A Study on Dynamic Memory Allocation Mechanisms for Small Block Sizes in Real-Time Embedded Systems
Abstract

Embedded real-time and battery-powered systems are increasing in numbers, and their software complexity is growing. This creates a demand for more efficient dynamic memory allocation in real-time embedded systems. Small improvements in dynamic memory allocation can greatly reduce system overall memory usage, fragmentation and energy consumption. Most of today's general-purpose allocators are unsuitable for real-time embedded systems since they are not designed for real-time constraints.

This thesis contains a study on the suitability of dynamic memory allocation mechanisms for small block allocation in real-time embedded systems. We first perform a literature survey on dynamic memory allocation mechanisms and then analyze general-purpose allocators. From this we arrive to a set of allocation mechanisms for additional experimental study. We then conduct simulations on the selected mechanisms by using both real and synthetic traces to measure the mechanism fragmentation and WCET. We then evaluate the mechanisms and their tradeoffs and present an analysis on their suitability for small block allocation in real-time embedded systems.

This thesis additionally introduces Bitframe allocator, a new bitmapped fits allocator. The introduced allocator demonstrates that bitmapped fits can be used effectively for dynamic memory allocation. We are however unsure if bitmapped fits can offer better efficiency than other mechanisms.

Our results confirm that TLSF is one of the best allocators for real-time systems in terms of performance and fragmentation. Our results also confirm that reaps has low fragmentation and very low WCET when small blocks are allocated. Our results also show that simple segregated storage and region mechanism should not be used in real-time systems due to high worst-case fragmentation.

Keywords
thesis, information processing science, algorithms, performance, memory management, dynamic storage allocation, dynamic memory allocation, fragmentation, real-time systems, embedded systems
Foreword

I was working on a Nintendo DS game project during 2009. As usual, the deadline was looming close and we still had a bunch of critical issues to fix. One showstopper issue was a random crash which happened after some minutes of gameplay. Other major issues were long loading times and occasional frame skipping.

Debugging revealed that all of the issues were caused by DMA. Our Lua scripting backend allocated a whopping amount of tiny blocks, peaking roughly at 1500 allocations per frame\(^1\). The DMA just wasn't up to this task. A large number of tiny blocks were scattered around the heap, and prevented large blocks from being allocated. This classic case of fragmentation was the source of random crashing.

I was given the task to fix the issues in the allocator, and I chose to create a custom allocator for efficiently allocating small memory blocks for Lua. This custom memory allocator was the first version of Bitframe allocator presented in this thesis. The allocator worked better than expected and solved all the issues. Our Lua scripting backend performance improved considerably and random crashing disappeared.

This experience on DMA motivated me to study them in my thesis. I thank my colleagues at now-defunct Farmind game studio, including Jukka Jylänki. I also thank Symbio colleagues Jarkko Kemppainen, Jari Karppinen and Risto Huotari for their support and help. I additionally thank Seamus Hickey for his help on selecting this topic, and my thesis supervisor Ari Vesanen for his insight and help to get this job done.

Special thanks go to my lovely Lion for her endless support. This thesis would not be here without you Xiaojie.

Valtteri Heikkilä

Oulu, December 16, 2012

---

\(^1\) This is a large number of operations on the Nintendo DS since games typically show 60 frames per second and the main processor runs at 32 MHz.
Abbreviations

_DSA_ Dynamic storage allocator (or allocation)
_DMA_ Dynamic memory allocator (or allocation)
_MSU_ Most significant bit(s)
_SSS_ Simple segregated storage
_TLSF_ Two-level segregated fit
_WCET_ Worst-case execution time
1. Introduction

Dynamic storage allocation (DSA) has been a fundamental part of most computer systems since 1960, and since then part of operating systems research. Similar to the topics of searching and sorting, there exists a large amount of research on DSA, and the topic is widely considered to be either solved or unsolvable. (Wilson, Johnstone, Neely & Boles, 1995a, pp. 1, 4; Masmano, Ripoll, Crespo & Real, 2004, p. 1)

A dynamic memory allocator (DMA), often referred simply as *allocator*, is a DSA for memory allocation. The goal of DMA is to provide memory dynamically to the application at run time. It has been measured that up to 60% of application running time can be spent in DMA, and a central aim in research is to find efficient algorithms which both balance and minimize the time and storage costs of the DMA. (Masmano et al., 2004, p. 79; Hasan & Chang, 2005, pp. 35, 40; Berger, Zorn & McKinley, 2002, p. 4; Risco-Martin, Colmenar, Atienza & Hidalgo, 2011, p. 755)

Many classic allocator designs were conceived in 1960’s, including sequential fits, buddy systems, simple segregated storage and segregated free lists. Modern computing is however very different. Embedded systems are increasing in numbers, and their software complexity is growing, and many of the new embedded systems are battery-powered. This development creates a demand for more efficient DMAs, especially to reduce the increasing energy consumption. (Wilson et al., 1995a, pp. 47, 70; Risco-Martin et al., 2011, p. 756; Zorn, 2010, pp. 47-49) Even a small improvement in system DMA could have a large impact on energy consumption. When less processing and memory is used, clock speeds, memory sizes and overall chip area could be reduced.

Because of the vast number of active computing systems today, improvements in basic DMA algorithms could have major indirect effects on a global scale. Small improvements could lead to reduced burden to the environment, electricity production, cooling and manufacturing. For these reasons we believe that DMAs are an important topic for current and future computing systems research. (Zorn, 2010, p. 49; Grunwald, Zorn & Henderson, 1993, p. 179; Wilson et al., 1995a, p. 4)

Many modern embedded systems also need to operate under real-time constraints. There exists however considerably less research on real-time systems DMA. Most general-purpose DMAs are unsuitable for real-time systems because they may have unpredictable or long worst-case execution time (WCET) (Masmano et al., 2004, p. 79), and hence more research is needed to find more suitable DMAs.

Research additionally shows that a large majority of allocations by are from small sizes (Zorn and Grunwald, 1992, p. 4; Grunwald, Zorn & Henderson, 1993, p. 184; Wilson et al., 1995a, pp. 28, 36; Berger, Zorn & McKinley, 2002, p. 8; Hasan & Chang, 2005, pp. 45-46; Lee, Chang and Hasan, 2000, p. 391). There is however little research focusing specifically on small block size allocation. This is unfortunate, since small blocks are allocated extensively by dynamic and object oriented programming languages (Deflets, Dosser and Zorn, 1994, p. 530; Chang, Hasan & Lee, 2000, p. 7; Risco-Martin et al., 2011, p. 755; Lee, Chang & Hasan, 2000, pp. 387, 391). These languages are becoming
more and more prevalent, and their performance largely depends on the performance of DMA.

Research has also shown that the implementation of allocator mechanism is the main source of wasted memory in DMA (Wilson et al., 1995a; Johnstone & Wilson, 1998, pp. 26, 32, 35-36; Masmano et al., 2008a, p. 156). For this reason, we have focused primarily on allocator mechanisms which are the building blocks behind all DMA. With this work, we want to provide new knowledge on the topics of real-time embedded systems DMA, small block allocation, and to the research on DMA mechanisms. We have noticed these specific topics have less coverage in the DMA research.

1.1 Research topic

The research problem in this thesis is defined by the following question:

"Which DMA mechanisms are suitable for small block allocation in real-time embedded systems, and what are the tradeoffs between the mechanisms?"

The research problem can be divided to the following four research questions:

1. "Which DMA mechanisms are suitable for real-time embedded systems?"
2. "Which of these mechanisms are suitable for small block allocation?"
3. "What differences do the mechanisms have in terms of execution time and wasted memory (fragmentation and other overhead)?"
4. "What comparative tradeoffs do the mechanisms have?"

The first two questions focus on small block allocation mechanisms and their suitability to real-time embedded systems. We will answer these questions with a DMA mechanism literature survey and by analysis of existing allocator implementations and their source code. Questions 3. and 4. address allocator performance and efficiency, and these questions are answered by quantitative means. We will perform simulations and measurements on implementations of allocator mechanisms and present analysis and evaluation of their efficiency. Measurements are done with both real and synthetic allocation traces.

We will additionally introduce a new bitmapped fits allocator designed for small block allocation in real-time embedded systems. We will evaluate and compare the allocator along with the other mechanisms in the simulation and evaluation part.

Most of this thesis is centered on DMA mechanisms. We are not specifically interested on DMA policy or strategy or general-purpose allocator implementations. We will contribute on the knowledge of various DMA mechanisms, their efficiency and characteristics, and their suitability for real-time systems and small block allocation.

2 In this study, we define chunks as small areas of memory. Blocks are chunks which are associated or referenced by either with the DMA or application.
1.2 Limitations and assumptions

This thesis limits its focus and allocation experimentation to small blocks. We will define small block having size of 1..512 bytes. Studied allocators are only required to allocate small blocks. Allocating other block sizes is optional. Our simulations also allocate only small block sizes, and only small block sizes are included in analysis and evaluation.

This thesis is limited to allocators that are written in C and behave similarly to C standard functions malloc and free. Other C standard functions, calloc and realloc are not studied because they can be implemented with malloc or free functions, and additionally because they are very rarely used (Chang, Hasan & Lee, 2000, p. 11). The C++ new and delete essentially use malloc and free (Hasan & Chang, 2005, p. 36). Allocation operation takes the size of the allocation (in bytes) as its only parameter, and returns a single memory address to allocated block. A zero return value means that the allocation failed. Free operation takes the memory address of a block as its only parameter, and does not return a value. Additionally, block addresses are aligned to 8 bytes. This is a common block alignment in 32-bit systems for C language (Evans, 2000, p. 5). This generally means that possible size classes in the allocator are 8 bytes apart (8, 16, 24 ...).

We assume memory management unit (MMU), paging and virtual memory are unavailable in the system. This is because real-time embedded systems often do not have a MMU (Masmano et al., 2004, pp. 80, 82; Puaut, 2002, p. 42). All pointers in this thesis are thus in linear (or physical) address space. We also assume 32-bit system and address space.

We also require that allocators can perform without the aid of operating system (OS) (for example functions sbrk and mmap in unix). The allocators are initialized to use a specified address space belonging to the allocator, and the allocator has full privileges to this address space. We also assume that heap size cannot grow, and expect heap size to be small (Masmano et al., 2004, p. 82).

This thesis assumes that the allocators do not know how the allocated memory blocks are used, what information is stored in the blocks, or the lifetime (duration between allocation and deallocation) of the blocks. None of the allocators examine the block contents. Allocator neither relocate blocks to compact memory.

We do not study locality or cache-related performance implications. We however acknowledge that allocator locality characteristics may have as high as 25% effect on application overall (average) performance (Grunwald, Zorn & Henderson, 1993, p. 177). Furthermore, we are aware that allocators with good locality characteristics have generally also low fragmentation (Johnstone & Wilson, 1998, p. 32).

We do not approach the question of information security in dynamic memory management. We assume a trusted environment where no malicious code can be executed (Masmano et al., 2004, p. 82). We however understand the importance of information security (Zorn, 2010, pp. 14-21).

We do not discuss the topic of custom dynamic memory allocators. However, in the light of current research, we believe that custom allocators may be more efficient than

---

3 Some use “deallocate” as a synonym for “free”, but we will use “free”.
general-purpose allocators, but more research is needed to show this (Risco-Martin et al., 2011, pp. 755-756; Masmano et al., 2008a, p. 153).

Topics such as garbage collection, reference counting, automatic and implicit memory management, are not directly discussed in this thesis. The topics are however closely related. Dynamic memory allocation algorithms are fundamental to programming environments with automatic or implicit memory management, and we believe, the algorithms play a key role in the runtime performance on these environments. We also believe that the popularity of programming languages, such as JavaScript and Python, shows the topic dynamic memory allocation is increasingly important.

1.3 Thesis structure

Chapter 2 will present key concepts and background for later chapters. The chapter will also describe the real-time requirements for embedded systems DMA. Chapter 3 will present a literature survey on DMA mechanisms and additional analysis, and addresses the first research question. The chapter begins with simple mechanisms and gradually progresses to more complex ones. The chapter provides analysis and insight on the suitability of the allocation mechanism for real-time embedded DMA.

Chapter 4 contains analysis on small block allocation mechanisms in existing allocators to address the second research question. The analysis is mainly based on literature, but source code analysis is also used. Chapter 5 presents Bitframe allocator, a new bitmapped fits allocator. The allocator is included in our simulation experiments.

Chapter 6 describes how simulation and measurements were performed on allocator implementations, presents the measurement results, and shows evaluation and analysis based on the results. This chapter addresses the third and fourth research questions, and presents some answers to the research problem. Chapter 7 will present our conclusions based on our previous work. We will present our research results and some future research topics.
2. Background

This chapter introduces memory management key concepts, such as dynamic memory allocation, fragmentation, and the allocator strategy-policy-mechanism model. We also discuss the special constraints in real-time embedded systems, which affect DMA design and implementation. We will refer to the introduced concepts throughout the thesis.

For the most part of this chapter, we rely on a survey by Wilson, Johnstone, Boles and Neely (1995a). This thorough survey contains an extensive review of past dynamic memory allocation literature. It also presents models and categorizations which are used in this research. We are not aware of later works in the field with this magnitude.

2.1 Static and stack-dynamic memory allocation

Static and stack-dynamic memory allocation are discussed briefly for completeness. In static memory allocation, program reserves the necessary amount of memory at start-up, and this amount is then fixed until the program terminates. Static memory allocation requires that program memory usage is bounded and known at compilation time. This requirement makes static memory allocation unsuitable for many algorithms and programs.

In stack-dynamic allocation, a memory area is reserved for a program to serve as a stack. This memory area can have either fixed size or it may grow depending on the system. The stack maintains a pointer to the top of the stack memory, and this pointer is usually decremented on allocation and incremented on free. This mechanism imposes at least two constraints: free operations must come in the reverse order as allocations (in a LIFO manner), and the memory usage must be bounded and well-defined. Violating to constraints will cause stack memory to corrupt or overflow.

Static and stack-dynamic memory allocation imposes rigid constraints on how programs may use memory. Many algorithms cannot be implemented alone with these methods. An algorithm, that needs to free memory in a different order that it was allocated, is not feasible with neither of the methods.

2.2 Dynamic memory allocation

Dynamic memory allocator is an algorithm from which an application can request memory chunks and free them at any time and any order. DMA has the goal to provide dynamically to the application the amount of memory it requires at runtime. The memory managed by DMA can be in two distinct states: allocated (“live”, potentially in use by the application) or free (available for allocation). A DMA must have explicit operations to allocate and free memory. (Masmano et al., 2004, pp. 79, 83; Wilson et al., 1995a, p. 1; Berger, McKinley, Blumofe & Wilson, 2000, p. 120)
The memory of DMA is maintained in a memory area called heap\(^4\), which defines a storage where memory chunks of various sizes can be allocated. Most modern OSes map the heap memory as shown in figure 1. The heap usually grows upward while stack grows downward. The brk pointer records where the current heap ends. This mapping generally requires virtual memory support. When this support is not available, heap might not be able to grow.

