

	Average		**Worst**		**Average operation**	
	Alloc	**Free**	**Alloc**	**Free**	**Operation**	**Change**
Buddy	85	131	276	379	108	0,00%
First Fit	72	120	95	167	96	-11,11%
Best Fit	97	128	150	189	112,5	4,17%
TLSF (Cont.)	256	197	262	292	226,5	109,72%
TLSF (Masm.)	170	165	188	245	167,5	55,09%
O1Heap	77	71	80	101	74	-31,48%
Half Fit CSTM	107	134	121	176	120,5	11,57%
Half Tree	105	140	157	240	122,5	13,43%

Table 5 *Performance results in stress trace. Units are in timer cycles.*

	Average		**Worst**		**Average operation**	
	Alloc	**Free**	**Alloc**	**Free**	**Operation**	**Change**
Buddy	84	131	311	379	107,5	0,00%
First Fit	79	131	111	199	105	-2,33%
Best Fit	107	128	165	203	117,5	9,30%
TLSF (Cont.)	251	193	262	292	222	106,51%
TLSF (Masm.)	166	160	188	245	163	51,63%
O1Heap	75	67	80	101	71	-33,95%
Half Fit CSTM	107	132	125	176	119,5	11,16%
Half Tree	90	151	167	254	120,5	12,09%

The average operation times in real traces are compiled in Figure 12.

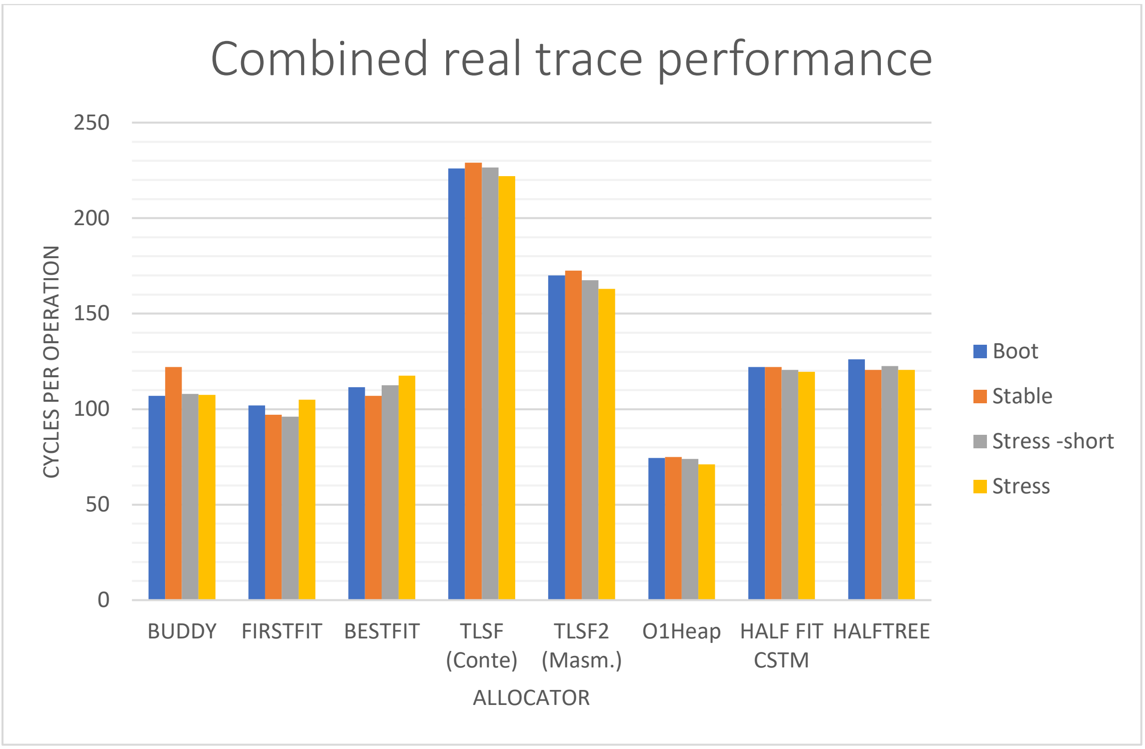

Figure 12 *Allocator performance in real traces.*

For the sake of brevity, only the average operation times from synthetic traces are listed in Table 6.

Table 6 *Average operation times in synthetic traces. Units are in timer cycles.*

	Small	**Large**	**Random**	**Typical**
Buddy	92,5	86,5	102	101,5
First Fit	113	102,5	111	110,5
Best Fit	141,5	110,5	126	125
TLSF (Cont.)	212,5	230	229	226,5
TLSF (Masm.)	151	178,5	175,5	170,5
O1Heap	64	73	72	69,5
Half Fit CSTM	117	122	122,5	120,5
Half Tree	129	126,5	130,5	116

The results from Table 6 are compiled in Figure 13.

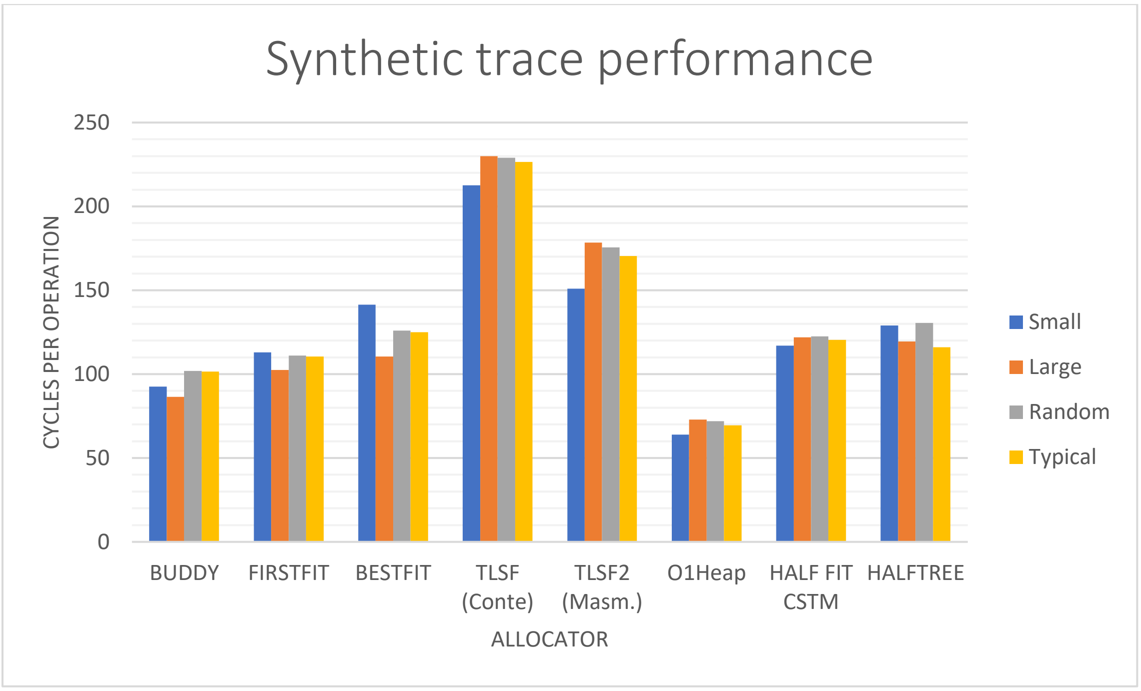

***Figure 13** Allocator performance in synthetic traces.*

Finally, Table 7 shows the worst-case performance of each allocator. The constant time allocators (TLSFs and O1Heap) did not have a designated worst-case trace but instead worst performance across the other traces was assumed to be the worst-case in general.

Table 7 *Worst-case performance of allocators.*

	Worst-case allocation	Worst-case free
Buddy	347	382
First Fit	6336	147
Best Fit	6336	147
TLSF (Cont.)	262	292
TLSF (Masm.)	188	245
O1Heap	81	101
Half Fit CSTM	4138	176
Half Tree	196	288

Additionally, Half Fit CSTM worst-case was tested with a threshold on when to give up search of smaller block size free list. With a threshold of 10 the worst allocation takes 157 cycles, with a threshold of 50 it takes 357 cycles, and with a threshold of 100 it takes 607 cycles.

5.2 Performance analysis

From the results in section 6.1, it is immediately clear that the Buddy allocator has great average performance for real traces, as seen in Figure 12. This is further improved in the synthetic trace results of Figure 13, but as mentioned in chapter 5.5, they have been designed in a way where Buddy is at least as good as its average. Out of the allocators examined, only O1Heap and First Fit have consistently better performance than Buddy in real traces.

Buddy allocator has the widest margin between its average and worst performance for any given trace. Unlike its great average performance, for all traces, excluding the worst trace, Buddy has the slowest worst execution times for both allocation and deallocation. With Puaut's declaration of worst observed performance being enough for soft real-time, all the tested allocators would improve upon Buddy [8]. However, having the theoretical worst-case be as catastrophic as First Fit, Best Fit and Half Fit CSTM in Table 7 is not acceptable. Luckily, Half Fit CSTM can have a threshold implemented to limit the severity of its worst-case, while not debilitating its regular operation.

Both versions of TLSF show much worse performance than the Buddy allocator. The version by Masmano is better in this regard than Conte's, at least partially caused by not supporting multiple heaps and having a larger header. The extra 4 bytes in the header contain a pointer to previous block, which is much faster than the boundary tags used by Conte. However, even Masmano's faster TLSF is still worse than Buddy in terms of performance with around 50% slower execution time across real traces.

O1Heap has excellent performance in all traces. It is consistently the fastest and even its worst execution times are competitive with the average of the second-best allocator. In addition, O1Heap has constant time complexity. With all these properties, O1Heap is the strongest allocator in terms of performance.

Half Fit CSTM and Half Tree have similar performance to one another with Half Fit CSTM being slightly faster on most real traces, with stable trace being the exception. The stable trace, results shown in Table 3, features memory block reuse more prominently leading to better utilization of Half Tree's designated victim mechanic, making it more effective there compared to other traces. This can be further seen in the typical trace in Table 6, which most closely resembles real program behaviour in stable run time out of the synthetic traces, having Half Tree perform noticeably better. For the real traces, both allocators mostly have 10–20% worse performance than Buddy, with Half Tree even outperforming it on the stable trace.

Based on performance alone, the most promising allocators would be O1Heap, Half Fit CSTM and Half Tree. O1Heap would be an improvement in performance while Half Fit CSTM and Half Tree have acceptable slowdowns depending on their fragmentation results. TLSF has its benefits in having constant time complexity but tends to perform at a level close to its worst-case too often. This leads to TLSF having poor average performance. It is not prohibitively slow, but slow enough to not be an enticing option.

5.3 Fragmentation results

Table 8 through Table 11 show fragmentation results from the real traces. The first column names the allocators used. Next three columns show the results, which are used memory, fragmentation percent calculated by Formula (1) and SBMM, for a large heap. The last 4 columns Include Cost Metric as well as the other fragmentation results measured with the smallest possible heap for that allocator and trace combination. This means the heap size for each allocator on the Cost Metric side can be different. The peak live memory for the trace remains constant and is given in the top left corner.

***Table 8** Fragmentation results in boot trace. Peak live memory for this trace is 14400 bytes. Units are in bytes.*

Live memory 14400	**Large heap**			**Cost Metric sized heap**			
	Memory	**Frag. %**	**SBMM**	**Cost Metric**	**Memory**	**Frag. %**	**SBMM**
Buddy	24576	70,67%	12288	21600	21552	49,67%	0
First Fit	17005	18,09%	69043	15552	15536	7,89%	20
Best Fit	15536	7,89%	70508	15560	15540	7,92%	834
TLSF (Cont.)	14952	3,83%	70581	15488	14960	3,90%	13
TLSF (Masm.)	15688	8,94%	69584	16472	15688	8,94%	16
O1Heap	22144	53,78%	63744	22304	22144	53,78%	0
Half Fit CSTM	15732	9,25%	70252	15816	15736	9,28%	0
Half Tree	15732	9,25%	70184	15880	15732	9,25%	0

***Table 9** Fragmentation results in stable trace. Peak live memory for this trace is 9382 bytes. Units are in bytes.*

Live memory 9382	**Large heap**			**Cost Metric sized heap**			
	Memory	**Frag. %**	**SBMM**	**Cost Metric**	**Memory**	**Frag. %**	**SBMM**
Buddy	24576	161,95%	12288	14112	14064	49,90%	0
First Fit	9910	5,63%	76138	9904	9888	5,39%	178
Best Fit	9654	2,90%	76390	9680	9660	2,96%	0
TLSF (Cont.)	9536	1,64%	75997	10088	9560	1,90%	37
TLSF (Masm.)	9776	4,20%	75496	10560	9776	4,20%	8
O1Heap	12736	35,75%	73152	13024	12864	37,11%	128
Half Fit CSTM	9696	3,35%	76288	9976	9696	3,35%	0
Half Tree	9792	4,37%	76124	10032	9884	5,35%	132

***Table 10** Fragmentation results in stress -short trace. Peak live memory for this trace is 11848 bytes. Units are in bytes.*

Live memory 11848	**Large heap**			**Cost Metric sized heap**			
	Memory	**Frag. %**	**SBMM**	**Cost Metric**	**Memory**	**Frag. %**	**SBMM**
Buddy	24576	107,43%	12288	16752	16704	40,99%	768
First Fit	15920	34,37%	70128	14296	14280	20,53%	1182
Best Fit	13850	16,90%	72194	13872	13852	16,91%	1124
TLSF (Cont.)	13728	15,87%	71805	14344	13874	16,34%	1097
TLSF (Masm.)	14184	19,72%	71088	14968	14184	19,72%	136
O1Heap	20640	74,20%	65248	20800	20640	74,21%	1344
Half Fit CSTM	13928	17,56%	72060	14008	13928	17,56%	1132
Half Tree	13924	17,52%	71992	14152	14004	18,20%	1072

Table 11 *Fragmentation results in stress trace. Peak live memory for this trace is 16037 bytes. Units are in bytes.*

Live memory 16037	**Large heap**			**Cost Metric sized heap**			
	Memory	**Frag. %**	**SBMM**	**Cost Metric**	**Memory**	**Frag. %**	**SBMM**
Buddy	24576	53,25%	12288	24096	24048	49,95%	0
First Fit	27841	73,60%	58207	17536	17520	9,25%	339
Best Fit	17092	6,58%	68952	17120	17100	6,63%	20
TLSF (Cont.)	17052	6,33%	68481	17256	16700	4,13%	377
TLSF (Masm.)	17176	7,10%	68096	17960	17176	7,10%	16
O1Heap	25856	61,23%	60032	26144	25984	62,03%	704
Half Fit CSTM	17340	8,12%	68644	17424	17344	8,15%	0
Half Tree	17340	8,12%	68576	17488	17340	8,12%	0

Fragmentation results for large heaps are compiled in Figure 14.