![Figure 1. System memory map. (Hasan & Chang, 2005, p. 36)](image)

 Allocators use bookkeeping data structures to maintain information of allocated blocks. These may be linear lists, totally or partially ordered trees, bitmaps, or any suitable data structures. The performance of the DMA is mainly determined by its data structures. Since different bookkeeping data structures have different tradeoffs in time and storage costs, it is acknowledged that there exist no “optimal” DMA. For every DMA there is a use case that results in suboptimal performance or memory use. (Wilson et al., 1995a, p. 5; Hasan & Chang, 2005, p. 47; Masmano et al., 2004, p. 81; Chang, Hasan & Lee, 2000, p. 8; Risco-Martin et al., 2011, p. 756)

Efficient DMA design requires careful balancing of time and space costs. The main design goal in DMA is however to minimize space costs, and this is often more difficult than designing for low execution time. More research is generally needed to find new DMA algorithms with lower storage costs. (Wilson et al., 1995a, p. 5; Hasan & Chang, 2005, p. 40; Chang, Hasan & Lee, 2000, p. 8; Puaut, 2002, p. 49; Grunwald, Zorn & Henderson, 1993, p. 181)

### 2.3 Allocator strategy, policy and mechanism

Wilson and others (1995a) introduced a three-level model for allocator analysis. The levels in this model are allocator strategy, policy and mechanism. The model defines an allocator as follows: an allocator is a mechanism implementing a placement policy which is motivated by strategy to minimize fragmentation. The model assumes that the three levels interact during the allocator design process and can be determined from an implementation. The distinction between strategy and policy is however not clear-cut, and a strategy could be viewed as a policy, or vice versa, at a different level of

---

4 Not to be confused with heap data structure.
abstraction. (Wilson et al., 1995a, pp. 6-7) We refer to this model frequently throughout the thesis.

Allocation strategy takes into account the regularities in the program behavior. It determines a range of acceptable placement policies which define where to allocate requested blocks. The strategy attempts to minimize fragmentation by selecting suitable policies depending on the heap state. (Wilson et al., 1995a, pp. 6-7) For example, a best fit policy always selects the block that most closely matches the allocation size.

Mechanism is a set of algorithms and data structures that implement the policy. The mechanism is chosen to implement the policy efficiently in terms of time and space complexity or overheads. (Wilson et al., 1995a, p. 7) As an example, a best fit policy can be implemented by searching a linked list of available blocks to locate the closest matching block.

2.4 Fragmentation and wasted memory

Fragmentation is generally defined as the inability to reuse memory that is free, and it is traditionally divided into internal and external fragmentation. Fragmentation is one source of wasted memory in the allocator, while the other source is the overhead from the allocator implementation. (Wilson et al., 1995a, pp. 8, 14; Masmano et al., 2008a, pp. 156-157)

*Internal fragmentation* occurs when a block is allocated to hold an object, but the block is larger than the allocation, and the remainder is wasted. Internal fragmentation is defined as wasted memory inside an allocated block. (Wilson et al., 1995a, pp. 8-9; Peterson & Norman, 1977, p. 424; Masmano et al., 2008a, p. 156; Ogasawara, 1995, p. 22)

*External fragmentation* occurs when free blocks of memory are available for allocation but are too small (or otherwise unable) to hold future allocations. This situation is caused by “holes” in the heap coming from isolated free blocks. (Wilson et al., 1995a, pp. 8-9; Masmano et al., 2008a, p. 156; Ogasawara, 1995, p. 22)

*Implementation overhead* is wasted memory in the internal data structures of the allocator implementation and bookkeeping. For example, allocated and free blocks usually contain block headers with information on the block (Wilson et al., 1995a, p. 30; Johnstone & Wilson, 1998, p. 27; Masmano et al., 2008a, p. 156).

It has been shown that fragmentation is not a serious issue for a large majority of programs and that wasted memory is mainly caused by implementation overhead from allocation mechanisms – not by fragmentation or allocation policy. There are good allocation policies which have been shown to be efficient, and there are good mechanisms to implement them. (Wilson et al., 1995a; Johnstone & Wilson, 1998, pp. 26, 32, 35-36; Masmano et al., 2008a, p. 156)

2.4.1 Quantifying fragmentation

Fragmentation is difficult to quantify. This is mainly because the DMA and heap state can be organized in almost countless ways. Fragmentation depends at least on the current number and sizes of non-reusable memory holes in the heap, on the future
behavior of the application and of the DMA, on the past and present allocated block sizes, their distribution, their quantity, and the order in which allocations are made. There are many methods for measuring fragmentation in DMA, and there is no single correct way to measure fragmentation. (Wilson et al., 1995a, pp. 14-15; Puaut, 2002, p. 49; Peterson & Norman, 1977, pp. 424, 426; Masmano et al., 2008a, p. 156; Johnstone & Wilson, 1998, p. 32) Researchers also use different methods to quantify fragmentation in the experiments, and this makes the comparison of results difficult.

Due to the complexity of fragmentation problem domain, research cannot generally rely on solely analytical methods to quantify fragmentation (Peterson & Norman, 1977, p. 426; Puaut, 2002, p. 49). The methodology relies instead on simulation and measurements. DMA experimentation often involves construction of a working DMA implementation, or its model, and simulation with traces of allocation and free requests. The traces are created by two sources: real programs and synthesis. Traces from real programs correspond better to real-world use than synthetic traces. Traces can also be synthesized with different methods of which probabilistic methods are most common. Probabilistic methods are also combined with more complex payload models. Synthetic traces reveal information on the worst-case behavior of a DMA, whereas traces from real programs show DMA behavior under real-world use. Synthetic traces are preferred when evaluating real-time DMA because worst-case behavior needs to be understood. (Masmano et al., 2004, p. 86; Masmano et al., 2008a, p. 162)

Johnstone and Wilson (1998, p. 32) summarizes four methods for calculating fragmentation. Two of the methods are essentially same as the ones presented in (Peterson & Norman, 1977, p. 424) and (Deflets, Dosser & Zorn, 1994, p. 535). Our research will use the following method from (Masmano et al., 2008a, pp. 157, 163; Masmano et al., 2006, p. 72).

\[
F = \frac{H - M}{M}
\]

Here \( F \) is the fragmentation, \( H \) is the maximum memory used by the DMA, and \( M \) is the maximum allocated live memory used by the trace during simulation. The fragmentation calculation is illustrated by figure 2. In the figure, point 1 corresponds to the location of maximum memory used by the DMA (\( H \)), and point 2 corresponds to the location of maximum allocated live memory (\( M \)).
Special requirements from real-time embedded systems

Real-time embedded systems are embedded systems which operate within real-time constraints. They are usually dedicated to perform a specific task under a low response-time constraint. The task of the real-time embedded system defines its capabilities, but often it will have only little memory (Masmano et al., 2004, p. 80). The increasing complexity in the systems has created a demand for DMA algorithms which are more suitable for real-time embedded use.

Most DMAs are unsuitable for real-time systems since they are designed for low average execution time and not for low WCET. Real-time systems need to ensure fast response time, and for this it is necessary to determine the WCET of all running code in the system. The WCET needs to be low enough to meet the requirements in the system response time. DMA algorithms with O(1) time complexity are generally considered to be most suitable for real-time systems. (Masmano et al., 2008a, p. 175; Masmano et al., 2004, p. 79; Nilsen & Gao, 1995, p. 151; Ogasawara, 1995, p. 22)

Many real-time systems are not allowed to exhibit unreliable behavior under any circumstance, and they may execute for weeks, months and even years. This makes fragmentation a relevant issue for real-time DMA. A badly designed DMA will accumulate fragmentation over time, which may lead to unpredictable system behavior and response times. Systems developers often fear that DMA is too unreliable for real-time systems, and generally try to avoid it whenever possible. (Masmano et al., 2008a, p. 152; Masmano et al., 2004, pp. 79-81; Puaut, 2002, pp. 41, 46; Ogasawara, 1995, p. 21; Nilsen & Gao, 1995, p. 151)

The special nature of real-time systems imposes strict requirements for DMA. A good summary of these requirements are presented in (Masmano et al., 2008a, pp. 152, 175-
16; Masmano et al., 2006, p. 69; Masmano et al., 2004, p. 80). They define requirements for real-time DMA as the following:

- Bounded execution time. The WCET of DMA operations must be bounded and known. This requirement is mandatory.

- Fast completion time. The WCET of DMA operations must be short. This requirement is not mandatory.

- Minimize memory pool size. Allocation operations need to be satisfied without exception. The DMA must have low worst-case fragmentation to prevent the exhaustion of system memory. This requirement is mandatory.

Both requirements of bounded WCET and low fragmentation are considered mandatory, while the requirement for low WCET is only preferred. A low WCET is naturally more desirable, and it defines the usability of the DMA in real systems.

2.6 Related work

While DMA research is an old and broad topic spanning roughly half a century of research, real-time systems DMA is a fairly new research topic. To our knowledge, the earliest research focusing explicitly on real-time DMA originates during the 1990's. We will summarize some of the relevant research on the field.

Nilsen and Gao (1995) performed measurements on several general-purpose C and C++ allocators to determine their suitability for real-time use. Some of the measured allocators are well-established allocators, such as dlmalloc and the SunOS allocator. Relying on their measurements, they conclude that allocators utilizing traditional methods are unusable in real-time systems. (Nilsen & Gao, 1995, pp. 143, 151)

Ogasawara (1995) introduces a Half fit allocator with O(1) time complexity. The allocator is shown to have bounded WCET, and also lower fragmentation than binary buddies under synthetic trace experiments. The study emphasizes the suitability of O(1) time algorithms for real-time system DMA. (Ogasawara, 1995, pp. 21, 24) The Half fit allocator is described in section 4.2.2.

Puaut (2002) presents measurements and analysis on timing behavior of different real-time DMA using both real and synthetic payloads. Worst-case behavior is obtained using synthetic payloads. The study shows that WCET obtained analytically is a pessimistic and context-independent estimate, while WCET obtained empirically is context-sensitive and less pessimistic. The study also notes that allocators with a low average execution time may not have a low WCET. (Puaut, 2002, pp. 45, 47-49)

Masmano, Ripoll, Crespo and Real (2004) introduces TLSF, a real-time system DMA with O(1) time complexity, bounded WCET and low worst-case fragmentation. They also perform a brief evaluation of various DMA and TLSF by various synthetic worst-case workloads. They conclude that TLSF has excellent WCET and fragmentation characteristics. (Masmano et al., 2004, pp. 79, 86-88) The TLSF allocator is further discussed in section 4.2.6.

Masmano, Ripoll and Crespo (2006) continued experimentation from their previous study from 2004. The new study compares TLSF with other allocators: first fit, best bit,
binary buddies, dlmalloc, and Half fit. They design a custom model to synthesize workload for the allocator experiments. Based on these experiments, authors conclude that first fit, best fit, dlmalloc, and surprisingly also, binary buddies are not suitable for real-time systems. (Masmano et al., 2006, pp. 68-69, 70-71, 75)

A later study by Masmano, Ripoll, Balbastre and Crespo (2008a) repeats their previous experiments from earlier study in 2006, and includes the following DMA: first fit, best fit, AVL tree, binary buddies, dlmalloc, Half fit and TLSF. This time both real and synthetic workloads were used to cover real and worst-case scenarios. First fit, best fit and dlmalloc are not recommended for real-time use due high WCET. TSLF and Half fit are evaluated as best for real-time use. AVL tree is shown to have higher WCET than binary buddies. Half fit and binary buddies are shown to have high but acceptable worst-case fragmentation. (Masmano et al., 2008a, pp. 161-162, 164, 168-169, 173)
3. Allocation Mechanisms

This chapter will discuss categories of allocation mechanisms in three main sections. The first and second section will discuss low-level and basic allocation mechanisms, while the third section will discuss other allocation mechanisms. In this chapter follows the categorization in (Wilson et al., 1995a). We make some additions mainly in the low-level and other mechanisms sections to present and analyze some mechanisms relevant to real-time DMA.

3.1 Low-level mechanisms

This section describes fundamental low-level mechanisms commonly used in allocators. These mechanisms provide the basis for implementing different higher level mechanisms and allocation policies. (Wilson et al., 1995a, p. 27) Some of these mechanisms are generally known data structures that are used in many algorithms, not exclusively by allocators.

3.1.1 Free lists and link fields

Free lists keep track of free blocks of memory in the allocator (Wilson et al., 1995a, p. 28; Hasan & Chang, 2005, p. 36). When a block is allocated, it is removed from the free list and returned to the client. Client is then responsible for maintaining the allocated block, and also freeing it later. The free lists are usually searched on allocation request to find a suitable block to satisfy an allocation. This search involves a policy to select a suitable block which is often called a fit. (Knuth, 1973, p. 436-437) Different fits are discussed in section 3.2.1. Figure 3 illustrates the free list data structure.

![Figure 3. Illustration of sequential free list. Sequences of white rectangles represent free blocks. Arrows represent links between free blocks. (Hasan & Chang, 2005, p. 37)](image)

In order to form a free list, free blocks need to contain link fields (Wilson et al., 1995a, p. 28). Both doubly and regular linked lists are often used in free lists depending on the requirements, and the link fields are essentially linked list nodes. Allocated blocks often store a minimal set of link fields to reduce overhead per block (Wilson et al., 1995a, p. 28; Knuth, 1973, p. 436).
3.1.2 Block headers

A block header is a bookkeeping data structure stored at the head of a memory block. The allocator can rapidly access the block header from a client block address, for example, by subtracting header size from the address. Allocated and free blocks have often different types of headers. Allocated block header should be minimized to reduce per-block allocation overhead. Free block headers however do not introduce overhead, since the whole block is free anyways. Block headers generally consist of the following: block header fields, link fields, boundary tags, and possible padding bytes from block alignment. We will discuss these next, except for link fields which were discussed in the previous section.

Header fields store information relevant to the memory blocks, for example block size and allocated/free status. Boundary tags are used to mark starts and ends of blocks to track write overflows and free list corruption. Modern implementations of boundary tags often omit the end tag. Block alignment constraints the address and size of the memory blocks. The alignment is commonly one or more machine words, and it may be required by the system or hardware. The alignment can also be used to save bits from block header address fields\(^5\). (Wilson et al., 1995a, pp. 27-28) Block alignment also introduces padding bytes in block headers.

3.1.3 Coalescing and splitting

Block coalescing refers to process of combining consequent free blocks into a larger one. In the process, a free block's neighbors are checked, and if a neighbor is also free, it is joined with the original block. Coalescing reduces fragmentation and increases memory reuse since large coalesced blocks can store bigger future allocations. Coalescing is often performed when a block is freed, but it can also be deferred for later (see next section). (Wilson et al., 1995a, p. 9; Hasan & Chang, 2005, p. 37) Coalescing is illustrated in figure 4.