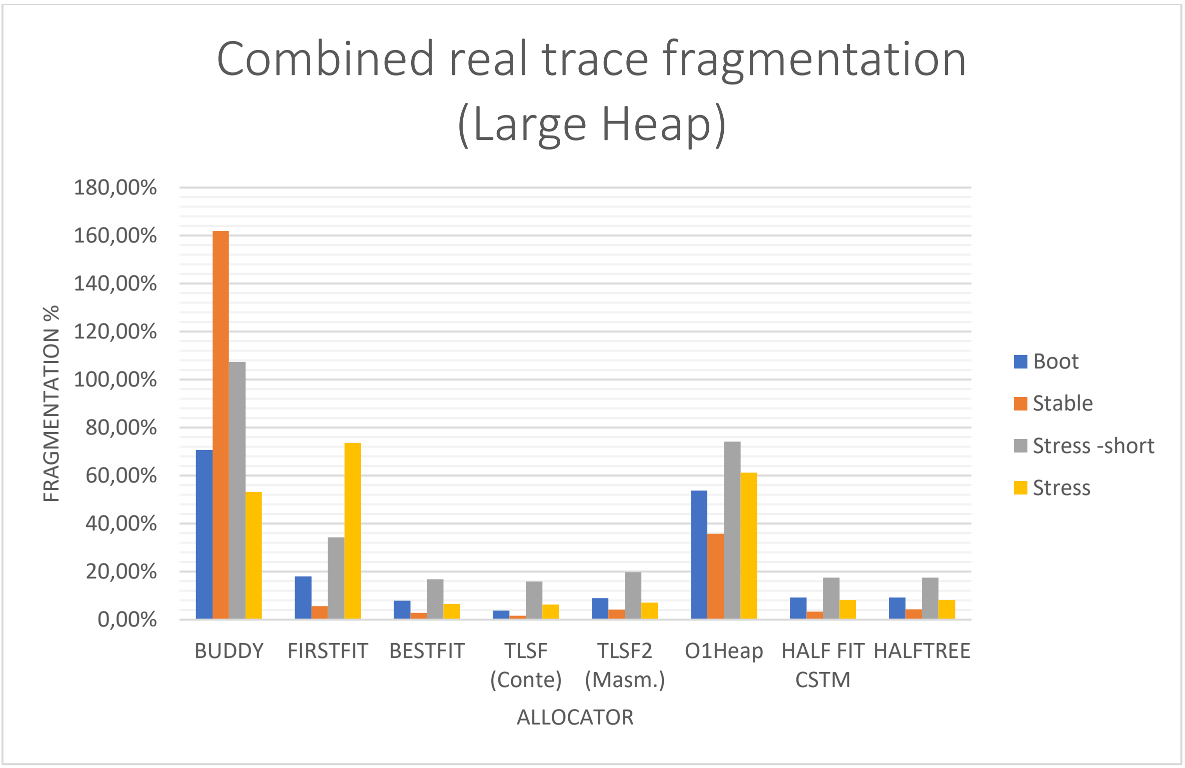

Figure 14 *Allocator fragmentation in real traces with large heap.*

Similarly, the fragmentation results for minimum size heaps are compiled in Figure 15. It is worth noting that due to Buddy allocator showing such high fragmentation with a large heap, the two figures have different scales.

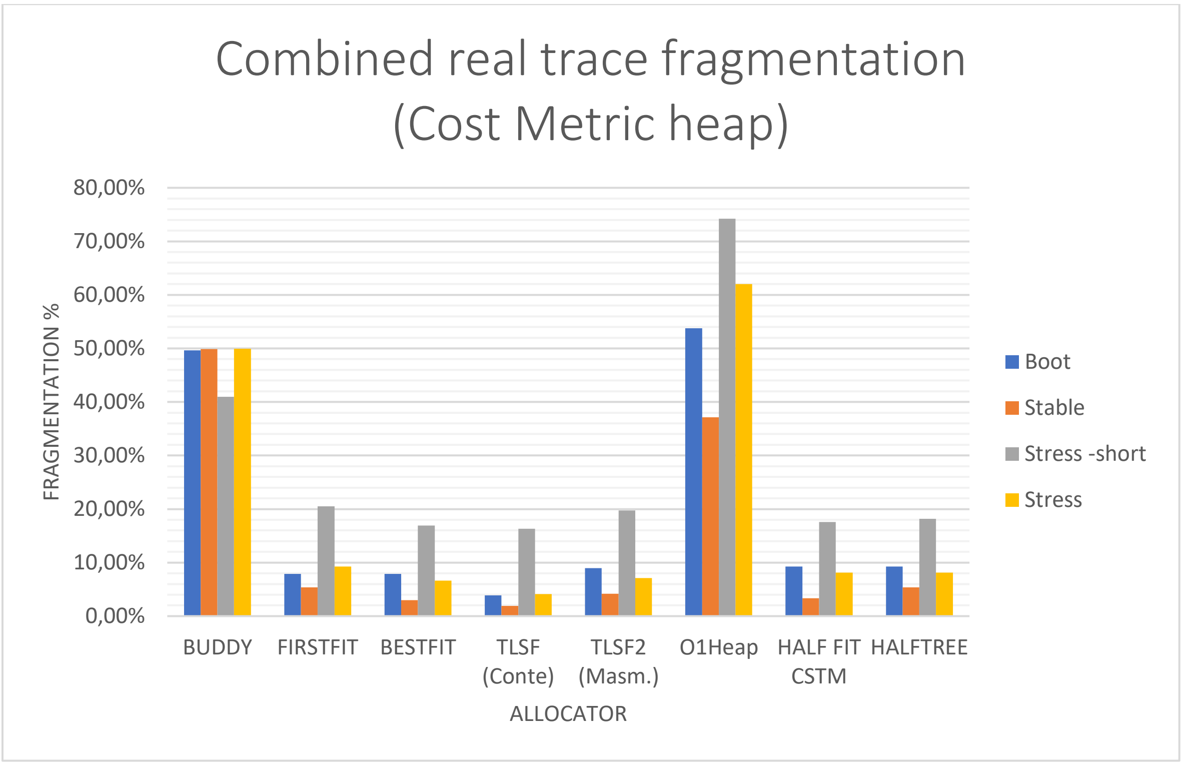

Figure 15 *Allocator fragmentation in real traces with minimum sized heaps.*

Cost metrics for real traces are shown in Figure 16.

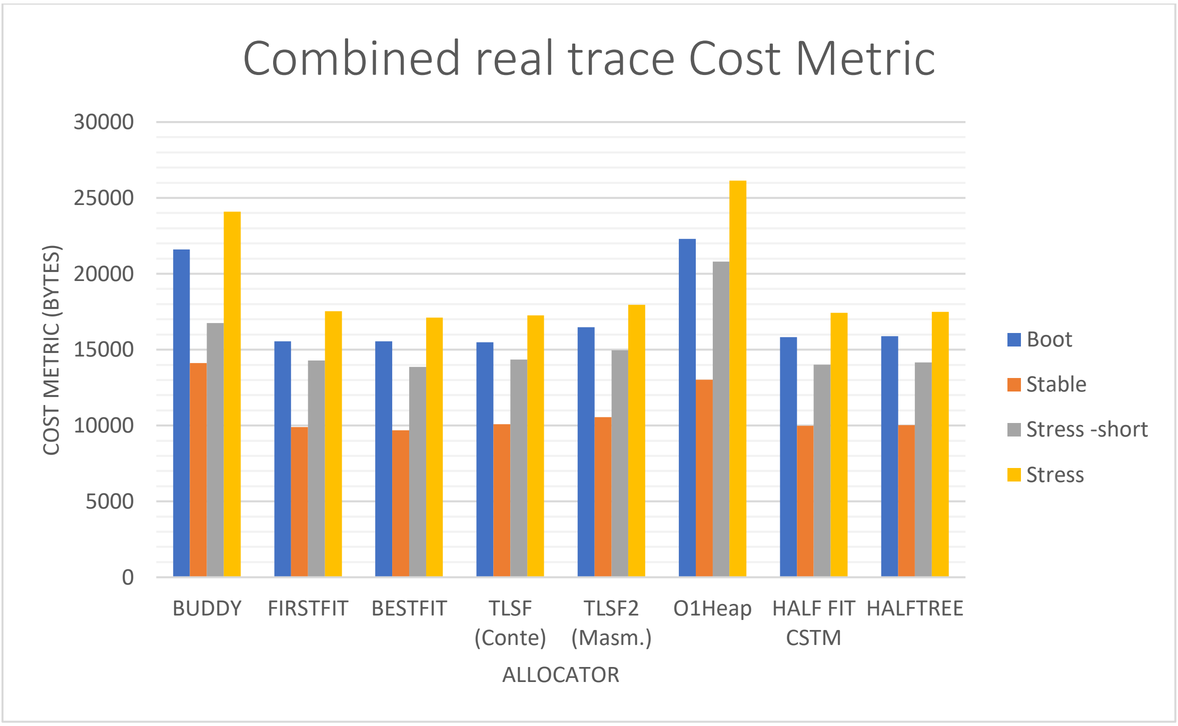

Figure 16 *Cost metric in real traces.*

To further illustrate the change in minimum heap required, Figure 17 compares the Cost Metrics of different allocators to the Buddy allocator’s Cost Metric.

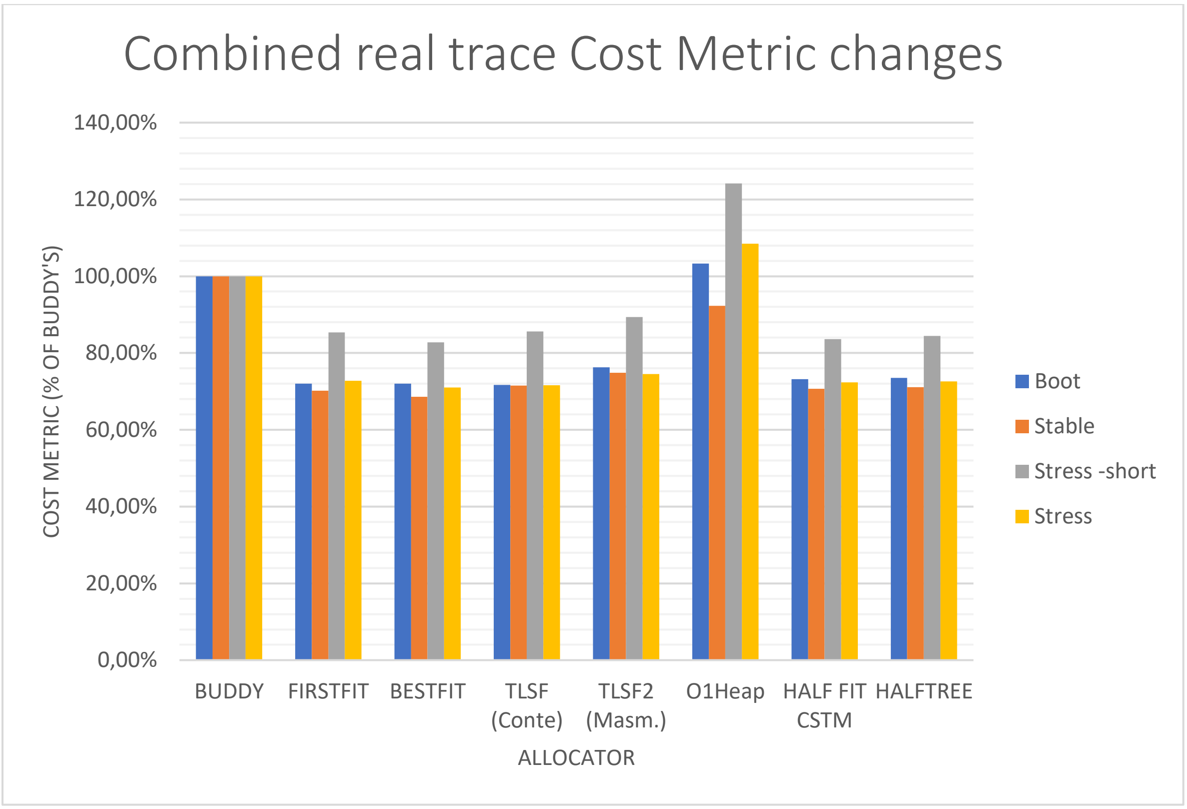

Figure 17 *Cost Metrics as a % of the corresponding Buddy allocator's Cost Metric.*

For the sake of brevity, only fragmentation percentages are listed in Table 12. Live memory differs for all traces but is omitted in this compilation. The full fragmentation test results for synthetic traces are available in Appendix B.

Table 12 *Fragmentation results from synthetic traces.*

	Large Heap				**Cost Metric Heap**			
	Small	**Large**	**Random**	**Typical**	**Small**	**Large**	**Random**	**Typical**
Buddy	46,24%	8,67%	65,78%	147,14%	28,79%	8,67%	49,20%	54,47%
First Fit	64,34%	22,04%	43,19%	84,57%	28,23%	22,06%	20,95%	42,56%
Best Fit	19,99%	22,04%	16,95%	25,30%	13,48%	22,05%	7,01%	6,92%
TLSF(Cont.)	18,82%	21,97%	26,79%	19,63%	11,62%	23,95%	26,22%	8,29%
TLSF(Masm.)	18,82%	25,37%	19,80%	28,56%	15,77%	24,64%	7,43%	9,57%
O1Heap	74,49%	141,49%	131,75%	87,93%	67.28%	81,12%	65,78%	87,93%
Half Fit CSTM	21,59%	22,08%	9,97%	19,55%	14,66%	22,09%	12,68%	10,06%
Half Tree	19,85%	22,08%	21,55%	26,23%	19,44%	22,08%	12,69%	10,94%

The results can also be seen in Figure 18 and Figure 19 for large heap and minimum heap respectively.