Figure 4. Illustration of coalescing. Blocks with length 8 and 4 can be coalesced whereas block with length 2 cannot. (Hasan & Chang, 2005, p. 37)

Block splitting is a reverse process to coalescing. It involves dividing a free block into two smaller blocks (Wilson et al., 1995a, p. 9). The splitting is usually performed when no suitably small block is found to satisfy an allocation operation. In this situation, a large free block is selected and split in two parts, where the other part is used to satisfy the allocation, and the other is put to a free list for later use.

3.1.4 Deferred coalescing

Deferred coalescing (or deferred free) is a mechanism where freed blocks are not immediately coalesced, but instead their coalescing is deferred to some point in the future. This is motivated by the observation, that majority of allocated block sizes are

---

\(^5\) For example, a block alignment of 4 bytes frees 2 bits from link fields, since addresses are always aligned by 4. The freed bits can be used as flags by the allocator, for example to store the allocated/free state of a block.
exactly the same as previously freed ones, and thus repeated coalescing and splitting wastes time in the allocator. Deferred coalescing improves allocator performance, but increases fragmentation. (Hasan & Chang, 2005, pp. 37, 40, 47; Wilson et al., 1995a, pp. 15, 18, 22; Johnstone & Wilson, 1998, p. 35; Masmano et al., 2004, p. 82)

To implement deferred coalescing, an allocator places freed blocks in special deferred free lists. Blocks from the lists are reused directly by the allocator. (Wilson et al., 1995a, pp. 22-23) Coalescing may be performed later when a heuristic triggers it, for example, after some number of allocation or free operations. This occasional coalescing however produces unpredictable timing behavior and affects the WCET of the DMA. For this reason deferred coalescing should not be used in real-time applications (Masmano et al., 2004, p. 82).

### 3.1.5 Lookup tables

Lookup tables store precomputed or generated data elements for later access by an algorithm. Lookup table data is accessed directly by a table index or by other mapping. Lookup tables reduce complex algorithms to array indexing operations with constant time complexity. They are sometimes also in DMA. (Wilson et al., 1995a, p. 29) For example, TLSF and Bitframe allocator utilizes lookup tables (see chapter 5.1) (Masmano et al., 2004, p. 83).

### 3.1.6 Bitmaps

A bitmap (or a bit table) is a vector of bits where each bit maps to a single data element. The mapping is usually linear so that a bit index directly corresponds to an index of an element in an array of elements. Many allocators use bitmaps to mark blocks or free lists since bitmaps are size-efficient and fast to manipulate.

Allocators often perform scans for one (or zero) bit in a bitmap starting from a specified index. The scan is mostly performed with bit-scan instructions found in majority of processors. The instructions are efficient for accelerating bitmap scanning, and mostly execute in constant-time. (Ogasawara, 1995, p. 23; Masmano et al., 2006, p. 69) Such instruction are for example CLZ (Count Leading Zeros) (ARM, 2010, Chapter 4, p. 54), BSF (Bit Scan Forward) and BSR (Bit Scan Reverse) (Intel, 2011, Chapter 3, pp. 92-97).

Bitmaps are also sometimes searched for sequences of zeros (or ones) of a desired length. A basic search implementation is a bit-by-bit scan, which has time complexity $O(N)$, where $N$ is the size of bitmap. Improved search algorithms however exist, which use lookup tables and bit manipulation techniques. For example, a 256-way lookup can store bit sequence lengths in an 8-bit sequence, and another one can store the free run lengths across byte boundary. (Wilson et al., 1995a, p. 42)

### 3.1.7 Pointer bumping

Pointer bumping is an allocation mechanism over a chunk of memory. Upon allocation, a pointer is “bumped” – incremented or decremented – towards the beginning or the end of the chunk, starting from the opposite end. When pointer reaches the opposite end, the whole chunk has been traversed and no more allocations can be made in the chunk.
Pointer bumping is a common technique used at least by region and reap mechanisms. (Berger, Zorn & McKinley, 2002, p. 7)

3.1.8 Special treatment of small blocks

Because programs usually allocate and free a large number of small blocks, allocators often treat small blocks differently from others. A common method is to choose a suitable size limit and use a specific mechanism to manage blocks under that size. Some allocators use more than two size ranges, for example, small, medium and large blocks, where each range uses a different mechanism. This thesis focuses on efficient small block allocation mechanisms, and many of the analyzed and simulated mechanisms can be useful for special treatment of small blocks in allocators.

Special treatment of small blocks may reduce time and space costs on the average, but not necessarily on the worst-case. Additionally, fragmentation and WCET analysis may become more complicated when more than one allocation mechanism in used in an allocator. This reduces the usefulness of this method in real-time systems, and we believe this mechanism should not be used as a part of real-time DMA.

3.2 Basic allocator mechanisms

This section will introduce a classification of allocators as defined by Wilson and others (1995a, pp. 26-46). The classification is made primarily by underlying allocation mechanism. Five categories are defined: sequential fits, segregated free lists, buddy systems, indexed fits and bitmapped fits (Wilson et al., 1995a, p. 30; Masmano et al., 2004, p. 81). While the following sections will mainly introduce mechanisms, fundamental policies are also discussed along the way. After the introduction sections on the mechanisms, we will lastly analyze their suitability for real-time DMA.

3.2.1 Sequential fits

Several classic allocators use the sequential fits mechanism. All sequential fits mechanisms use a single free list containing free blocks. Allocated blocks contain headers and free blocks additionally contain boundary tags. (Wilson et al., 1995a, p. 30; Knuth, 1973, pp. 440-442) The free list is searched with some algorithm which also defines the variant of the sequential fit. There are three common variants: best fit, first fit, next fit and good fit. Other more uncommon variants are worst fit and optimal fit (Wilson et al., 1995a, pp. 32-33), but we will not discuss these variants because they are irrelevant to our topic.

Best fit

Sequential best fit allocator searches the full sequential free list, and returns the smallest available free block large enough to satisfy the allocation. Search is exhaustive, but may stop when a perfect fit is found. Sequential best fit allocator implements naturally a best fit policy; it always finds the best block to store the allocation. This policy is considered to be the best policy to minimize fragmentation. It minimizes wasted space after block split, and if a split is not made, it minimizes wasted space inside the block. Best fit policy can be implemented more efficiently at least with indexed or segregated fits. (Johnstone & Wilson, 1998, pp. 27, 33; Masmano et al., 2004, p. 83; Robson, 1977, pp. 243-244; Hasan & Chang, 2005, p. 40; Wilson et al., 1995a, p. 30; Knuth, 1973, p. 437)
**First fit**

First fit searches the sequential free list from the beginning, and uses the first block large enough to satisfy the allocation request. The block can be split if it is larger than necessary, and the remainder is put to the free list. The motivation for first fit is to reduce average execution time of the allocator in comparison to best fit. A variant of first fit is the address-ordered first fit. In address-ordered first fit the free blocks are inserted to free list in address-order. The insertion requires a search both when block is allocated and freed. The address-ordered first fit has low fragmentation similar to best fit, and it can be implemented efficiently using a Cartesian tree. (Wilson et al., 1995a, pp. 30-31; Hasan & Chang, 2005, p. 36; Johnstone & Wilson, 1998, pp. 33, 37; Wilson et al., 1995b, p. 34)

**Next fit**

Next fit is a variation of first fit: a pointer records the position in free list where last search was satisfied, and the next search continues from that position. The rationale behind this is to decrease the average search time. It has been however shown that next fit actually increases average search time compared to first fit. Next fit also suffers from worse fragmentation and locality than first fit and best fit. (Wilson et al., 1995a, p. 31; 1995b, p. 27; Bays, 1977, pp. 191-192)

**Good fit**

Good fit is an “almost best fit” policy (Masmano et al., 2004, p. 83), and it is not strictly a sequential fits mechanism. Good fit policy is common in segregated free list allocators where best fit search is often omitted, and a block with estimated best fit is used instead. Good fit policy has been shown to produce low fragmentation similar to best fit (Wilson et al., 1995a, p. 9; Masmano et al., 2008a, p. 156).

### 3.2.2 Segregated free lists

Probably originating from paper by Comfort (1964), segregated free lists mechanism involves an array of free lists where each list holds blocks from a specified size range. Upon allocation, a block is removed from a free list matching the allocation size, and upon free, the freed block is similarly added to a free list of the matching size. Block coalescing may not be performed immediately but may be deferred for later time. There exists several segregated free list mechanisms which are classified in two main categories: simple segregated storage and segregated fits. (Wilson et al., 1995a, p. 36)

**Simple segregated storage**

Simple segregated storage (SSS) allocator uses an array of free lists, where each list serves blocks of a certain size class. The size classes are constant, and no coalescing or splitting is performed by the allocator. Upon allocation, blocks are taken directly from a suitable free list, and if a free list becomes empty, more blocks are allocated from the system. (Wilson et al., 1995a, p. 36)

**Segregated fits**

Segregated fits mechanism uses multiple free lists to hold blocks of a size class or size range. It performs coalescing and splitting, and may use deferred coalescing. Upon allocation, a segregated fits allocator chooses a suitable free list matching requested size, and then usually sequentially searches a lists for a suitable block. If there are no suitable blocks in the list, the next list with a larger size class will be used, and so on, until a free block is found. The mechanism has usually good fit or best fit policy.
Wilson and others (1995a, p. 37) define three subcategories for segregated fits allocators:

**Exact lists** category allocators use a free list for each possible block size, which can be many. Accelerating data structures, such as binary trees, may be necessary to reduce cost of finding a suitable free list for allocation. (Wilson et al., 1995a, p. 37) In practice, programming systems and hardware forces block sizes to be multiples of some number of bytes, and thus the sizes are not “exact”.

**Strict size classes with rounding** category allocators maintain a number of segregated free lists where each list holds only blocks of one size class, and allocation sizes will be rounded up to next matching size class (Wilson et al., 1995a, p. 37). Because every free list contains blocks of one size class only, sequential search is not needed, and the allocation executes in constant-time.

**Size classes with range lists** category allocators allow the free lists to contain blocks in a specified larger size range. The allocator performs a sequential search on a matching free list in order to satisfy an allocation request. This category allocator was first introduced in a paper by Purdom, Stigler and Cheam (1971). (Wilson et al., 1995a, pp. 36-38)

### 3.2.3 Buddy systems

Buddy systems are significantly different from other mechanisms. In buddy systems, block placement and size are constrained by a simple mathematical function, and the memory is hierarchically split to two or three parts to form a data structure similar to binary or tertiary tree. Neighboring blocks that belong to a same level in hierarchy are called *buddies*. Only buddies can be coalesced, and only if they are free. Main advantage in the mechanisms is that buddies can be found quickly by deriving from the mathematical function used to constrain the blocks. (Wilson et al., 1995a, p. 38; Hasan & Chang, 2005, p. 37) Figure 5 illustrates a buddy system hierarchy where blocks are divided to two smaller parts (binary subdivision).

![Figure 5. Example of a buddy system. (Knuth, 1973, p. 449)](image)

There are four well known buddy system variants: binary buddies, Fibonacci buddies, weighted buddies, and double buddies. We will introduce these next. Other buddy

---

6 For C language, sizeof(double) is commonly used as alignment for blocks returned by malloc(). A common alignment on 32-bit systems is 8 bytes (Feng & Berger, 2005, p. 70), and 64-bit systems generally use 16 bytes.
systems exist, such as tertiary buddies introduced by Yadav and Sharma (2010), but we do not discuss them further, since they do not seem to offer relevant benefits (Yadav & Sharma, 2010, p. 66).

**Binary buddies**


**Fibonacci buddies**

Fibonacci buddies use the Fibonacci series as the source for buddy sizes. The algorithm was first introduced by Hirschberg (1973), and possibly originated from an exercise in (Knuth, 1973). Fibonacci buddies mechanism splits blocks to two unequal sizes of the Fibonacci series \( L_i = L_{i-1} + L_{i-2} \). The block sizes in Fibonacci buddies is more closely-spaced than in binary buddies, which reduces internal fragmentation compared to binary buddies. Cranston and Thomas (1975) introduced a method for rapid buddy address calculation that is comparable or slightly slower than binary buddy address calculation. (Yadav & Sharma, 2010, pp. 63, 66; Wilson et al., 1995a, pp. 39, 49; Peterson & Norman, 1977, p. 421)

**Weighted buddies**

Weighted buddies were first introduced by Shen and Peterson (1974). The system uses a custom size class series different from binary and Fibonacci buddy systems. The size classes include powers of two, but in between them, there exist sizes that are three times a power of two. For example, 2, 3, 4, 6, 8, 12... This means some sizes can be split in two ways. The address calculation is however quite straightforward and fast. (Wilson et al., 1995a, p. 40; Peterson & Norman, 1977, p. 421)

**Double buddies**

Double buddies were first introduced in (Wise, 1978). Double buddies offer a closer spacing of block sizes which is accomplished by using two binary buddy systems with staggered sizes. For example, the other binary buddy system could have size classes 2, 4, 8, 16... and the other could have sizes 3, 6, 12, 24... However, as with binary buddies, double buddies can only be split in half. (Wilson et al., 1995a, p. 40)

### 3.2.4 Indexed fits

Indexed fits is a “catch-all” category of allocators rather than a mechanism. For this reason, indexed fits cannot be explicitly analyzed as a mechanism. Indexed fit allocators use more complex data structures for block indexing to improve search times. Hence the desired allocation policy determines the most suitable indexing data structure. For example, a best fit policy allocator could be implemented using a binary tree data structure where blocks are ordered by size. (Wilston et al., 1995a, p. 40)

An example of an indexed fits allocator is Stephenson's (1983) “Fast fits” allocator. It uses Cartesian trees introduced by Vuillemin (1980) to store blocks sorted by both
address and size. Cartesian trees do not however necessarily maintain a good balance, and search executes in worst-case $O(N)$. (Wilson et al., 1995a, pp. 40-41; Stephenson, 1983, pp. 30-31)

### 3.2.5 Bitmapped fits

Bitmapped fits allocators use bitmaps to record which parts of memory are allocated and which parts are free. A bitmapped fits allocator divides memory in fixed-size chunks, and uses a bitmap to mark each chunk allocated or free. Linear mapping is often maintained, where a block address maps directly to a bit index in the bitmap. (Wilson et al., 1995a, pp. 41-42; Hasan & Chang, 2005, p. 37)

To allocate a block, bitmapped fits allocator scans the bitmap to find a suitable sequence of free chunks, marks them allocated, and returns a block spanning the chunks. To free a block, chunks hosting the block are marked free. (Wilson et al., 1995a, p. 42) Block size may be stored in block header or in another bitmap which marks the endings of the chunk sequences.

Bitmapped fits allocation is generally slow because of bitmap scanning. Some researchers suspect that bitmapped fits has never been used in conventional allocator (Wilson et al., 1995a, p. 42; Chang & Hasan, 2005, p. 37). We however observed that jemalloc (Evans, 2006) uses bitmaps for block bookkeeping.

In addition to bookkeeping, bitmaps can also be used for indexing akin to indexed fits. Many modern allocators use bitmap indexing: Half fit (Ogasawara, 1995), TLSF (Masmano et al., 2004) and jemalloc (Evans, 2006). This is probably caused by the availability of efficient processor bit-scan instructions.

### 3.2.6 Analysis on real-time use of basic mechanisms

In this section we will compare the basic allocator mechanism and analyze their suitability for real-time DMA. To determine the suitability, we need to understand the worst-case time and space properties of the mechanisms. We will use the analysis in this section to support our selection of evaluated allocation mechanisms later on. A summary of the analysis is presented in table 1 at the end of this section.

Sequential fits allocation executes in $O(N)$, where $N$ is the number of blocks on the free list. Search time increases when the free lists grows, which makes the mechanism unsuitable for real-time use. Sequential fit policies however produce low fragmentation, and best fit policy is generally considered to produce the least fragmentation of all known policies. Best fit or good fit policy can be implemented efficiently with indexed or segregated fits. (Wilson et al., 1995a, pp. 30-31, 33; Masmano et al., 2008a, pp. 166, 168; Masmano et al., 2004, p. 81; Hasan & Chang, 2005, p. 36; Puaut, 2002, p. 48; Johnstone & Wilson, 1998, pp. 27, 33-34, 36; Wilson et al., 1995b, p. 30)

All segregated free list mechanisms, except for size classes with range lists mechanism, are acceptable for real-time use since the search time is independent of the number of free blocks. Segregated storage allocators are all fast. SSS is likely to be the fastest, but it also has the worst fragmentation. (Masmano et al., 2004, p. 81; Hasan & Chang, 2005, pp. 44, 46-47; Wilson et al., 1995a, pp. 36-38; Grunwald, Zorn & Henderson, 1993, p. 185)
Buddy systems timing behavior is predictable. Buddy system operations have time complexity $O(\log_2 N)$, and the allocators is suitable for real-time applications. Buddy systems however suffer from high internal fragmentation. Research shows that internal fragmentation for binary, Fibonacci, double and weighted buddies is usually in the range 25-40%, and roughly 50% at worst-case, and binary and weighted buddies exhibit higher fragmentation than Fibonacci and double buddies. Overall fragmentation in buddy systems may be acceptable for real-time DMA. (Masmano et al., 2008a, p. 166; Masmano et al., 2004, p. 81; Puaut, 2002, pp. 46, 48; Yadav & Sharma, 2010, p. 66; Hasan & Chang, 2005, p. 37; Peterson & Norman, 1977, pp. 421, 429; Johnstone & Wilson, 1998, pp. 28, 34; Wilson et al., 1995a, pp. 38-40)

Indexed fits can perform better than segregated free lists in terms of WCET (Masmano et al., 2004, p. 81). Hence indexed fits are suitable for real-time DMA, but the indexing data structures used by the allocator must ensure low and bounded WCET. Indexed fits with bitmapped indexing are common, and bitmapped indexes are suitable for real-time allocators. Bitmapped indexing executes in constant-time when bit manipulation and bit-scan instructions are used (Masmano et al., 2004, p. 81; Hasan & Chang, 2005, p. 37).

Bitmapped fits (bitmap used for chunk bookkeeping) are normally unsuitable for real-time use. This is because bitmap scan generally performs in $O(N)$ time, where $N$ is the size of bitmap. The scan can be however improved, for example, with the techniques mentioned in section 3.1.6. Bitmapped fits have a constant overhead per chunk\(^7\). This could be used to reduce wasted memory in some implementations. (Wilson et al., 1995a, p. 42; Masmano et al., 2006, p. 69; Wilson et al., 1995b, p. 35)

### Table 1. Summary of basic allocator mechanisms suitability for real-time DMA.

<table>
<thead>
<tr>
<th>Basic Mechanism</th>
<th>Suitability for Real-Time DMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential fits</td>
<td>Not suitable. Sequential fits execute in $O(N)$, and this is unacceptable for real-time DMA.</td>
</tr>
<tr>
<td>Segregated free lists</td>
<td>Suitable except for size classes with range lists mechanism. Search time is independent of the number of free blocks.</td>
</tr>
<tr>
<td>Buddy systems</td>
<td>Suitable. Timing behavior is predictable. Internal fragmentation is potentially high, but overall fragmentation may be acceptable.</td>
</tr>
<tr>
<td>Indexed fits</td>
<td>Suitable, but this depends on the indexing data structure and its search cost. Low and bounded WCET is acceptable.</td>
</tr>
<tr>
<td>Bitmapped fits</td>
<td>Generally not suitable. Bitmap scan normally executes in $O(N)$, but scan can be improved. Low and bounded WCET is acceptable.</td>
</tr>
</tbody>
</table>

### 3.3 Other allocation mechanisms

Following sections will introduce mechanisms that are not explicitly low-level or basic allocation mechanisms. These mechanisms are sometimes discussed in the research literature, and we are not aware if any particular categorization exists for them. These mechanisms are relevant in the context of small block allocation and real-time DMA.

---

\(^7\) Overhead is one bit per chunk. Thus when chunk size grows, relative overhead diminishes. For example, an 8-byte chunk has 1.56% overhead from a single bit, but for a 32-byte chunk the relative overhead would be 0.39%.
3.3.1 BIBOP

Big bag of pages (BIBOP) is originally an object typing mechanism used in MACLISP (Steele, 1977) and later in Chez Scheme. In BIBOP, dynamically allocated objects are contained in aligned equal-size pages that contain objects of a single type. Object address high bits represent a page index which can be used to look up the object type. Since the type information is stored in page instead of objects, overhead is greatly reduced – especially when allocating small blocks. Large block allocation however does not necessarily benefit from BIBOP. (Steele, 1977, pp. 3-4; Dybvig, Eby & Bruggeman, 1994, pp. 5, 10, 13; Wilson et al., 1995a, p. 36; Schneider, Antonopoulos & Nikolopoulos, 2006, p. 85)

BIBOP mechanism needs to store the page information such as object type or block size. Dybvig, Eby and Bruggeman (1994) present three ways to for this: fixed page table, dynamic page table and page headers. Fixed page tables use a static table to record information of each page, and since full table is not usually utilized by all applications, majority of page table remains unused. Dynamic page table is similar to static page table, except that the table can grow and relocate to waste less memory. Page headers store the relevant information in a page head. (Dybvig, Eby and Bruggeman, 1994, p. 8)

Only static page table and page headers are suitable for real-time DMA, since they have predictable behavior. Dynamic page tables may need to relocate, and this requires a memory copy operation. Page headers are the most scalable alternative, but have worse locality characteristics (Dybvig, Eby and Bruggeman, 1994, p. 8).

BIBOP mechanism and its variations are used in many allocators. For example, region (see 3.3.2) and reap mechanisms (see 3.3.3) generally assume BIBOP. Also general-purpose allocators such as jemalloc (see 4.2.4) and Hoard (see 4.2.3) utilize BIBOP.

3.3.2 Regions

Regions (also known as arena, group or zone) allocate blocks simply by bumping a pointer across a range of memory. Blocks cannot be freed individually, but the entire region can be freed instead when none of its blocks are in use. Region allocation and free operations are very fast. (Berger, Zorn & McKinley, 2002, pp. 1-2, 5) Regions can be allocated in pages, and a free counter can be used to count the free operations in the region. Region allocation is illustrated in figure 6.

The inability to free individual blocks often complicates the use of regions in some applications. Additionally, regions may considerably increase memory consumption compared to other mechanisms, since a region cannot be freed unless all its blocks are unused. Compilers and parsers however may greatly benefit from them. (Berger, Zorn & McKinley, 2002, pp. 2, 4-7, 9)
3.3.3 Reaps

Reaps were introduced by Berger, Zorn and McKinley (2002). Reaps combine the favorable features of regions and heaps. They add to regions the possibility to free blocks anywhere inside the region without compromising performance. Reaps are shown to reduce memory consumption compared to regions. Reaps mechanism is used in the Hoard allocator. (Berger, Zorn & McKinley, 2002, pp. 1, 11)

Reaps first allocate memory like regions, with pointer bumping. When an individual block is freed, it is put to an associated free list. The reaps allocate memory in pages. When a page becomes full, another page is allocated and the reap allocator returns to the region style of operation. The original reap method adds block headers to every block. (Berger, Zorn & McKinley, 2002, p. 7) However, block headers are not necessary if BIBOP mechanism is used (Schneider, Antonopoulos & Nikolopoulos, 2006, pp. 85, 87). Reap allocation is illustrated in figure 7.

Figure 6. Illustration of region allocation. (Berger, Zorn & McKinley, 2002, p. 5)

```
regionCreate(r);
x1 = regionMalloc(r,8);
x2 = regionMalloc(r,8);
x3 = regionMalloc(r,16);
x4 = regionMalloc(r,8);
```

Figure 7. Illustration of reap operation. (Berger, Zorn & McKinley, 2002, p. 6)
4. Small Block Allocation Mechanisms in General-Purpose Allocators

This chapter contains analysis on mechanisms used for small block allocation in various general-purpose allocators. We will provide a short description of each general-purpose allocator and its small block allocation mechanisms. We will then summarize the mechanisms, and finally analyze the mechanism suitability for real-time use. This analysis is used as basis for selecting the mechanisms for the framework implementation in chapter 6.

We wanted initially to focus only on allocators designed for real-time systems, but because there exists only a few such allocators (Half fit and TLSF), we decided to broaden our scope to general-purpose allocators. We tried to select the most well-known or otherwise prominent allocators for analysis.

This chapter defines a general-purpose allocator to be a dynamic memory allocator that implements at least malloc and free functionality from the C language standard. In literature, such allocators are also often referred to as malloc replacements. These allocators can operate either as a shared language runtime allocator or a statically linked custom allocator contained within applications.

4.1 Motivation

It has been confirmed by multiple authors that modern programs make mostly small allocations (Wilson et al., 1995a, p. 36). Berger, Zorn and McKinley (2002, p. 8) measured memory use in various programs, and show that 88% of allocations are under 64 bytes and almost all (99.54%) are under 256 bytes. Similarly, Lee, Chang and Hasan (2000, p. 391) report that 90% of allocations are below 512 bytes, and have usually a short life-span. Small blocks are usually allocated in large quantities and large blocks in smaller quantities (Berger, Zorn & McKinley, 2002, p. 8; Hasan & Chang, 2005, pp. 45-46).

Multiple authors have studied the most common allocation size in programs. Wilson and others (1995a, p. 28) state that sizes average on the order of 10 machine words (40 bytes on a 32-bit machine). Measurements by Zorn and Grunwald (1992, p. 4) show that, for various programs, the most common allocation size is smaller than 32 bytes, and that the median block size is from 14 to 32 bytes. Later research by same authors confirm the result with other programs (Grunwald, Zorn & Henderson, 1993, p. 184). Measurements by Deflets, Dosser and Zorn (1994, p. 530) show that 39.6 bytes is the median of average allocation sizes in various large C and C++ programs.

C++ programs naturally tend to allocate large quantities of objects\(^8\) from a few size classes. In C++ programs, main sources of allocations are constructors, copy constructors and the overloaded assignment operator=. C++ programs may also use up

\(^8\) A C++ application may allocate 20 times more memory than equivalent C application (Hasan & Chang, 2005, p. 36).
to 38% of their total runtime in DMA. (Chang, Hasan & Lee, 2000, p. 7; Risco-Martin et al., 2011, p. 755; Lee, Chang & Hasan, 2000, pp. 387, 391)

Block headers are the main source of overhead when small blocks are allocated. A single word in a block header or footer can increase memory usage by 10% to 20% (Wilson et al., 1995a, pp. 28, 36). For example, if the block header is 4 bytes, alignment is 8 bytes, and we allocate a 32-byte block, then real size of the block is 40 bytes (=32+4+4, alignment adds 4 padding bytes), thus the resulting overhead from header is 20% (=8/40). Since a great majority of allocations are small, this scenario is very frequent. Wasted memory can be however reduced by using mechanisms with smaller overhead per block or by eliminating block headers. Some of these were described in section 3.3. Additionally bitmapped fits have low overhead.

4.2 Allocator descriptions and analysis

In this section and the following subsections, we will describe and analyze small block allocation mechanisms in the following general-purpose allocators: dlmalloc (Lea, 2011), Half fit (Ogasawara, 1995), Hoard (Berger et al., 2000), jemalloc (Evans, 2006), Kingsley allocator, and TLSF (Masmano et al., 2004). From the analyzed allocators, only Half fit and TLSF are designed for real-time use. A more thorough analysis was conducted on jemalloc and its source code, since the allocator relative new and no research literature was found to describe it. TLSF is also described more thoroughly since it is included in the simulation and evaluation in chapter 6.

The following allocators were excluded from the analysis: CustoMalloc (Grunwald & Zorn, 1993), PHKmalloc (Kamp, n.d.), QuickFit (Weinstock & Wulf, 1988), Slab allocator (Bonwick, 1994), and Zone allocator (Van Sciver & Rashid, 1990). According to Bonwick (1994, p. 4), QuickFit and CustoMalloc allocators require a priori knowledge of the common allocation sizes. Slab and Zone allocators also require client-driven (application specific) customization, and because of this the allocators are not general-purpose allocators by our definition. The Slab allocator is a kernel allocator (Bonwick, 1994, p. 11). The excluded allocators either share same functionality with the ones analyzed, or were omitted because of the limited scope of this study.

4.2.1 Dlmalloc

Doug Lea's general-purpose allocator is a well-known and established allocator, frequently addressed in the research literature. Dlmalloc is claimed to be an all-around general-purpose allocator with good average execution time and low fragmentation. It uses three categories of allocation sizes: small, medium and large, where small blocks are managed with segregated free lists. The allocator also uses deferred free to coalesce blocks. (Johnstone & Wilson, 1998, pp. 28, 36; Berger, Zorn & McKinley, 2002, pp. 2, 11; Masmano et al., 2004, p. 81; Risco-Martin et al., 2011, p. 756; Chang, Hasan & Lee, 2000, p. 8)

Masmano and others (2008a) however show that dlmalloc has very high WCET, and claim that it executes in O(N). Their measurements support a further claim that dlmalloc should not be used in real-time applications. (Masmano et al., 2008a, pp. 175, 166, 168) Current version of dlmalloc (Lea, 2011) however seems to have a configuration option for real-time systems. Unfortunately we noticed this too late, and the allocator was not
included in our evaluation. Instead, we chose to implement a SSS mechanism, since we believe this mechanism is fairly close to how dlmalloc allocates small blocks.