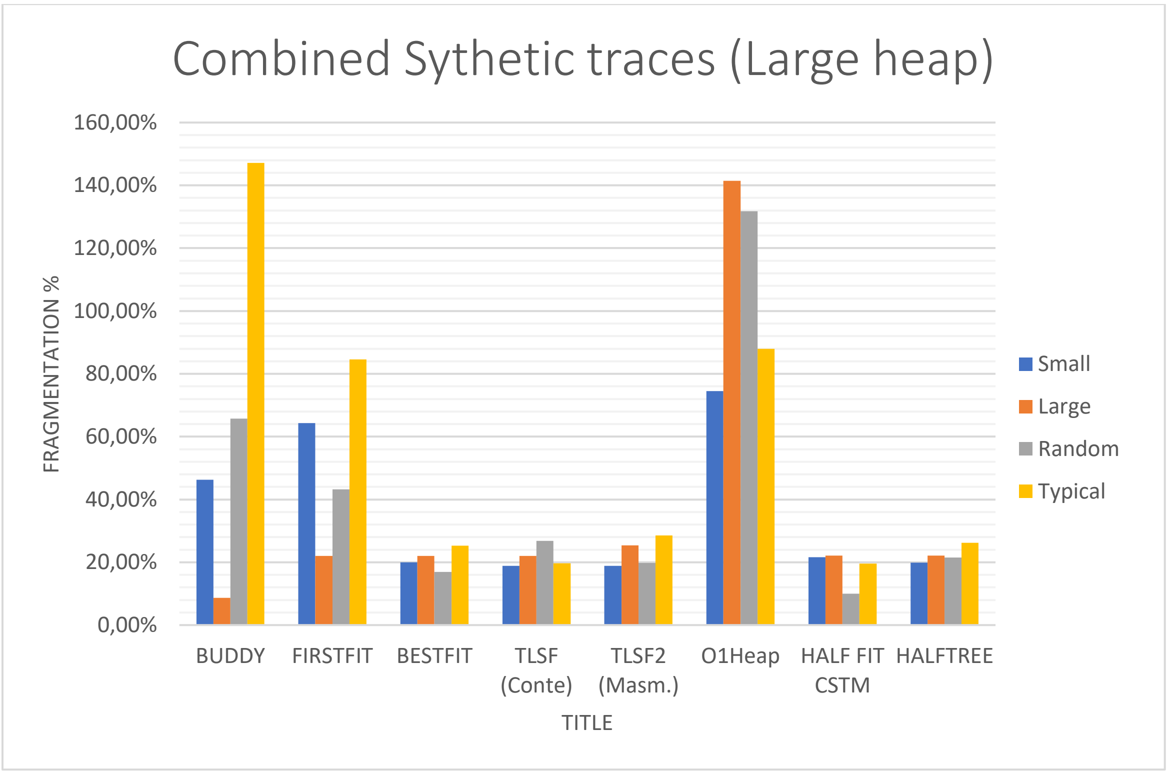

Figure 18 *Fragmentation results in synthetic traces with a large heap.*

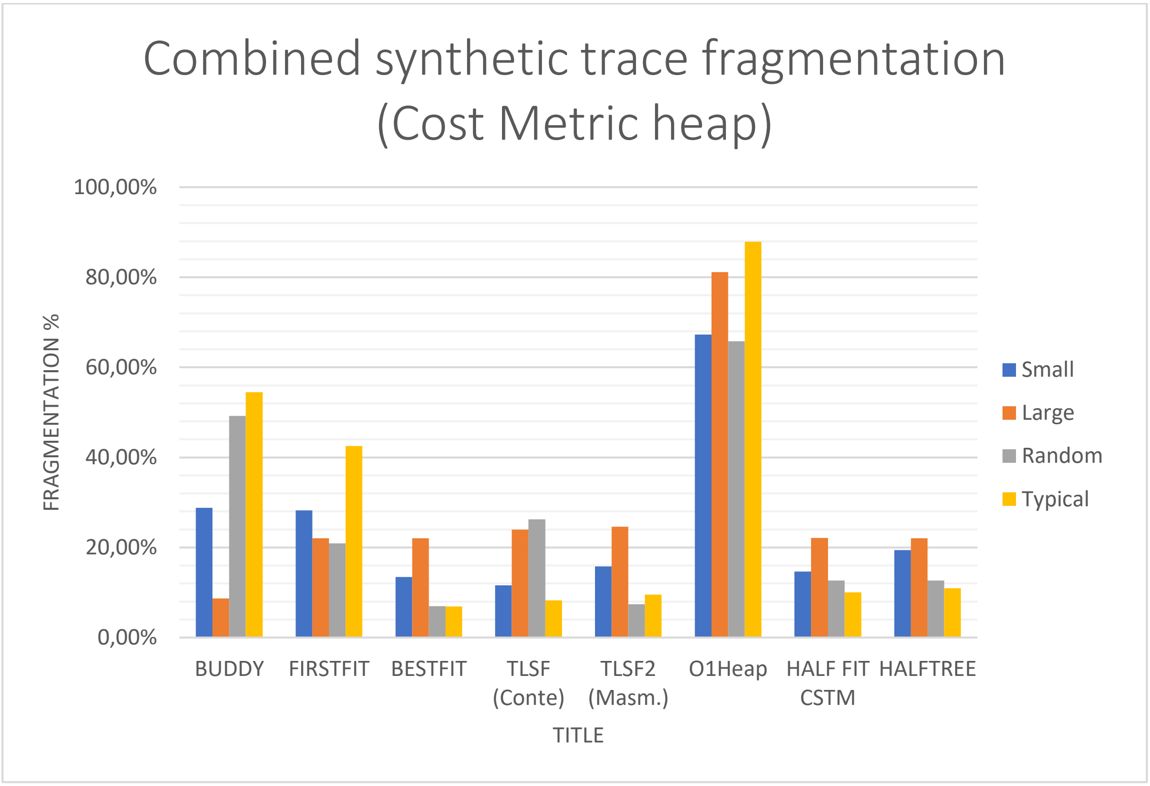

Figure 19 *Fragmentation results in synthetic traces with a minimum heap.*

Finally, the Cost Metrics are listed in Table 13 and the Cost Metric change related to Buddy’s is shown in Figure 20.

Table 13 *Cost Metrics in synthetic traces. Units are in bytes.*

	Small	**Large**	**Random**	**Typical**
Buddy	7488	36912	55344	7728
First Fit	7424	41424	44840	7104
Best Fit	6576	41424	39680	5336
TLSF (Cont.)	6976	42720	47328	5912
TLSF (Masm.)	7472	43064	40600	6232
O1Heap	9824	61600	61600	9504
Half Fit CSTM	6704	41496	41840	5552
Half Tree	7048	41560	41912	5664

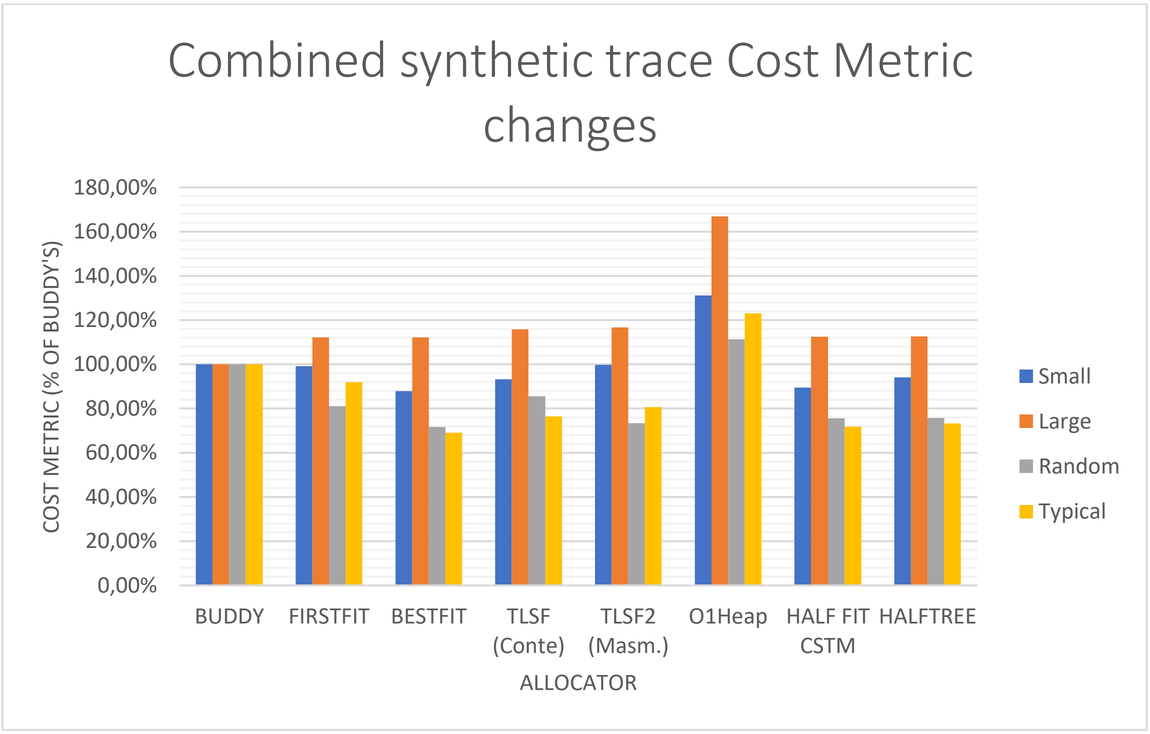

Figure 20 *Synthetic trace Cost Metrics as a % of the corresponding Buddy allocator's Cost Metric.*

5.4 Fragmentation analysis

Buddy allocator's problems with fragmentation are clearly seen in Figure 14 and Figure 15. The results with a large heap show some unrealistically high fragmentation as discussed in chapter 5, but even with the minimum size heap Buddy has approximately 50% fragmentation on all traces. As for the synthetic traces, the large trace seems to be an outlier where Buddy has exceptionally good fragmentation. This is a result of the small number of blocks allocated at once and maximum size being close to Buddy's maximum block size. Now Buddy will always use 3 maximum size blocks while live memory is the highest from a sequence of 3 allocations.

Buddy allocator has Cost Metric side SBMM of 0 in 3 of the 4 real traces. Since the allocator does not have any free blocks left but has high fragmentation, it can be determined that fragmentation is mostly internal. Other allocators tend to have more of a mix of internal and external fragmentation. In addition to Buddy, Half Fit CSTM and Half Tree have 0 SBMM in some traces but they also show low fragmentation. This is a combination of minimum block size preventing splits causing minor internal fragmentation and per block header and footer causing most of the wasted memory.

Stress trace -short is distinct in fragmentation from the other traces visible in Figure 15. For Buddy allocator this trace shows its best fragmentation results in real traces with Cost Metric heap. However, for all other allocators and heap sizes, excluding First Fit with large heap, this trace proves to be the most problematic real trace. This result is a likely indication that stress trace -short is relatively favourable for Buddy while also being unfavourable for the other allocators. Even still, most of them show far better fragmentation than Buddy in this trace. Another indication of the trace being unfavourable is the large SBBM values across the board in Table 10. Each allocator could have served a lot more requests without increasing Cost Metric, therefore reducing fragmentation, if the trace was more favourable for them.

From the baseline results of First Fit and Best Fit, we can see that they perform about as expected. First Fit is generally fine in terms of fragmentation but can vary wildly depending on heap size and allocation sequence, for example large heap stress trace fragmentation is high. As for Best Fit, it performed as well as hoped and is consistently among the allocators with the lowest fragmentation. Further, in most cases Best Fit has the lowest Cost Metric, meaning with implementation overhead included it is the most memory efficient allocator in these tests. These results serve as an indication that Best Fit is suitable for use as a reference point for good fragmentation results.

Both versions of TLSF show low fragmentation across the traces. Conte’s is generally slightly more efficient than Masmano’s even with the latter using a higher number of second-level bins. However, both are close to the level of Best Fit, with Conte’s having even lower fragmentation on the real traces. On synthetic traces, Masmano’s version does much better with random trace, likely due to using a higher number of SLI helping with more spread-out allocation sizes.

On the other hand, this low fragmentation of TLSF does not come without a trade-off. The high implementation overhead results in them having a relatively high Cost Metric, often higher than other allocators’ that have significantly more fragmentation. From Cost Metric side of the results, the per region overhead can be calculated by comparing used memory to the Cost Metric. Conte’s version has 528 bytes of overhead while Masmano’s has 784 bytes per region.

O1Heap has high fragmentation on all traces. Surprisingly, this fragmentation is not much worse than Buddy’s, especially on the real traces. Only the boot trace shows significantly increased fragmentation while the others are slightly worse or better. Synthetic traces on the other hand heavily penalise O1Heap’s aggressive rounding of sizes up and down to powers of two. It also has a relatively large SBBM on the stress trace in Table 11 which is a potential sign of it missing large enough blocks due to the rounding. In terms of average fragmentation, O1Heap falls short.