4.2.2 Half fit

Half fit allocator by Ogasawara (1995) is an allocator specifically designed for real-time systems. Its free and allocate operations have low bounded WCET and execute in $O(1)$. The allocator has possible worst-case fragmentation similar to binary buddies. (Ogasawara, 1995, p. 21; Masmano et al., 2008a, pp. 152-153, 175; Masmano et al., 2006, p. 68; Masmano et al., 2004, p. 81)

Half fit allocator maintains segregated free lists with blocks with sizes $2^k .. 2^{(k+1)} - 1$. It marks each of the lists empty or non-empty in a one word bitmap, which is searched in constant-time using bit-scan instructions. The bit-scan automatically uses the list of next available size if the list of requested size is empty. All blocks in the allocator have headers that contain at least links for a doubly linked-list. Efficient immediate coalescing and splitting is performed. The allocator has no special treatment for small blocks. (Ogasawara, 1995, p. 23; Masmano et al., 2008a, pp. 152-153, 175; Masmano et al., 2006, p. 69; Masmano et al., 2004, p. 81)

The Half fit allocator has lower WCET than TLSF, but has higher worst-case fragmentation. Otherwise it has significant similarities with TLSF. (Masmano et al., 2008a, pp. 150, 168; Masmano et al., 2006, pp. 73-74) Because of the similarities with TLSF, and Half fits higher worst-case fragmentation, we do not perform simulations with the Half fit allocator in chapter 6.

4.2.3 Hoard

Hoard allocator, introduced by Berger and others (2000), is a general-purpose allocator designed for deliver high-performance in multiprocessor systems. Hoard allocates memory through OS virtual memory system in large units called superblocks. A superblock can allocate a number of blocks of one size class only, and it contains a free list to store and reuse freed blocks in LIFO fashion. (Berger et al., 2000, p. 118)

Hoard recycles its free superblocks to reduce external fragmentation. Small blocks are allocated using superblocks, but blocks larger than half of superblock size are allocated by using OS virtual memory system. The allocator uses size classes power of b apart, where b is greater than 1. (Berger et al., 2000, pp. 119-120) Reaps mechanism was later used in place of superblocks in Hoard (Berger, Zorn & McKinley, 2002, p. 11). The allocator uses a mechanism similar to BIBOP to manage its superblocks.

4.2.4 Jemalloc

Jemalloc is a general-purpose allocator introduced by Jason Evans. It is an open-source high-performance allocator focused on multithreaded scalability and cache locality (Evans, 2006, p. 2). It is used in FreeBSD (Evans, 2006, p. 1), Mozilla Firefox and Facebook server. We analyze version 3.0.0 of the allocator.

The measurements by Evans (2006, pp. 7-11) show that jemalloc has slightly better overall performance compared to dlmalloc and PHKmalloc, and has good multithreaded
scalability. On multiprocessor systems, the allocator uses four arenas per processor, issuing one arena for one thread at a time. Use of thread-specific arenas improves multithreaded performance by eliminating locks. Single-processor systems use one arena only. (Evans, 2006, pp. 1-4, 7-8)

The allocator handles its memory in fixed 2 MB memory chunks requested from the underlying OS. The chunks are aligned in memory to allow constant-time calculation of chunk index from memory address high bits. (Evans, 2006, p. 4) The jemalloc chunks behave like large pages in BIBOP mechanism. Figure 8 illustrates chunk and arena allocation.

![Figure 8. Chunk and arena allocation in jemalloc. Huge allocations span multiple chunks.](Evans, 2006, p. 4)

The allocator handles blocks in three size categories: small (1, 2048] bytes, large (2, 1024] KB, and huge (1, +∞) MB. The small category has three subcategories: tiny 2 .. 8, quantum-spaced 9 .. 512 and sub-page 513 .. 2048 bytes. (Evans, 2006, p. 5) Each category is treated differently in the allocator. Chunks are divided to 4 KB pages when they store blocks from small category. Pages form page runs which store blocks of one size class. The page runs store a bitmap in their header for block bookkeeping. (Evans, 2006, pp. 5-6)

**Source code analysis**

We will next perform a brief source code review on jemalloc source code (Evans, 2012) to investigate its tiny and quantum-spaced size subcategory block allocation. Tiny and quantum-spaced subcategories contain the sizes 2 .. 512 bytes (Evans, 2006, p. 5).

Upon small block allocation request, the allocator first calculates an index to a cache bin using a lookup table. If the cache bin has free blocks, one is returned. This cache operations is similar to SSS. Otherwise, if no block is found from cache, arena bin pointer to a page run is checked, and if it is not null, the pointed page run's bitmap is searched for a free block which is then returned.
Otherwise, if the pointer to a page run was null (no page run), a binary buddy (red-black tree), containing page runs, is searched to find a suitable run for the bin size (Evans, 2006, p. 4). If a run is found, it is returned. Otherwise if no run is found, a new run is allocated from existing chunks by again searching the red-black tree. If this later search yields no chunks, new one run is allocated. When new run is created, its bitmap is also initialized.

When a small block is freed, it is first put in the cache for quick reuse. When the allocator has performed a number of operations (allocate and free), it performs a deferred free cycle to one cache bin. This cycle involves locating the parent chunk and run of the blocks in the bin, and using offset calculations and lookup techniques to mark the blocks free in the page run bitmap. The next coalescing cycle is then performed on the next bin.

**Summary**

Small block allocation and free are quite complex in the jemalloc. Basic mechanisms involve SSS cache with deferred free policy, bitmapped fits for actual block allocation from page runs, and binary buddies (red-black trees) for page run allocation from chunks. BIBOP is used to manage chunks on high-level.

The allocation operation in jemalloc is fairly complex in the worst-case. However, the regularities in the allocation request stream may converge the allocator to a state where most small block allocations can be satisfied directly from the SSS cache. This happens when application allocates a large number of small blocks. In such case jemalloc allocation may have amortized complexity \( T(n) \in O(1) \).

### 4.2.5 Kingsley allocator

Chris Kingsley's allocator implementation is another widely known allocator in addition to dlmalloc. It is a segregated free lists allocator with size classes of powers of two minus a some constant. The allocator is known to have high performance, and its allocation executes in \( O(1) \). The allocator is however known to have high worst-case fragmentation. Kingsley allocator has been used in FreeBSD 4.2. (Chang, Hasan & Lee, 2000, p. 8; Grunwald, Zorn & Henderson, 1993, p. 178; Risco-Martin et al., 2011, p. 756)

Kingsley allocator is suitable for real-time systems. We however note that the Kingsley allocator resembles strongly Half fit by its power-of-two size classes and segregated free lists. Because of the similarities, we do not analyze Kingsley allocator further in this study.

### 4.2.6 TLSF

Two-level segregated fit (TLSF) is a general-purpose real-time systems DMA introduced by Masmano, Ripoll, Crespo and Real (2004). Its allocation and free operations perform in \( O(1) \) and have low and bounded WCET. It uses the same allocation mechanism regardless of block size, and only a small variation in execution time can occur. The allocator implements a good fit policy. (Masmano et al., 2004, pp. 79, 83, 86-87; Masmano et al., 2008a, p. 175) TLSF can be seen as an extension to the Half fit (Ogasawara, 1995) allocator (Masmano et al., 2008, p. 150).
TLSF uses a large number of segregated lists containing blocks from different size ranges, and it uses a novel two-level indexing structure to reduce the list selection to a constant-time operation. This indexing is illustrated in figure 9. First-level index contains size ranges in power-of-twos, for example 16 .. 31, 32 .. 63, 64 .. 127 bytes, and so on, and then a second-level index divides these ranges linearly. For example, a size range 32 .. 63 bytes can be divided to four sub-ranges: 32 .. 39, 40 .. 47, 48 .. 55, and 56 .. 63 bytes. (Masmano et al., 2004, p. 83; Masmano et al., 2008a, pp. 157-158)

![Figure 9. Illustration of TLSF indexing data structure. (Masmano et al., 2004, p. 82)](image)

A word-size bitmap is used on both levels to mark free lists empty or non-empty. Bit-scan instructions are then used to perform the list selection in constant-time. The bit-scan search also automatically selects the free list of a larger size class if the free list of the desired size class is empty. (Masmano et al. 2004, pp. 83-85; Masmano et al., 2008a, p. 158)

The first and second level indexes are calculated from the requested allocation size. First level index is first obtained by locating the most significant set bit in the size by using bit-scan instructions. Second level index is then obtained from the following bits by using basic bit manipulation. (Masmano et al., 2004, p. 84; Masmano et al., 2008a, pp. 157-159) Figure 10 shows an example of the index calculation. First level index \( f \) is 8 and second level index \( s \) is 12. Second level index is represented by the 4 bits following the most significant set bit.

![Figure 10. Example of first and second level index calculation from allocation size. (Masmano et al., 2004, p. 84)](image)
TLSF blocks have headers, and block splitting and coalescing is performed immediately on allocation and free. Blocks are split if allocation occurs from a larger size free list than was requested. The authors claim roughly 3% internal fragmentation, and a low overall fragmentation. Memory requirement of internal data structures can also be calculated offline. (Masmano et al., 2004, pp. 84-85; Masmano et al., 2008a, p. 161; Masmano et al., 2006, p. 73) This makes the time and space costs of the allocator very predictable.

**Summary**
TLSF uses the same mechanisms to manage all blocks regardless of block size. Its allocate and free operations execute in O(1) time with very little variation in execution time. It uses two levels of bitmapped indexing, and implements a good fit policy. All blocks contain headers, and immediate coalescing and splitting is performed.

### 4.3 Summary

We have now described and analyzed small block allocation in various general-purpose allocators. Our intent in this section is to distinguish mechanisms that are more frequently used than others. A summary of the small block allocation mechanisms in the analyzed general-purpose allocators is presented in table 2.

**Table 2.** Summary of small block allocation mechanisms in general-purpose allocators from previous sections.

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Small Block Allocation Mechanisms (Other Mechanisms)</th>
<th>Designed for Real-Time?</th>
</tr>
</thead>
<tbody>
<tr>
<td>dlmalloc</td>
<td>Simple segregated storage, deferred coalescing, block headers</td>
<td>No.</td>
</tr>
<tr>
<td>Half fit</td>
<td>Segregated free lists, immediate coalescing and splitting, bitmapped indexing, block headers</td>
<td>Yes.</td>
</tr>
<tr>
<td>Hoard</td>
<td>Reaps, BIBOP</td>
<td>No.</td>
</tr>
<tr>
<td>jemalloc</td>
<td>Simple segregated storage, deferred coalescing, BIBOP (red-black trees/indexed fits, bitmapped fits and indexing)</td>
<td>No.</td>
</tr>
<tr>
<td>Kingsley</td>
<td>Segregated free lists, block headers, bitmapped indexing</td>
<td>No.</td>
</tr>
<tr>
<td>TLSF</td>
<td>Segregated free lists, two-level bitmapped indexing, block headers, immediate coalescing and splitting</td>
<td>Yes.</td>
</tr>
</tbody>
</table>

Segregated free lists are the most popular mechanism for small block allocation, and it is used by almost all of the allocators. Segregated free lists are a constant-time mechanism, offering a high performance and throughput necessary for small block allocation. Simple segregated storage is a specific type of segregated free list mechanism.

Another popular mechanism is bitmapped indexing, which is used by most of the allocators. Bitmapped indexes usually consist of a word-size bitmap which is used for constant-time segregated free list selection using bit-scan instructions. TLSF uses a special type of two-level bitmapped indexing.
Block headers are also stored by many of the allocators. Main reason for the use of blocks headers is probably the efficient coalescing it provides – links to previous and next block can be referenced quickly from the header. On the other hand, at least Hoard and jemalloc uses BIBOP or reaps to eliminate block headers to reduce per-block overhead. Immediate coalescing is performed by at least Half fit and TLSF, while other allocators perform either deferred coalescing, or no coalescing at all (Kingsley allocator, Hoard). While deferred coalescing is a good mechanism to reduce average execution time (to increase throughput), it is not suitable for real-time allocation (see section 3.1.4).

Based on the analysis, we are confident that both segregated free lists and bitmapped indexing are good mechanisms for small block allocation, and also for real-time DMA since both mechanisms have a low constant time cost. If block headers are used, they should be as small as possible, since they increase per-block overhead. Using BIBOP may be beneficial since it removes per-block overhead. While deferred coalescing improves average performance, it should not be used in real-time DMA, and immediate coalescing is preferred.
5. Bitframe Allocator Description

This chapter introduces Bitframe allocator, a new DMA aimed for small memory block allocation. Its allocation and free operations perform in O(1) time and have bounded WCET, making is suitable for real-time applications. The allocator was originally created as custom DMA for Lua core in a released Nintendo DS game.

The Bitframe allocator is based on the bitmap allocator mechanism where one bit stores the allocated/free state of a single memory chunk. To eliminate bitmap scanning the Bitframe allocator divides the bitmap in 8 bit bitframes and uses lookup tables to locate the longest bit sequence in each bitframe.

To allocate blocks spanning more than 8 chunks, the allocator uses a larger chunk size depending on the size class. A number of bitframes and their associated memory chunks are stored together in pages, where each page contains only bitframes having the same chunk size. The allocator manages pages with the BIBOP mechanism.

In the next section we describe the lookup tables in the allocator. Sections 5.2, 5.3 and 5.4 describe the data structures, and sections 5.5 and 5.6 describe the allocation and free operations. We conclude the chapter with an analysis of the allocator.

5.1 Lookup tables

The allocator has two lookup tables: one to look up the longest zero-bit sequence in a frame (lookup_longest) and the other to look up its starting index (lookup_longest_idx). These correspond to the longest free chunk sequence and its location. The frame bits are used directly as lookup table index during allocation. Figure 13 illustrates the process of allocating a chunk sequence using the lookup tables. Both lookup tables contains $2^8$ elements, totaling 512 items. The lookup tables can be packed, but this causes additional unpacking computation.

---

9 The 8 bits division is smallest choice, but this depends entirely on the implementation. For example 16 bits is also reasonable choice, but will have 256 times larger lookup tables ($2^{16}$ elements). This may be too much for some applications. To my knowledge there exist no suitable CPU instructions in common hardware which could be used in place of the lookup tables.
5.2 Bitframe data structure

Each bitframe manages 8 consequent chunks of memory and stores its state. For example, a bitframe with 64-byte chunk can allocate blocks in the range of 64 to 512 bytes.

The information stored in a bitframe:

- A bit for each chunk to mark it allocated or free (1 x 8 = 8 bits)
- Allocation termination bit for each chunk to store the information of the allocation lengths (1 x 8 = 8 bits)
- Links to previous and next bitframe in a circular doubly linked list (2 x 8 = 16 bits)

This totals to 32 bits\(^{10}\), resulting to 4 bits for each of 8 chunks in the bitframe. Figure 12 shows the data structure of the bitframe in more detail.