Half Fit CSTM and Half Tree show similar fragmentation on almost all traces, with Half Fit CSTM having slightly better fragmentation overall. The one significant outlier is large heap random trace, where Half Tree is noticeably worse. Half Tree works best with more real program -like reuse of blocks and the completely random blocks prove to be the most problematic for fragmentation as well. Even then, the fragmentation is still on acceptable levels and far outperforms Buddy allocator.

In terms of Cost Metric both Half Fit CSTM and Half Tree are better than Buddy on all but the synthetic large trace. They are also very close to the versions of TLSF when it comes to Cost Metric, being better in some traces and worse in others. However, the amount of per memory region Implementation overhead, 80 bytes for Half Fit CSTM and 148 bytes for Half Tree, is significantly lower.

From fragmentation test results it can be seen that most of the allocators offer a vast improvement over Buddy. Best Fit along with both versions of TLSF are the best allocators in terms of fragmentation. However, TLSF suffers from higher implementation overhead, especially if using multiple heaps. Half Fit CSTM and Half Tree are a bit worse than the previous 3 but not far off. First fit is often good but shows its weaknesses with wild variance even with this limited sample size. O1Heap tends to have worse fragmentation than the Buddy allocator.

5.5 Evaluation of allocator suitability

For an allocator to be a suitable replacement for the Buddy allocator it must meet the following criteria in order of importance. Firstly, the allocator needs to have a low bound worst-case performance. Secondly, the allocator must have lower fragmentation than the Buddy allocator. Lastly, the average performance of the allocator should not be drastically worse than that of the Buddy allocator.

As expected, the sequential allocators First Fit and Best Fit are not suitable replacements for Buddy. They fail on the first criterion with very poor worst-case performance. Best Fit does technically succeed in filling the other 2 criteria with its excellent fragmentation results and decent average performance. However, out of these 2 allocators, Best Fit is also more likely to drift towards its worst-case performance.

Both versions of TLSF excel on the first criterion with their constant time complexity. They are also outstanding in fragmentation. Unfortunately, their performance is severely lacking in comparison to Buddy allocator. With performance being the limiting factor when considering TLSF for our application, Masmano's version is the favoured. Conte's TLSF has lower fragmentation, but this is not worth the cost in performance. Another thing to consider is the large implementation overhead required by TLSF. When working with multiple heaps the total cost of overheads can prove too drastic. Therefore, Masmano's TLSF is not the ideal candidate, but can be considered as a suitable allocator to replace the Buddy allocator if the performance trade-off is deemed worth it and the number of heaps remains low.

O1Heap has really good performance and constant time complexity with a low bound. However, it has generally worse fragmentation than Buddy allocator. This makes it not suitable for our needs.

Half Fit CSTM has a lot of interesting properties. Without a threshold on when to give up a search of smaller blocks it has the same worst-case as the sequential fits. With such threshold set to 50 Half Fit CSTM has similar worst-case allocation to the Buddy allocator. However, stopping the search at a threshold opens the possibility for incomplete memory use, should a free list ever contain more than that number of blocks. While the free lists are unlikely to grow to 50 blocks, it is still a risk worth considering.

In the other 2 criteria Half Fit CSTM shows really good results. In terms of fragmentation, it offers a reduction in fragmentation of approximately 57–93% resulting in Cost Metric reduction of ~27%. Performance results are not necessarily ideal with Half Fit CSTM being 0–15% slower on real traces. However, this is a small price, especially compared to TLSF's 50% slowdown, for the significantly improved memory efficiency. Half Fit CSTM has potential as a suitable allocator, but it has issues with reliability making it less promising.

Half Tree improves upon the weakness of Half Fit CSTM while making as little sacrifices as possible in other areas. It has O(log(n)) time complexity that scales up slowly with heap size, resulting in a low bound worst-case execution time for a constrained heap. It also removes the possibility of incomplete memory use becoming an issue. As for the other criteria, Half Tree is marginally slower than Half Fit CSTM in most traces but only about 0 to 4 cycles per operation. Fragmentation shows similar results with Half Tree being very slightly worse with up to 2% more fragmentation.

All things considered, Half Tree is the most suitable allocator to replace the Buddy allocator. The main trade-off it has is lower performance, which is up to 18% on real traces. However, the 18% decrease is from the short boot trace. When comparing Buddy and Half Tree on long running stable trace, Half Tree even outperforms Buddy. In addition, it offers both faster worst-case allocation and worst-case deallocation. In terms of fragmentation, it offers sizeable improvements leading to approximately 27% smaller heaps being sufficient for the same allocation sequences. The per region overhead is larger, about 100 bytes more, but the improvements in fragmentation more than make up for this with reasonably sized heaps.

6. CONCLUSIONS

The aim of this thesis was to improve upon the dynamic memory allocator used in U-Blox GNSS chips. The focus area for optimisation was fragmentation, since the currently used Buddy allocator had major shortcomings in this area. Other key aspects to evaluate were performance and worst-case execution times, which we hoped would remain similar to those of Buddy's.

To get a more thorough understanding of dynamic memory allocation, a brief overview was provided in the beginning of the thesis. The fundamental building blocks were introduced as well as policies and techniques to improve fragmentation or performance.

To find viable alternatives to Buddy allocator, many different allocators were considered. The vast majority of the allocators designed for general-use computers are not suitable for a constrained embedded device. This may be due to unnecessary complexity, high overhead, reliance on the operating system to provide more memory as needed or a number of other factors. In the end, 2 allocators designed for real-time applications were chosen for further investigation, TLSF and Half Fit. After further investigation, the most interesting version of TLSF was Masmano's original. Respectively, the most viable variant of Half Fit was determined to be O1Heap.

In addition to the allocators found from other sources, 2 customised allocators were made for this thesis. First, Half Fit CSTM which operates similarly to Half Fit while attempting to fix its largest problems with fragmentation and incomplete memory use, or at least make them less likely to occur. Second, a new allocator called Half Tree, which combines design ideas from both Half Fit and DLmalloc. The design goal of Half Tree was to be faster than TLSF, more memory efficient than O1Heap and more reliable than Half Fit CSTM.

In order to compare the allocators to one another, a test framework was implemented on the target device. The allocators were evaluated on their performance, worst-case execution times, fragmentation and required heap size. From the tests each allocator's suitability was evaluated. Table 14 compiles the key takeaways from the analysis.

Table 14 *Summary of allocator suitability analysis.*

Allocator	Advantages	Drawbacks
TLSF	Very low fragmentation Lots of research O(1) time complexity	Slow Large overhead
O1Heap	Very fast O(1) time complexity	High fragmentation
Half Fit CSTM	Decent performance Low fragmentation	Potentially unreliable Slightly slower than Buddy
Half Tree	Decent performance Low fragmentation Low bound worst-case performance	Considerable overhead Slower than Half Fit CSTM

From the allocators in Table 14 most suitable were determined to be TLSF and Half Tree. TLSF had some major upsides, but also significant trade-offs. Half Fit CSTM has some issues with unbound worst-cases leading to it being less reliable than other candidates. On the other hand, Half Tree achieved all its design goals. It provided a middle ground of still achieving lowered fragmentation while mitigating the downsides of the other allocators. It still has considerable implementation overhead, but much less so than TLSF. This led to the Half Tree allocator being proposed as the replacement for the Buddy allocator.

Ultimately, the objectives of this research were met. The proposed Half Tree allocator has much lower fragmentation than Buddy allocator while not making drastic compromises in performance. Future work in the scope of U-Blox involves porting Half Tree allocator to the product firmware and testing it further there. Outside the scope of U-Blox, Half Tree allocator suitability for other embedded device’s memory allocation is worth considering. It is not necessarily viable for less constrained general-use computer applications and is likely to be a worse version of DLmalloc in such environments.