\(^{10}\)Again, an implementation could use more or less bits depending on the requirements.
5.3 Bitframe page

Each page contains a page header, a number of bitframes and the memory managed by the bitframes. All bitframes in the page have the same chunk size as defined by the page header. The following pseudo code shows the page header data structure.

```c
struct {
    cdl_node   size_class_pages;
    bitframe   *bitframes;
    u8          head_bits;
    u8          chunk_size_shift;
    void*       chunks;
} page_header;
```

The `bitframes` array contains the bitframes of the page and in addition 8 dummy bitframes at indexes 0...7 to serve as heads to circular doubly linked bitframe lists. This is to simplify the list management, since bitframes use 8-bit indexes as links to save space. The circular doubly linked lists store bitframes sharing the same longest free chunk length. The `head_bits` bitmap marks each of the lists empty or non-empty. Bitframes without free chunks are orphan and are not stored in any of the lists. Figure 13 illustrates the bitframe lists inside a page.

Each page in the allocator belong to a circular doubly linked list (`size_class_pages`) determined by its size class, or to no list if the page is full. The page size class is determined by the longest free chunk sequence of all bitframes in the page. This can be queried quickly from the index of the most significant set bit of the `head_bits`.

The `chunk_size_shift` is used in address calculations to chunks, and the pointer `chunks` provides the base address to the address calculation.

![Figure 13. Example of circular doubly linked bitframe lists. Empty list heads are marked with zeros. The gray bitframes in the bottom do not have free chunks and thus do not belong to any list.](image-url)
5.4 Bitframe size classes

The Bitframe allocator manages size classes relative to a quantum size which is the minimum allocatable block size. To calculate size classes, a maximum allocatable block size needs to be decided. Here we use 8-byte quantum and 512-byte maximum allocation, but both can be decided by the implementation.

Since we have 8-byte quantum and bitframes can store up to 8 chunk sequences first 8 size classes are 8, 16, 32 … 64 bytes. The next size classes use 64-byte chunk size which was the maximum of the previous chunk size. This results in size classes 64, 128, 192 … 512 bytes. The allocator has now two chunk sizes: 8 and 64, and 16 different size classes. Notice however that there are two 64-byte size classes, so the allocator must decide which to prefer on allocation.

The Bitframe allocator has an array of circular doubly linked lists to store one list for each size class (size_class_lists), and a bitmap to mark the lists non-empty or empty (size_class_bits). The lists link together pages sharing the associated size class. By using the size class bitmap, allocator can rapidly locate a page with suitable sequence of free chunks to satisfy the allocation.

5.5 Allocate operation

We assume that the Bitframe allocator has initially allocated a page for all possible size classes. When page is initialized, all its bitframes are cleared to longest free sequence of 8 chunks and are put on the 8th circular double linked bitframe list11.

Upon allocation, the size class of the requested block size is first calculated. Then the size_class_bits in the allocator are scanned starting from the calculated size class bit to find the best suitable page. The resulting page will have size class greater than or equal to the requested block size class and is guaranteed to satisfy the allocation. The page bitframe head contains a bitframe which satisfies the allocation request. Following pseudo code explains this procedure.

```cpp
class_idx = calculate_size_class(size_bytes)
class_idx = bsf(size_class_bits >> class_idx) + class_idx
page = size_class_lists[class_idx].next
head_idx = (size_bytes - 1) >> page.chunk_size_shift
bitframe_idx = page.bitframes[head_idx].next
```

Here `size_bytes` is the requested block size. The procedure `calculate_size_class()` calculates the size class index from given block size. The array `size_class_lists` and variable `size_class_bits` contain the size class list heads and state (non-empty or empty). The procedure `bsf()` is a bit-scan instruction.

After the suitable bitframe has been located, a chunk index for the block is looked up using the bitframe bits. Then, a linear mapping is performed to obtain a pointer to the block returned by the allocator. The following pseudo code illustrates the chunk index lookup and pointer mapping.

```cpp
bitframe_bits = page.bitframes[bitframe_idx].bits;
chunk_idx = lookup_longest_idx[bitframe_bits]
```

11Naively this involves initializing every bitframe in the page which is a costly operation. Alternatively a pointer-bumping mechanism can be used to initialize new bitframes in the page when they are needed.
offset = (chunk_idx + (8 * bitframe_idx))
    << page.chunk_size_shift
result = page.chunks + offset

After this, the bitframe bits and block terminator bits are modified accordingly. The longest free chunk sequence in the bitframe is also likely changed, and bitframe is transferred to a list matching the new longest free sequence length. The change in bitframe lists state may also affect the size class of the page, and the page is transferred to a list matching its new size class.

All the previously mentioned steps of the allocation operation execute in O(1) time, taking advantage of lookup tables, bit manipulation, bit-scan instructions and circular linked lists. Since total time spent in an algorithm is the sum of all its steps, the resulting time complexity of the allocation operation is O(1).

5.6 Free operation

The Bitframe allocator uses BIBOP mechanism to manage pages. The mechanism aligns pages to a fixed page size. When a block is freed, its address is used in a simple calculation to determine the page which contains the block. After the page is determined, bitframe and the chunk indexes in the page calculated from the block address relative to page's chunks pointer. This is the reverse of the mapping performed in the allocation operation. After this, the calculated bitframe termination bits are scanned from the starting chunk index to determine the length of the allocation in chunks. The chunk length is used to create a bit mask to clear the bitframe bits. Bitframe termination bits are also cleared with the same bit mask.

Similarly to the final steps in the allocation operation, the free operation alters both longest free chunk sequence in the bitframe and the page size class. The allocator needs to transfer the bitframe and the page to a list matching the new state. All the previously mentioned steps execute in O(1) time, including bitmap scanning which is performed with bit-scan instructions. Thus as with the allocation operation, free also performs in O(1) time.

5.7 Analysis

Like all DMA, bitframe allocator has its shortcomings. A major limitation is set by the bitframe data structure, which constrains the maximum allocatable sequence to 8 chunks. This reduces the flexibility of the allocator.

There are issues in current size class calculation which considerably increase internal fragmentation when allocating specific sizes. A worst-case example is the allocation of 65 bytes. This would use two 64-byte chunks to store the 65-byte block causing roughly 50% memory to be wasted. One solution to this problem would be to constrain the smallest chunk sequence length allowed in a bitframe.

The bitframe data structure also limits the organization of chunk sequences. A chunk sequence must end at every 8th chunk and cannot continue to the first chunk on the next bitframe. This has a serious limitation to the maximum number of possible allocations of sequences longer than 4 chunks (half of the bitframe 8 bits). A bitframe can only contain one sequence that is longer than 4 chunks. Thus a page with M bitframes can
only contain M blocks spanning more than 4 chunks. This is a notable limitation especially for blocks spanning 5 chunks.

One solution to the previous problem is to limit the maximum sequences to 4 chunks while keeping the bitframe size of 8 bits. This solution is only partial, since then sequences of 4 chunks would then have the same maximum number of allocations as sequences of 3 chunks (maximum two such sequences per bitframe). This solution also increases the number of chunk sizes for pages which complicates the size class calculation.

Better solution to the previous problem would be to allow the allocations to cross bitframe boundaries. This would involve reading bits from consequent bitframes and merging them before bit manipulation. However, as a results lookup tables would need to have more items to cover the new merged bits. For example, with maximum sequence length of 8, the new lookup tables would require $2^{15}$ items. However for maximum sequence length of 4 chunks (discussed in previous paragraph), the new lookup tables would require $2^{11}$ elements.

### 5.8 Conclusion

The bitframe allocator is an O(1) DMA with bounded WCET suitable for real-time applications. The allocator is designed for rapid allocation of small memory blocks. There are possibilities for reducing the fragmentation in the allocator. One possibility would be to allow the allocation of chunk sequences across frame boundaries, and another would be to limit the sequence lengths.
6. Simulation and Evaluation

For the simulation, we created an ad hoc framework with a total of seven different allocators. Six of the allocators were implemented from scratch, and for the seventh we used an open-source implementation by Masmano, Ripoll, Brugge and Scislowicz (2008b). The framework was written in C, but had occasional blocks of inline assembly. The source code is available at http://bitbucket.org/tsone/memwork

Our framework operates similarly to trace processor described in (Johnstone & Wilson, 1998, p. 31). Memory allocation traces are given as input to the framework which performs simulation and produces output logs of either timing (cycles) or memory use. The output logs contain information on each allocation and free operation of the input trace. The simulation used 3 MB heap size for all traces.

A separate program processes the logs and produces plots and analysis results. Following the evaluation methodology in (Masmano et al., 2008a, p. 162), the program calculated worst-case, mean and standard deviation of execution time from trace simulations. It created plots containing information on allocation and free operation cycles, allocated memory in the trace, and memory use, internal fragmentation, implementation overhead by the allocators. It calculated fragmentation with the equation 1 described in section 2.4.1. Following this method, it also calculated the ratio of implementation overhead on fragmentation with the following equation.

\[
I = \frac{L}{H - M}
\]

Here \(L\) is the maximum implementation overhead during simulation, \(H\) is the maximum memory used by the DMA, and \(M\) is the maximum allocated live memory used by the application during the trace.

The simulations were performed on Acer Aspire One ZG5 netbook, running an Intel Atom N270 CPU at 1.6 GHz with 512 KB L2 cache. The machine had 512 MB of RAM which front-side bus speed was 533 MHz. Fedora 17 LXDE GNU/Linux was used as OS, and a prebuilt kernel version 3.6.7 was used. We did not modify the kernel. The test framework was compiled with GCC version 4.7.2\(^{12}\).

The next three sections will describe the memory traces, the simulated allocator implementations, and an analysis on the worst-case behavior of the implementations. The sections are followed by description of the timing and fragmentation measurement methodology. This is followed by sections presenting simulation results and analysis for both timing and memory measurements separately. We end the chapter with an analysis of the overall efficiency of the implemented mechanisms.

\(^{12}\)We also tried Clang version 3.0. While Clang produced faster code for some allocators, GCC provided better overall performance.
6.1 Memory traces

We used four traces for the simulation: uniform, small, boot and stable. The uniform and small traces were synthesized from probabilistic distributions, and boot and stable traces were obtained from an actual real-time embedded system. Only allocations of 1 .. 512 bytes were included in the traces. Figures 14, 15, 16 and 17 show the number of allocations for each size class in all of the traces, and figures 18 and 19 show the allocated live memory for boot and stable traces. The size classes are 8 bytes apart (8, 16, 32 … 512).

The uniform and small traces were synthesized from probabilistic distributions, and neither tries to mimic real application behavior. Instead, we use synthetic traces to give information on worst-case fragmentation behavior of allocators when allocating small block sizes, since it is mandatory to reveal worst-case behavior in real-time context (see section 2.5). Both of the traces have random object lifetimes; for every other allocated block a block is randomly freed from the live blocks. The uniform trace allocates all sizes uniformly, and the small trace allocates mainly small blocks following a normal distribution. Heap sizes in both traces exhibit linear growth.

We modeled the normal distribution for small trace according to the results in (Zorn & Grunwald, 1992, p. 4) and (Berger, Zorn & McKinley, 2002, p. 8). Their results show that roughly 90% of allocations in real programs are below 64 bytes (see section 4.1). We used a normal distribution with a mean 32 bytes and standard deviation of 19.455 bytes to ensure 90% of allocations fall under 64 bytes. We additionally ensured that every block size over 108 bytes was allocated at least once.

The boot and stable traces are both subsets of a larger trace recorded from a real-time embedded system allocator, and reflect the boot and stable phases mentioned in (Masmano et al., 2008a, p. 175). The large trace recorded all allocation and free operations from boot to a system stable phase. The boot trace is a subset of this trace, and captures the boot phase of the system, starting from its beginning and ending at a position where heap size stops growing. The stable trace is also a subset of the larger trace, and captures the stable phase of the system, containing roughly 500000 allocation and free operations.
Figure 14. Number of allocations for each size class in uniform trace.

Figure 15. Number of allocations for each size class in small trace.

Figure 16. Number of allocations for each size class in boot trace.

Figure 17. Number of allocations for each size class in stable trace.

Figure 18. Allocated live memory in boot trace.

Figure 19. Allocated live memory in stable trace.
6.2 Allocator implementations

We implemented six allocators from scratch and used an open source implementation for the seventh allocator. Total seven allocator implementations were used: bbuddy, bframe, reaps, regis1, regis2, sss and tlsf. These allocators implement mechanisms we found suitable for small block allocation and real-time DMA based on our survey and analysis in chapters 3 and 4. Segregated free lists mechanism was used in all of the implementations except bbuddy. We also chose to experiment with BIBOP mechanism, since it can reduce overall wasted memory when small blocks are allocated. BIBOP is implemented in bframe, reaps, regis1 and regis2. Additionally, bitmapped indexing was also used by bbuddy, bframe and tlsf. We will next describe the implementations in more detail.

A region-based allocator regis1 uses region mechanism to allocate blocks. It maintains a single region at a time stored in a single page. When the page becomes full it is discarded, and a new page is allocated with a new region. Page headers store a minimal footer containing only a single live counter, and the page is freed when its live counter becomes zero. Page size for this allocator is 1024 bytes, and the allocator uses BIBOP mechanism to manage its pages.

Size class region allocator regis2 is similar to regis1, but maintains a region for every size class. The classes are 8 bytes apart with sizes 8, 16, 24 … 512 bytes, totaling to 64 size classes. Each region is stored in its own page and will allocate only blocks in one size class. The page headers store a live counter and a size class id. As with regis1, a page is freed when its live counter becomes zero.

Reap allocator reaps uses reap mechanism to allocate blocks and manages pages with the BIBOP mechanism. Each page contains a reap which allocates blocks of a single size class. Size class sizes are 8, 16, 24 … 512 bytes, totaling 64 size classes. The page headers store the reap data structure, a live counter, the page size class and a circular doubly linked list node. The circular list is maintained to connect pages having free blocks of the same size class. Pages are removed from the list when page becomes full (no free blocks in page) or when live counter becomes zero, in which case the page is also freed. The page size 1 KB was used.

Binary buddy allocator bbuddy is based on the description by Knuth (1973, pp. 442-445). Our implementation however divides the heap initially to 1024-byte buddies because larger than 512-byte blocks are not allocated in our experiments. Buddies have a 16-byte header, so to host both the buddy header and a minimum 1 byte block, a buddy must have 32-byte minimum size. This means buddy size range is between 32 and 1024 bytes (or 2^5 and 2^10) and so maximum number of splits and merges by buddy system is 5. This means allocate and free operations in bbuddy execute in O(1) time.