REFERENCES

[1] R.C Seacord, Effective C, No Starch Press, 2020, 272 p., ISBN : 1-0981-2567-3

[2] P.R Wilson, M. S. Johnstone, M. Neely and D. Boles, Dynamic storage allocation: A survey and critical review, International Workshop on Memory Management, Springer. 1995, 116 p., ISSN: 0302-974

[3] T. Ogasawara, An Algorithm with Constant Execution Time for Dynamic Storage Allocation, Proceedings Second International Workshop on Real-Time Computing Systems and Applications, IEEE, 25-27 Oct. 1995, pp. 21–25, ISBN: 0818671068

[4] M. Masmano, I. Ripoll, A. Crespo, Dynamic storage allocation for real-time embedded systems, Real-Time Systems Symposium, Cancun, Mexico, December 2003, 4 p., available: http://wks.gii.upv.es/tlsf/files/papers/tlsf_work.pdf

[5] D. A. Alonso, et al., Dynamic Memory Management for Embedded Systems, Springer, 2015, 243 p., ISBN : 3-319-10572-8

[6] M. Masmano, I. Ripoll, P. Balbastre, A. Crespo, A constant-time dynamic storage allocator for real-time systems, Real-Time Systems, vol. 40, no. 2 Springer Science+Business Media, LLC, Nov. 2008, pp. 149–179, ISSN: 0922-6443, available: http://www.gii.upv.es/tlsf/files/jrts2008.pdf

[7] M. Masmano, I. Ripoll, A. Crespo, A comparison of memory allocators for real-time applications, Proceedings of the 4th international workshop on java technologies for real-time and embedded systems, Vol.177, 2006, pp. 68–76, ISBN: 9781595935441, available: https://www.researchgate.net/publication/234785757_A_comparison_of_memory_allocators_for_real-time_applications

[8] I. Puaut, Real-Time Performance of Dynamic Memory Allocation Algorithms, Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS’02), IEEE, 2002, pp. 41–49, ISSN: 1068-3070

[9] M Masmano, I. Ripoll, A. Crespo, J. Real, TLSF: a New Dynamic Memory Allocator for Real-Time Systems, 16th Euromicro Conference on Real-Time Systems (ECRTS 2004), 2004, 10 p., available: http://www.gii.upv.es/tlsf/files/ecrts04_tlsf.pdf

[10] M. S. Johnstone, P. R. Wilson, The Memory Fragmentation Problem: Solved?, ACM SIGPLAN Notices, Vol. 34, No. 3, ACM, 1998, pp. 26–36, ISSN: 0362-1340

[11] D. E. Knuth, The Art of Computer Programming, Volume1, Fundamental Algorithms, 3rd edition, Addison-Wesley Professional,1997, 672 p. ISBN : 0-13-348878-0

[12] D. Lea, A Memory Allocator, December 1996, [internet, referenced: 2.2.2022], available: http://gee.cs.oswego.edu/dl/html/malloc.html

[13] K. C. Knowlton, A fast storage allocator, Communications of the ACM, Volume 8, Issue 10, October 1965, pp. 623–624, ISSN: 0001-0782

[14] W. T. Comfort, Multiword list items, Communications to the ACM, Volume 7, Issue 6, June 1964, pp. 357–362, ISSN: 0001-0782

[15] J. L. Peterson, T. A. Norman, Buddy Systems, Communications of the ACM, Volume 20, Number 6, June 1977, ISSN: 0001-0782

[16] C.J. Stephenson, New methods for dynamic storage allocation (Fast Fits), ACM SIGOP Operating Systems Review, Volume 17, Issue 5, October 1983, pp. 30–32, ISSN: 0163-5980

[17] Google, Tcmalloc, Source code, commit: 1ed14fb, available: https://github.com/google/tcmalloc

[18] Microsoft, mi-malloc Documentation, version 1.7/2.0, available: https://microsoft.github.io/mimalloc/

[19] R. Hempel, Umm_malloc, source code, commit: ed58a3d, available: https://github.com/rhempel/umm_malloc

[20] J. Evans, Jemalloc, [internet, referenced 23.3.2022] available: http://jemalloc.net/

[21] J. Heter, Timing-Predictable Memory Allocation In Hard Real-Time Systems, doctoral dissertation, Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes, Saarbucken, 2014, 183 p., ISBN 978-3-8442-8742-4, available: https://publikationen.sulb.uni-saarland.de/bitstream/20.500.11880/26614/1/diss.pdf

[22] R. Moore, eheap vs. dlmalloc, Micro Digital, Feb. 2016, [internet, referenced 25.3.2022] available: https://www.smxrtos.com/articles/eheapvdlmalloc.htm

[23] D. Yadav, A. K. Chaturvedi, S. Pansari, A. Krishnan, Memory Management: Tertiary Buddy System, 2nd WSEAS Int. Conf. (CEA'08), Acapulco, Mexico, Jan. 2008, pp. 46–49, ISBN: 978-960-6766-33-6

[24] T. B. Ferreira, R. Matias, A. Macedo, L. B. Araujo, An experimental Study on Memory Allocators in Multicore and Multithread Applications, IEEE, 12th International Conference on Parallel and Distributed Computing, Applications and Technologies, Federal University of Uberlandia, Brazil, 2011, 7 p., ISBN: 1457718073

[25] V. Heikkilä, A study on Dynamic Memory Allocation Mechanisms for Small Block Sizes in Real-Time Embedded Systems, Master's Thesis, University of Oulu, 2012, 67 p., available: http://jultika.oulu.fi/files/nbnfioulu-201302081026.pdf

[26] M. Masmano, I. Ripoll, H. Brugge, A. Scislowicz, TLSF 2.4.6 source code, available: http://wks.gii.upv.es/tlsf/files/src/TLSF-2.4.6.tbz2

[27] M. Masmano, I. Ripoll, A. Crespo, Description of the TLSF Memory Allocator Version 2.0, November 2005, 4 p., available: http://www.gii.upv.es/tlsf/files/papers/tlsf_desc.pdf

[28] M. Masmano, I. Ripoll, J. Real, A. Crespo, A. J. Wellings, Implementation of a constant-time dynamic storage allocator, John Wiley & Sons, Ltd, Software Practice and Experience, Vol. 38, Issue 10, 2008, pp. 995–1026, ISSN: 0038-0644, available: http://www.gii.upv.es/tlsf/files/spe_2008.pdf

[29] M. Conte, TLSF, source code, commit: deff9ab, available: https://github.com/mattconte/tlsf

[30] P. Kirienko, O1Heap, source code, commit: b21b069, available: https://github.com/pavel-kirienko/o1heap

[31] X. Sun, J. Wang, X. Chen, An Improvement of TLSF Algorithm, 15th IEEE NPSS Conference on Real Time, 2007, 5 p., ISBN: 9781424408665

[32] D. Lea, dlmalloc, source code, Version 2.8.6, Aug 2012, available: https://github.com/ARMmbed/dlmalloc/blob/master/source/dlmalloc.c

[33] C. D. Rosso, The method, the tools and rationales for assessing dynamic memory efficiency in embedded real-time systems in practice, IEEE, 2006 International Conference on Software Engineering Advances (ICSEA'06), Finland, Oct 2006, 7 p., ISBN: 0769527035

[34] A. Kurth, B. Forsberg, L. Benini, HEROv2: FULL-Stack Open-Source Research Platform for Heterogeneous Computing, Cornell University, Jan. 2022, 14 p., arXiv:2201.03861, available: https://arxiv.org/abs/2201.03861