Bitframe allocator bframe was implemented as described in chapter 5. Bitframe page size was 16 KB. This is the maximum amount of memory a page can hold for the minimum chunk size of 8 bytes.

Simple segregated storage allocator sss was implemented following the description in section 3.2.2. Our implementation uses size classes of 8, 16, 24 … 512 bytes. When a free block is not found on a size class list, our implementation allocates a new block

---

13 Notice that in our implementation a 1024-byte buddy is used to store a 512-byte allocation. This is because the buddy also needs to store its header. So, in the case of 512-byte allocation, almost 50% of memory is wasted to internal fragmentation. This is an example of worst-case fragmentation by binary buddy mechanism.
from heap by pointer-bumping. Block headers contain only a pointer to a size class data structure matching the block size. When a block is freed, it is placed on the segregated free list of the size class.

TLSF allocator implementation *tlsf* is based on source code provided by Masmano, Ripoll, Brugge and Scislowiczthe (2008b). The source code was slightly modified and stripped down, and we added instrumentation code to integrate with the test framework.

6.3 Worst-case analysis of the allocator implementations

This section will present an analysis of the worst-case behavior of the allocator implementations presented in the previous section. As described in section 2.5, for real-time DMA, we need to understand the possible WCET of the allocator (both allocate and free operations) and also possible worst-case fragmentation or wasted memory. We performed the by manually inspecting the source code. For *buddy* and *tlsf*, we also relied on the worst-case analysis from earlier research literature.

*Bbuddy* allocation WCET occurs when a block of the minimum size is requested and the heap is empty. This causes the allocator to perform a maximum number of splits to satisfy the allocation. Free WCET occurs in an opposite case, when the last block is freed, and the freed block has minimum size. This causes the allocator to perform a maximum number of merges to coalesce the blocks. (Masmano et al., 2008a, p. 165)

Our implementation has a large block header (16 bytes), and this causes reasonable overhead. Also because of the header, sizes slightly less than power of two may cause considerable internal fragmentation. For example, a 512-byte allocation would need a 1024-byte buddy, and waste roughly 500 bytes.

*Bframe* allocation WCET occurs when a page contains no free chunks, and a new page must be allocated for the allocation. Our implementation is not well optimized for the page allocation, and the operation takes a long time to complete. The allocator free WCET occurs when the longest chunk sequence in a page changes, causing both a bitframe and a page to be transferred to another doubly linked list. In our implementation, worst-case internal fragmentation is caused by allocating blocks with size of 65 bytes, which causes roughly 100% internal fragmentation. Worst-case overall fragmentation occurs when pages are allocated but not used. This causes considerable overhead from bitframe and page bookkeeping data structures and external fragmentation.

Both *regis1* and *regis2* have high internal fragmentation. Pages in the allocators cannot be freed unless all blocks of the page are freed. On the other hand, the allocator has almost no implementation overhead because block headers are not used and page footer is very small. WCET in the allocation operation occurs when a new page is allocated, and similarly WCET in the free operation occurs when a page is freed. WCET bound is very low overall.

The *reaps* allocator has low internal fragmentation and implementation overhead. Similarly to *regis1* and *regis2*, allocation and free operation WCET occurs when a new page is allocated or freed.

The *sss* allocation WCET behavior occurs when no suitable free block is found on segregated lists, and a new block must be allocated by pointer bumping. This situation also causes fragmentation to accumulate, since the allocator is not effectively reusing
memory. The free operation contains no branches, thus it should always execute in the same number of cycles\textsuperscript{14}.

The \textit{tlsf} allocation WCET behavior occurs when a small block is allocated and the allocator has only one large free block. WCET for free operation occurs when a freed block has two neighbors that are coalesced. Worst-case internal fragmentation is expected to be 3\%. (Masmano et al., 2008a, pp. 161, 166)

Based on this worst-case analysis, we are certain that the \textit{boot} and \textit{stable} traces will contain allocation and free operations which will cause WCET behavior in all allocator implementations except \textit{bframe}. The page size in \textit{bframe} is so large that the allocations in \textit{boot} and \textit{stable} traces do not cause new pages to be allocated. We are certain however that our traces will cause worst-case fragmentation behavior in all of the implementations.

### 6.4 Timing measurement method

All of the memory instrumentation code was omitted when the framework was compiled for timing measurements. We used the following GCC options to compile code specifically for Intel Atom and to enable aggressive optimizations for performance\textsuperscript{15}:

```
-O3 -march=atom -mtune=atom -fomit-frame-pointer -finline-functions -fno-stack-protector -ffast-math
```

Our timing instrumentation code used RDTSC instruction which returns CPU cycles elapsed since boot. Proper use of RDTSC is however error-prone, and many sources can interfere with the timing measurements: OS scheduling, simultaneous multithreading and interrupt handling, CPU out-of-order execution, CPU power modes and cache operation. Modern CPUs use every possible method to improve throughput (average execution time), and these methods will interfere with the measurements (Masmano et al., 2008a, p. 162).

CPU out-of-order execution is not a problem on the Intel Atom processor since it does not reorder instructions. We can also disable CPU power mode changes and simultaneous multithreading from the kernel. We cannot however disable the CPU cache\textsuperscript{16}, and CPU cache was probably the single most important source of interference in our measurements.

We implemented our RDTSC instrumentation code following the guidelines in (IBM, 2011) and (Paoloni, 2010). According to (Paoloni, 2010, p. 7), RDTSC instrumentation should be done in a kernel module because it allows exclusive ownership of the CPU. Unfortunately our test framework is a user space program, so we needed to take extra actions. We took the following steps to ensure all possible CPU time was granted to our

\footnote{CPU cache operation affects the execution time, and the measured execution time will fluctuate.}

\footnote{We observed that compiler optimizations had a large impact on our timing measurements. We chose aggressive optimizations since we believe most real-time applications also want this level of optimizations. We are aware that aggressive optimizations may introduce bugs.}

\footnote{We tried to disable CPU cache from the CPU CR0 register, but this caused the measurements to become more erratic. Disabling the CPU cache also slowed down the system to a grinding halt.}
simulation process, and also to minimize interference from various sources in the system:

- Linux kernel was given following commands on boot:
  
  ```
  nosmp nohalt idle=poll
  ```
  
  This disables symmetric multiprocessing (resulting in single core operation),
  disables CPU halting and forces explicitly kernel idle loop to poll.

- The system was booted to runlevel 1 and all possible OS daemons were stopped.
  (IBM, 2011, pp. 5-6)

- Linux kernel real-time throttling was disabled with:
  
  ```
  echo -1 > /proc/sys/kernel/sched_rt_runtime_us
  ```
  
  This permits our framework's real-time process to use 100% of CPU time. (IBM,
  2011, p. 6)

- CPU power saving was disabled by setting scaling governor to “performance”:
  
  ```
  echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
  ```
  
- Test framework process was run as root, and process was elevated to highest
  available scheduling priority with real-time SCHED_FIFO policy (IBM, 2011, p.
  4). This ensures framework process will not be preempted unless a process with
  greater or equal priority is issued (which is not often, since real-time throttling
  was disabled).

- Interrupts were disabled during instrumentation with CLI and STI instructions.

- Test framework process stdout was forwarded to /dev/null.

Each allocation trace was run 1000 times, and the cycles for each allocation and free
operation were measured. We then calculated a minimum of each operation cycles,
resulting in an optimistic estimate. Our aim was to eliminate interference from the CPU
cache, but this did not fully succeed. It only worked when heap size was well below
CPU cache size (512 KB). Otherwise we observed notable interference. It made timing
measurements from uniform and small traces unusable for our evaluation, and we
decided to omit them.

### 6.5 Fragmentation measurement method

Fragmentation was measured by instrumentation code which was added to relevant
positions and functions in the allocator implementations. We used more relaxed
optimization options for memory measurements.

Maximum heap address was calculated from the highest address touched by the
allocator – from the highest address of any page or block allocated by the allocator. The
maximum heap address was calculated relative to a heap start address which was
aligned depending on the allocator in question. The resulting alignment padding was omitted from the measurements.

6.6 Timing results

This section presents the timing measurement results. The figures 20 and 21 show measured cycles for allocate and free operations during boot trace, while figures 22 and 23 show cycles for stable trace. These plots are similar to ones in (Masmano et al., 2006, pp. 73-74). Measured maximum, mean and standard deviation for boot and stable traces are found in tables 3 and 4.

Note that there is some interference in our measurements. We will address this issue in the next section. The uniform and small trace measurements contained a much higher amount of the interference, and for this reason we decided to omit them.
Figure 20. Allocate operation cycles in boot trace.

Figure 21. Free operation cycles in boot trace.
Figure 22. Allocate operation cycles in stable trace.

Figure 23. Free operation cycles in stable trace.
Table 3. Measured maximum, mean and standard deviation of allocate and free operations in boot trace. Units are in CPU cycles.

<table>
<thead>
<tr>
<th></th>
<th>Maximum</th>
<th>Mean</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Allocate</td>
<td>Free</td>
<td>Allocate</td>
</tr>
<tr>
<td>bbuddy</td>
<td>444</td>
<td>240</td>
<td>119</td>
</tr>
<tr>
<td>bframe</td>
<td>264</td>
<td>216</td>
<td>215</td>
</tr>
<tr>
<td>reaps</td>
<td>132</td>
<td>108</td>
<td>56</td>
</tr>
<tr>
<td>regis1</td>
<td>156</td>
<td>72</td>
<td>37</td>
</tr>
<tr>
<td>regis2</td>
<td>144</td>
<td>108</td>
<td>55</td>
</tr>
<tr>
<td>sss</td>
<td>96</td>
<td>60</td>
<td>49</td>
</tr>
<tr>
<td>tlsf</td>
<td>312</td>
<td>312</td>
<td>186</td>
</tr>
</tbody>
</table>

Table 4. Measured maximum, mean and standard deviation of allocate and free operations in stable trace. Units are in CPU cycles.

<table>
<thead>
<tr>
<th></th>
<th>Maximum</th>
<th>Mean</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Allocate</td>
<td>Free</td>
<td>Allocate</td>
</tr>
<tr>
<td>bbuddy</td>
<td>264</td>
<td>432</td>
<td>114</td>
</tr>
<tr>
<td>bframe</td>
<td>276</td>
<td>228</td>
<td>214</td>
</tr>
<tr>
<td>reaps</td>
<td>396</td>
<td>264</td>
<td>57</td>
</tr>
<tr>
<td>regis1</td>
<td>204</td>
<td>264</td>
<td>39</td>
</tr>
<tr>
<td>regis2</td>
<td>288</td>
<td>288</td>
<td>57</td>
</tr>
<tr>
<td>sss</td>
<td>252</td>
<td>216</td>
<td>51</td>
</tr>
<tr>
<td>tlsf</td>
<td>660</td>
<td>396</td>
<td>163</td>
</tr>
</tbody>
</table>

6.7 Timing results analysis and evaluation

We note that there is some interference in our measurements. The sss free operation has no branches and should always use the same number of cycles. However the measurements show there is variation, and this behavior is also deterministic since simulation was run 1000 times for every trace. Hence we are confident the interference is caused by the CPU cache. The interference is higher in the stable trace, and we believe that the measured maximum cycles in stable trace are not accurate enough to be used in analysis.

The results with uniform and small traces contained even more interference, and we have omitted the results. The higher amount of interference is explained by the size of
the traces and the larger amount of allocated live memory. Since the amount of active memory is higher during simulation, probability of CPU cache misses is higher. Cache misses cause long delays in memory accesses, causing the interference.

The cycle plots however show that all of the implementations have bounded and low WCET, and fulfill this requirement for real-time DMA. The measured standard deviation in execution times is also relatively small.

The timing measurements show that *bbuddy*, *bframe* and *tlsf* use more cycles than others. Standard deviation is highest with *tlsf*, but we believe this is primarily caused by CPU cache interference. The highest mean execution time is with *bframe*, however has comparably good WCET. The *bbuddy* performs surprisingly well compared to others. The *regis1*, *regis2*, *reaps* and *sss* allocators are shown to be fastest and most predictable WCET.

The large page size in *bframe* plays a large role in the measurements. The *boot* and *stable* traces are too small to cause new pages to be allocated by the implementation, thus the *bframe* allocation WCET condition does not occur. Note that we did not have time to implement a faster page allocation, while it would have been possible\(^ {17}\). Because of this, the new page allocation operation in *bframe* takes a very long time to complete, which would have distorted the WCET results. We believe that our measurements gives a hint on the WCET performance of the Bitframe allocator.

### 6.8 Fragmentation results

The figures 24 and 25 show the maximum memory used by allocators (top continuous lines) and allocated live memory (gray at the bottom) in *boot* and *stable* traces. The figures visualize fragmentation by each allocator: the higher the maximum memory use of the allocator in figure, the higher the fragmentation.

Figures 26 and 27 show the implementation overhead for the allocators in *boot* and *stable* traces. And figures 28 to 33 show the sum of internal fragmentation and implementation overhead as total wasted memory by the allocators in *boot* and *stable* traces. These figures display the ratio of implementation overhead to total wasted memory and internal fragmentation by each allocator.

We have omitted previous figures of the *uniform* and *small* traces for two reasons. Firstly, the *uniform* and *small* traces figures only displayed linear growth of memory usage without special cases, since the traces contain twice the amount of allocations than frees. Secondly, the traces do not represent real-world memory usage, and they are only used in small block allocation worst-case fragmentation analysis.

Tables 5 to 8 present the calculated fragmentation and the ratio of implementation overhead for each allocator during *boot*, *stable*, *uniform* and *small* traces. Fragmentation was calculated with equation 1 (see section 2.4.1). Ratio of implementation overhead to fragmentation was calculated with equation 2 (see section 6). The results from *boot* and *stable* traces represent real-word usage, whereas the *uniform* and *small* traces represent worst-case fragmentation when allocations of specific size are made.

---

\(^ {17}\)The problem with Bitframe page allocation is more precisely in page initialization. An efficient way to implement Bitframe page initialization is discussed in footnote 11 in section 5.5.
Figure 24. Allocated live memory and maximum memory use in boot trace.

Figure 25. Allocated live memory and maximum memory use in stable trace.
Figure 26. Implementation overhead in *boot* trace.

Figure 27. Implementation overhead in *stable* trace.
Figure 28. Sum of internal fragmentation and overhead (dashed line) in boot trace for bframe, reaps and sss.

Figure 29. Sum of internal fragmentation and overhead (dashed line) in stable trace for bframe, reaps and sss.

Figure 30. Sum of internal fragmentation and overhead (dashed line) in boot trace for bbuddy and tlsf.

Figure 31. Sum of internal fragmentation and overhead (dashed line) in stable trace for bbuddy and tlsf.

Figure 32. Sum of internal fragmentation and overhead (dashed line) in boot trace for regis1 and regis2.