APPENDIX A: HALF TREE PSEUDOCODE

```
/* Pseudocode for Half Tree allocator */
/* When the code updates fields, this is in reference to the fields required
by the tree structure. These can be fields in the block at hand, its parent,
left or right child, next block in its free list of same size or any block
that is going to be one of those things */

Rebin(block)
   Find bin index
   Set direction to first bit after first set bit
   Update non-empty bin bitmask
   If direction == 0
         If bin->left == NULL
               Place node as left child of bin and update fields
               Return
         Else
               Set node to bin->left
   Else
         If bin->right == NULL
               Place node as right child of bin and update fields
               Return
         Else
               Set node to bin->right
   Move direction by 1 bit
   While direction has more bits to examine
         If node->size == block->size
               Place block in the list in node->next and update fields
               Return
         If direction == 0
               If node->left == NULL
                     Place node as left child of bin and update fields
                     Return
               Else
                     Set node to node->left
         Else
               If bin->right == NULL
                     Place node as right child of bin and update fields
                     Return
               Else
                     Set node to node->right
         Move direction by 1 bit
```

```
Unbin(block)
   Find bin index
   Set direction to first bit after first set bit
   If direction == 0
         Set node to bin->left
   Else
         Set node to bin->right
   Move direction by 1 bit
   While direction has more bits to examine
         If node->size == block->size
               While node is not block
                     If node->next == block
                           Update fields to remove block
                           Return
                     Set node to node->next
               Break
         If direction == 0
               Set node to node->left
         Else
               Set node to node->right
   If node->next is not NULL
         Replace the node with node->next and update fields
   Else if node->left == NULL and node-> NULL // node is a leaf
         Remove the node from tree and update fields
   Else // The node is an internal node and only one of its size
         While leaf not found
               Move to left node if not NULL, if it is move to right
         Remove leaf from tree and update fields
         Replace node with leaf and update fields
   If bin is empty
         Update the non-empty bin bitmask
```

```
Alloc(size)
    Add overhead to size and round it up to minimum block size if needed
    Block = NULL
    If designated victim is not NULL
        If designated victim->size >= size
            Block = designated victim
            Designated victim = NULL
    Else
        Find the optimal bin for the size
        If optimal bin not empty
            Set direction to first bit after first set bit
            If bin->right is not NULL
                Block = bin->right
            Else if direction == 0 and bin->left is not NULL
                Block = bin->left
            Move direction by 1 bit
            If Block is not NULL
                While Block->size < size
                    // Prefer right here for performance
                    If Block->right is not NULL
                        Block = Block->right
                    Else if direction == 1
                        Block = NULL
                        Break
                    Else
                        If Block->left is not NULL
                            Block = Block->left
                        Else
                            Block = NULL
                            Break
                    Move direction by 1 bit
                If Block is not NULL
                    Unbin(Block)
        If Block == NULL
            Find next non-empty bin with larger blocks
            If found a bin
                // Prefer left here since its always smaller
                If bin->left is not NULL
                    Block = bin->left
                Else
                    Block = bin->right
                Unbin(Block)


    If Block is not NULL
        If Block->size – size > Minimum block size
            Split the block
            If leftover < designated victim max size
                If designated victim == NULL
                    Designated victim = leftover
                Else if designated victim->size < threshold
                    Rebin(designated victim)
                    Designated victim = leftover
                Else
                    Rebin(leftover)
    Return Block
```

```
Free(Block)
    // Coalesce the block with free neighbours
    If previous is free
        If previous is designated victim
            Designated victim = NULL
            Combine the blocks // Block now at previous block's header
        Else
            Unbin(previous)
            Combine the blocks
    If next is free
        If next is designated victim
            Designated victim = NULL
            Combine the blocks // Block now at Block's header
        Else
            Unbin(next)
            Combine the blocks
    If designated victim = NULL
        Designated victim = Block
    Else if designated victim->size < threshold
        Rebin(designated victim)
        Designated victim = Block
    Else
        Rebin(Block)
    Return
```

APPENDIX B: SYNTHETIC TRACE TEST RESULTS

Performance results in small trace. Units are in timer cycles.

	Average		**Worst**		**Average operation**	
	Alloc	**Free**	**Alloc**	**Free**	**Operation**	**Change**
Buddy	69	116	275	328	92,5	0,00%
First Fit	88	138	149	201	113	22,16%
Best Fit	147	136	217	208	141,5	52,97%
TLSF (Cont.)	239	186	261	292	212,5	129,73%
TLSF (Masm.)	154	148	187	245	151	63,24%
O1Heap	68	60	79	101	64	-30,81%
Half Fit CSTM	105	129	133	176	117	26,49%
Half Tree	88	170	194	288	129	39,46%

Performance results in large trace. Units are in timer cycles.

	Average		**Worst**		**Average operation**	
	Alloc	**Free**	**Alloc**	**Free**	**Operation**	**Change**
Buddy	74	99	133	159	86,5	0,00%
First Fit	77	128	103	189	102,5	18,50%
Best Fit	88	133	124	196	110,5%	27,75%
TLSF (Cont.)	260	200	261	292	230	165,90%
TLSF (Masm.)	185	172	187	245	178,5	106,36%
O1Heap	76	70	79	101	73	-15,61%
Half Fit CSTM	107	137	113	176	122	41,04%
Half Tree	102	151	165	260	126,5	46,24%

Performance results in random trace. Units are in timer cycles.

	Average		**Worst**		**Average operation**	
	Alloc	**Free**	**Alloc**	**Free**	**Operation**	**Change**
Buddy	80	124	310	370	102	0,00%
First Fit	86	136	120	200	111	8,82%
Best Fit	111	141	160	207	126	23,53%
TLSF (Cont.)	258	200	261	292	229	124,51%
TLSF (Masm.)	181	170	187	245	175,5	72,06%
O1Heap	75	69	79	101	72	-29,41%
Half Fit CSTM	108	137	120	176	122,5	20,10%
Half Tree	110	151	165	260	130,5	27,94%

Performance results in typical trace. Units are in timer cycles.

	Average		**Worst**		**Average operation**	
	Alloc	**Free**	**Alloc**	**Free**	**Operation**	**Change**
Buddy	78	125	239	290	101,5	0,00%
First Fit	85	136	121	200	110,5	8,87%
Best Fit	110	140	157	207	125	23,15%
TLSF (Cont.)	256	197	261	292	226,5	123,15%
TLSF (Masm.)	175	166	187	245	170,5	67,98%
O1Heap	73	66	79	101	69,5	-31,53%
Half Fit CSTM	107	135	120	176	120,5	18,72%
Half Tree	82	150	162	256	116	14,29%

Fragmentation results in small trace. Peak live memory for this trace is 5777 bytes. Units are in bytes.

Live memory 5777	**Large heap**			**Cost Metric sized heap**			
	Memory	**Frag. %**	**SBMM**	**Cost Metric**	**Memory**	**Frag. %**	**SBMM**
Buddy	8448	46,24%	12288	7488	7440	28,79%	48
First Fit	9494	64,34%	76554	7424	7408	28,23%	300
Best Fit	6932	19,99%	79112	6576	6556	13,48%	213
TLSF (Cont.)	6864	18,82%	78669	6976	6448	11,62%	193
TLSF (Masm.)	6864	18,82%	78408	7472	6688	15,77%	248
O1Heap	10080	74,49%	75808	9824	9664	67,28%	256
Half Fit CSTM	7024	21,59%	78960	6704	6624	14,66%	196
Half Tree	6924	19,85%	78992	7048	6900	19,44%	220

Fragmentation results in large trace. Peak live memory for this trace is 33923 bytes. Units are in bytes.

Live memory 33923	**Large heap**			**Cost Metric sized heap**			
	Memory	**Frag. %**	**SBMM**	**Cost Metric**	**Memory**	**Frag. %**	**SBMM**
Buddy	36864	8,67%	12288	36912	36864	8,67%	0
First Fit	41401	22,04%	44647	41424	41408	22,06%	5007
Best Fit	41401	22,04%	44643	41424	41404	22,05%	4964
TLSF (Cont.)	41376	21,97%	44157	42720	42048	23,95%	5481
TLSF (Masm.)	42528	25,37%	42744	43064	42280	24,64%	4608
O1Heap	81920	141,49%	24448	61600	61440	81,12%	8192
Half Fit CSTM	41412	22,08%	44572	41496	41416	22,09%	4968
Half Tree	41412	22,08%	44504	41560	41412	22,08%	4964

Fragmentation results in random trace. Peak live memory for this trace is 37061 bytes. Units are in bytes.

Live memory 37061	Large heap			Cost Metric sized heap			
	Memory	Frag. %	SBMM	Cost Metric	Memory	Frag. %	SBMM
Buddy	61440	65,78%	12288	55344	55296	49,20%	3072
First Fit	53067	43,19%	32981	44840	44824	20,95%	4604
Best Fit	43341	16,95%	42703	39680	39660	7,01%	1243
TLSF (Cont.)	46988	26,79%	38545	47328	46780	26,22%	4821
TLSF (Masm.)	44400	19,80%	40872	40600	39816	7,43%	616
O1Heap	85888	131,75%	22400	61600	61440	65,78%	2048
Half Fit CSTM	40756	9,97%	45228	41840	41760	12,68%	2424
Half Tree	45048	21,55%	40868	41912	41764	12,69%	2380

Fragmentation results in typical trace. Peak live memory for this trace is 4972 bytes. Units are in bytes.

Live memory 4972	Large heap			Cost Metric sized heap			
	Memory	Frag. %	SBMM	Cost Metric	Memory	Frag. %	SBMM
Buddy	12288	147.14%	12288	7728	7680	54,47%	192
First Fit	9177	84,57%	76871	7104	7088	42,56%	1215
Best Fit	6230	25,30%	79814	5336	5316	6,92%	115
TLSF (Cont.)	5948	19,63%	79585	5912	5384	8,29%	197
TLSF (Masm.)	6392	28,56%	78880	6232	5448	9,57%	128
O1Heap	9344	87,93%	76544	9504	9344	87,93%	1152
Half Fit CSTM	5944	19,55%	80040	5552	5472	10,06%	288
Half Tree	6276	26,23%	79640	5663	5516	10,94%	180