Figure 33. Sum of internal fragmentation and overhead (dashed line) in stable trace for regis1 and regis2.
Table 5.Maximum used memory (in bytes) and calculated fragmentation of each allocator in boot trace. Maximum live memory in boot trace was 16729 bytes.

<table>
<thead>
<tr>
<th></th>
<th>Maximum Memory Used by DMA</th>
<th>Fragmentation%</th>
<th>Implementation Overhead%</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbuddy</td>
<td>34408</td>
<td>105.68%</td>
<td>49.01%</td>
</tr>
<tr>
<td>bframe</td>
<td>33428</td>
<td>99.82%</td>
<td>16.46%</td>
</tr>
<tr>
<td>reaps</td>
<td>36368</td>
<td>117.39%</td>
<td>7.68%</td>
</tr>
<tr>
<td>regis1</td>
<td>85016</td>
<td>408.20%</td>
<td>0.52%</td>
</tr>
<tr>
<td>regis2</td>
<td>67088</td>
<td>301.03%</td>
<td>2.08%</td>
</tr>
<tr>
<td>sss</td>
<td>30076</td>
<td>79.78%</td>
<td>34.08%</td>
</tr>
<tr>
<td>tlsf</td>
<td>25680</td>
<td>53.51%</td>
<td>83.74%</td>
</tr>
</tbody>
</table>

Table 6. Maximum used memory (in bytes) and calculated fragmentation of each allocator in stable trace. Maximum live memory in stable trace was 9410 bytes.

<table>
<thead>
<tr>
<th></th>
<th>Maximum Memory Used by DMA</th>
<th>Fragmentation%</th>
<th>Implementation Overhead%</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbuddy</td>
<td>14440</td>
<td>53.45%</td>
<td>38.01%</td>
</tr>
<tr>
<td>bframe</td>
<td>33428</td>
<td>255.24%</td>
<td>11.44%</td>
</tr>
<tr>
<td>reaps</td>
<td>26128</td>
<td>177.66%</td>
<td>7.35%</td>
</tr>
<tr>
<td>regis1</td>
<td>38936</td>
<td>313.77%</td>
<td>0.60%</td>
</tr>
<tr>
<td>regis2</td>
<td>49680</td>
<td>427.95%</td>
<td>2.26%</td>
</tr>
<tr>
<td>sss</td>
<td>42532</td>
<td>351.99%</td>
<td>3.54%</td>
</tr>
<tr>
<td>tlsf</td>
<td>14280</td>
<td>51.75%</td>
<td>84.60%</td>
</tr>
</tbody>
</table>
Table 7. Maximum used and live memory (in bytes) and corresponding calculated fragmentation for each allocator in uniform trace.

<table>
<thead>
<tr>
<th>Allocators</th>
<th>Maximum Memory Used by DMA</th>
<th>Maximum Live Memory</th>
<th>Fragmentation%</th>
<th>Implementation Overhead%</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbuddy</td>
<td>3149928</td>
<td>2164377</td>
<td>45.54%</td>
<td>13.68%</td>
</tr>
<tr>
<td>bframe</td>
<td>3130004</td>
<td>2418900</td>
<td>29.40%</td>
<td>28.13%</td>
</tr>
<tr>
<td>reaps</td>
<td>3149328</td>
<td>2599579</td>
<td>21.15%</td>
<td>15.76%</td>
</tr>
<tr>
<td>regis1</td>
<td>3148824</td>
<td>1716651</td>
<td>83.43%</td>
<td>0.86%</td>
</tr>
<tr>
<td>regis2</td>
<td>3149328</td>
<td>1825517</td>
<td>72.52%</td>
<td>1.90%</td>
</tr>
<tr>
<td>sss</td>
<td>3149724</td>
<td>3000743</td>
<td>4.96%</td>
<td>62.94%</td>
</tr>
<tr>
<td>tlsf</td>
<td>3149744</td>
<td>2970255</td>
<td>6.04%</td>
<td>53.33%</td>
</tr>
</tbody>
</table>

Table 8. Maximum used and live memory (in bytes) and corresponding calculated fragmentation for each allocator in small trace.

<table>
<thead>
<tr>
<th>Allocators</th>
<th>Maximum Memory Used by DMA</th>
<th>Maximum Live Memory</th>
<th>Fragmentation%</th>
<th>Implementation Overhead%</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbuddy</td>
<td>3149928</td>
<td>1518551</td>
<td>107.43%</td>
<td>42.10%</td>
</tr>
<tr>
<td>bframe</td>
<td>2310804</td>
<td>1693554</td>
<td>36.45%</td>
<td>23.96%</td>
</tr>
<tr>
<td>reaps</td>
<td>2010640</td>
<td>1693554</td>
<td>18.72%</td>
<td>17.50%</td>
</tr>
<tr>
<td>regis1</td>
<td>3148824</td>
<td>1466063</td>
<td>114.78%</td>
<td>0.73%</td>
</tr>
<tr>
<td>regis2</td>
<td>3149328</td>
<td>1482624</td>
<td>112.42%</td>
<td>1.51%</td>
</tr>
<tr>
<td>sss</td>
<td>2259628</td>
<td>1693554</td>
<td>33.43%</td>
<td>67.62%</td>
</tr>
<tr>
<td>tlsf</td>
<td>2305224</td>
<td>1693554</td>
<td>36.12%</td>
<td>63.06%</td>
</tr>
</tbody>
</table>

6.9 Fragmentation results analysis and evaluation

By looking at the tables, there is no one allocator that clearly stands out having the lowest fragmentation. The problem of fragmentation is complex, and the results clearly demonstrate this. We need to take account the unique allocation behavior of each trace in order make good conclusions.

The boot and stable trace measurements reflect the allocator behavior under normal circumstances. The synthetic uniform and small traces however are only useful to support analysis. The measurements from uniform trace show how allocators behave when all block sizes are allocated with the same probability, whereas small trace shows how fragmentation is affected when majority of the allocations are small. By comparing the fragmentation in small and uniform traces we observe that small block allocation increases fragmentation and overhead in all cases except the reaps allocator.
The *boot* and *stable* traces allocate a low amount of live memory, below 17 KB, whereas *uniform* and *small* traces allocate above 1447 KB. This impacts fragmentation by *reaps*, *regis1*, and *regis2*, since these allocators use pages that are quite large in size. The new page allocations are clearly visible as steps in figures 24 and 25. The *sss* and *tlsf* do not use pages, and hence they are shown to have much more subtle and gradual heap growth.

Also having separate traces for boot and stable phases impacts our fragmentation measurements. The *boot* and *stable* traces have considerably different allocation behavior. The *boot* trace allocates many blocks with extremely long lifetimes (some live until the system is shut down), while *stable* trace mainly allocates blocks with short lifetimes. We believe this increased measured fragmentation of implementations that used paging (*bframe*, *reaps*, *regis1*, *regis2*) and especially *bframe*, since it leads to lower utilization of page memory. We believe it is crucial to use traces that contain all relevant allocation behavior, and not only to focus on the stable phase as suggested by Masmano and others (2008a, p. 175).

The *bbuddy* will theoretically have roughly 100% worst-case internal fragmentation, but we see more fragmentation due to large amount of overhead from block headers. The fragmentation and overhead is clearly worse when small blocks are allocated in *small* and *boot* traces.

The *bframe* allocator seems to have a constant maximum heap size and implementation overhead in the figures. The reason for this is that the *boot* and *stable* traces allocate too little memory, and hence no new pages are allocated. This causes higher fragmentation in the allocator because the currently allocated pages are not fully utilized. Implementation overhead is embedded in the pages, and since no new pages are allocated, overhead remains constant. For these reasons, *bframe* fragmentation is high in smaller *boot* and *stable* traces, and less in larger *uniform* and *small* traces.

Similar to *bframe*, *reaps* fragmentation is also quite high in *boot* and *stable* traces compared to *uniform* and *small* traces. As stated earlier, this is because *reaps* allocates memory in pages which causes higher fragmentation when allocated live memory is small. The *reaps* has a low relative overhead which seems to be under 20% even in the *small* trace. The *small* trace displays the low fragmentation that is possible when large numbers of small allocations are being made.

The *regis1* and *regis2* allocators have clearly the highest internal and overall fragmentation. The main cause is the region mechanism, which prevents the region from being freed unless all blocks in the region are also freed. Since the freed blocks cannot be reused by the allocator before the whole region is freed, large amount of practically free memory is unusable (internal fragmentation). The implementation overhead is however clearly the smallest.

The *sss* allocator has the most unpredictable behavior in terms of fragmentation. The lowest fragmentation was 4.96% while the highest was 351.99%. Great majority of fragmentation is external, and is probably caused by the inability to reuse the memory since no coalescing and splitting is done. We believe *sss* may have very high worst-case fragmentation.

The *tlsf* allocator has the lowest fragmentation. This is clear from the real trace measurements, and even with synthetic traces *tlsf* performs very well. Our results
confirms the claim by the authors that \textit{tlsf} has roughly 3% internal fragmentation. The allocator has however the highest ratio of implementation overhead to fragmentation.

For all allocators except \textit{regis1} and \textit{regis2}, the majority of wasted memory is from other sources than internal fragmentation and implementation overhead. Almost all allocators have trouble in effectively reusing the previously allocated memory. The best allocators in terms of fragmentation seem to be \textit{tlsf}, \textit{bbuddy} and \textit{reaps}, while \textit{reaps} seems to be the best allocator when large numbers of small blocks are allocated.

We find \textit{sss} too unstable to be used for real-time DMA. Also \textit{regis1} and \textit{regis2} exhibit such a high internal fragmentation that they cannot be used for real-time DMA. Other allocators seem to function quite well in terms of fragmentation. The \textit{reaps} and \textit{bf} are more usable when many small blocks are allocated and the heap size is quite large. The \textit{bbuddy} and \textit{tlsf} are both suitable for smaller heap sizes, and have stable behavior in terms of fragmentation. The \textit{bbuddy} however has more higher fragmentation compared to \textit{tlsf}, especially when allocating small blocks. The \textit{reaps} allocator has overall lowest fragmentation in small block allocation.

### 6.10 Evaluation

In previous sections we analyzed and evaluated the implementation worst-case timing and fragmentation aspects separately. We will now analyze and evaluate the implementations in both aspects in order to provide information on the space-time tradeoffs and suitability for real-time DMA.

The measurements with \textit{sss} show that while SSS mechanism is very fast, it should not be used in real-time DMA. If the mechanism is used, a great care needs to be taken to prevent the fragmentation from accumulating. If deferred coalescing is used with SSS, the coalescing needs to be implemented so that it does not interfere with the real-time system scheduling.

The measurements from \textit{regis1} and \textit{regis2} implementations show that region allocation mechanisms should not be used used in real-time DMA. While region allocators are fast, their worst-case fragmentation is also extremely high. The \textit{reaps} implementation however shows that, while reaps have slightly higher WCET, their worst-case fragmentation is much smaller than regions to justify the use. We believe that reaps should be always used instead of regions. Additionally reaps has very low fragmentation when small blocks are allocated.

The measurements from \textit{bf} show that Bitframe allocator is suitable for real-time DMA. The allocator has low and bounded WCET and predictable worst-case fragmentation. This shows that bitmap allocators in general can be used effectively for allocation. The fragmentation is however quite high even when a large number of small blocks is allocated, and reaps mechanism appears to be more effective. The Bitframe allocator would benefit from a situation where a large number of small blocks are allocated and stay allocated for a long duration. This is however unlikely to be common with small blocks, since their lifetime is usually short. In some cases however, such as dynamic language virtual machines, lifetimes of the blocks may be very unpredictable and long.

We observed that \textit{bbuddy} exhibits surprisingly low WCET, and that its worst-case fragmentation is not terrible, but however quite high. Our results are similar to the
results from previous research concerning binary buddies. The measurements also show that binary buddies have higher WCET and fragmentation than TLSF in all cases.

The results from *tlsf* implementation show that TLSF allocator has extremely good properties for real-time DMA. The allocator has good overall WCET; lower than binary buddies, but higher than rapid region and reap allocators. TLSF has additionally the most predictable worst-case fragmentation compared to other allocators in the experiment. Additionally TLSF has low fragmentation even when small blocks are allocated. We believe this is because TLSF has a strategy to reuse previously allocated memory, whereas the other mechanisms overly focus on reducing execution time and implementation overhead.
7. Conclusion

This study has now thoroughly approached the research topic concerning DMA mechanisms for small block allocation in real-time embedded systems. We answered our first research question concerning the suitability of DMA mechanisms for real-time embedded DMA in chapter 3 where we conducted a literature survey and analysis on DMA mechanisms. We then answered the second research question concerning DMA mechanism suitability for small block allocation in chapter 4 where we analyzed various well-known general-purpose allocators and their source code.

To answer the third and fourth research questions, we implemented a set of allocation mechanisms for experimentation based on the results from chapters 3 and 4. We performed simulations and measurements on the implementations by using real and synthetic traces, and determined the WCET and estimated worst-case fragmentation of the allocation mechanisms. Finally we presented analysis on the suitability of the mechanisms for small block allocation in real-time embedded systems. The simulation experimentation, evaluation and analysis are described in chapter 6.

Based on our findings, we conclude that reaps mechanism has low WCET and fragmentation when small blocks are allocated, and we recommend reaps for small block allocation in real-time embedded systems. We are also confident that reaps mechanism should be used almost universally in place of the region mechanism. Reaps are shown to have lower fragmentation and only slightly higher execution time. Our findings support the earlier findings concerning reaps.

Our findings also support earlier research concerning the TLSF allocator. We are confident that TLSF is a fast and reliable general-purpose allocator for real-time DMA. The efficiency of the simulated TLSF, binary buddy and Bitframe implementations also show that bitmapped indexing is useful mechanism for many DMA. Our measurements also show that SSS mechanism has unpredictable worst-case fragmentation. We discourage the use of SSS in real-time DMA.

We have additionally introduced Bitframe allocator, a novel bitmapped fits allocator, and demonstrated that bitmapped fits can be used effectively both in terms of time and storage costs. While Bitframe allocator and bitmapped fits do not seem to offer gains compared to other allocation mechanisms, we believe that more research is needed to determine this. Additionally, the strengths and weaknesses of bitmapped fits should be more thoroughly investigated.

Concerning DMA trace simulation methodology, we believe it is crucial to use traces that encompass relevant DMA use cases, including the boot phase of the real-time system. We disagree with Masmano and others (2008a, p. 175) who state that experimentation should focus in the stable phase of the real-time system.

There was some interference in our timing measurements which we believe was primarily caused by the CPU cache operation. We emphasize that care needs to be taken when timing measurements are performed with modern CPUs.
References


Steele, G., Jr. (1977). Data representations in PDP-10 MACLISP. *MIT AI Memo, 421.* Available online: http://hdl.handle.net/1721.1/6278


