

Dynamic Allocation for Embedded Heterogenous Memory

An experimental study

Thomas Peterson

**KTH ROYAL INSTITUTE OF TECHNOLOGY  
*INFORMATION AND COMMUNICATION TECHNOLOGY***

Abstract

Write an abstract. Introduce the subject area for the project and describe the problems that are solved and described in the thesis. Present how the problems have been solved, methods used and present results for the project.

The presentation of the results should be the main part of the abstract. Use about ½ A4-page.

**Keywords**

**Memory management, NVRAM, Scratchpad memory, Embedded systems**

Abstract

Svensk version av abstract – samma titel på svenska som på engelska.

**Nyckelord**

**Minneshantering, NVRAM, Scratchpad minne, Inbyggda system**

Table of Contents

[List of Tables i](#_Toc501395221)

[List of Figures ii](#_Toc501395222)

[Abbreviations 1](#_Toc501395223)

[Glossary 2](#_Toc501395224)

[1 Introduction 3](#_Toc501395225)

[1.1 Background 3](#_Toc501395226)

[1.1.1 Evaderis and Dycton 3](#_Toc501395227)

[1.1.2 Problem Background 4](#_Toc501395228)

[1.2 Problem 5](#_Toc501395229)

[1.3 Purpose 5](#_Toc501395230)

[1.4 Goal 5](#_Toc501395231)

[1.5 Methodology 6](#_Toc501395232)

[1.6 Delimitations 6](#_Toc501395233)

[1.7 Outline 7](#_Toc501395234)

[2 Background 8](#_Toc501395235)

[2.1 Embedded Systems 8](#_Toc501395236)

[2.2 NVRAM 8](#_Toc501395237)

[2.3 Checkpointing 9](#_Toc501395238)

[2.4 Memory Management 9](#_Toc501395239)

[2.5 Dynamic Data Structures 10](#_Toc501395240)

[2.5.1 Free Lists 10](#_Toc501395241)

[2.5.2 Bitmaps 11](#_Toc501395242)

[2.5.3 Buddy Systems 11](#_Toc501395243)

[2.6 Profiling 12](#_Toc501395244)

[2.7 Plateaus, Ramps and Peaks 13](#_Toc501395245)

[2.8 SystemC 14](#_Toc501395246)

[3 Methodologies 16](#_Toc501395247)

[3.1 Approach 16](#_Toc501395248)

[3.2 Phases 16](#_Toc501395249)

[3.3 Evaluation criteria 17](#_Toc501395250)

[3.3.1 Execution Time Evaluation 17](#_Toc501395251)

[3.3.2 Memory Usage Evaluation 18](#_Toc501395252)

[3.4 Data Structure Implementation 18](#_Toc501395253)

[3.5 Synthetic profiling 19](#_Toc501395254)

[3.6 Simulating Real Program Behavior 20](#_Toc501395255)

[3.7 Memory Manager 21](#_Toc501395256)

[3.7.1 Memory Architectures 21](#_Toc501395257)

[3.7.2 Strategies 22](#_Toc501395258)

[4 Results 24](#_Toc501395259)

[4.1 Profiling 24](#_Toc501395260)

[4.1.1 Plateau 25](#_Toc501395261)

[4.1.2 Ramp 26](#_Toc501395262)

[4.1.3 Peak 27](#_Toc501395263)

[4.1.4 Application 28](#_Toc501395264)

[4.1.5 Conclusions 30](#_Toc501395265)

[4.2 Memory Manager 31](#_Toc501395266)

[4.2.1 Architecture A 32](#_Toc501395267)

[4.2.2 Architecture B 33](#_Toc501395268)

[4.2.3 Architecture C 34](#_Toc501395269)

[4.2.4 Conclusions 35](#_Toc501395270)

[5 Conclusions and Future Work 36](#_Toc501395271)

[5.1 Conclusion 36](#_Toc501395272)

[5.2 Future Work 36](#_Toc501395273)

[References 37](#_Toc501395274)

[Appendix A 38](#_Toc501395275)

# List of Tables

[Table 1: Different read and write latencies for different memory types. 13](#_Toc501190782)

[Table 2: Results from synthetic and application profiling 23](#_Toc501190783)

[Table 3: Results from memory manager’s different strategies ran on 3 different architectures 30](#_Toc501190784)

# List of Figures

[Figure 1: Synthetically generated memory allocation patterns illustrated. The left most graph illustrates a plateau, the middle graph a ramp and the right most graph a peak. 13](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc501356781)

[Figure 2: The language architecture of SystemC 14](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc501356782)

[Figure 3: The heap and its elements illustrated. 18](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc501356783)

[Figure 4: The execution time of the free list, bitmap and buddy system when combined with the plateau, ramp, peaks and application. 29](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc501356784)

[Figure 5: The memory usage of the free list, bitmap and buddy system when combined with the plateau, ramp, peaks and application 29](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc501356785)

# Abbreviations

**INSA:** Institute National Des Sciences Appliquées

**JSON:** JavaScript Object Notation

**LFB:** Largest free block

**LMF:** Largest Memory Footprint

**NVRAM**: Non-Volatile Random Access Memory

**RAM:** Random Access Memory

**SOC:** System on Chip

# Glossary

**Allocator:** See Memory Allocator.

**Application:** The client program interacting with the memory allocator.

**Bank:** See Memory Bank.

**Bit:** A boolean digit with a value of 0 or 1.

**Block:** One contiguous piece of memory used by the allocator to store data.

**Byte:** 8 bits.

**Data structure:** A format for structuring blocks in memory. In this paper a free list, buddy system or bitmap.

**Dycton Project**: The project aiming at creating the software for Evaderis new architectures.

**Evaderis:** The company working with NVRAM located in Grenoble.

**Fragmentation:** The sum of the internal and external fragmentation.

**Grain:** The smallest amount of memory an allocator can work with.

**Heap:** A part of memory reserved for dynamic data.

**JSON:** A data storage format commonly used for communication on the web.

**JSON Parser:** A piece of software converting JSON data to objects.

**Memory Allocator:** A piece of software responsible for deciding where in a memory bank to put an object. Uses a data structure to structure memory.

**Memory Bank:** A portion of memory with a certain type of RAM.

**Memory footprint**: The portion of the heap in use by the memory allocator.

**Metadata:** The data used by the data structure to keep track of blocks.

**NVRAM:** Byte addressable memory retaining its content in absence of power supply.

**Object:** One contiguous piece of memory used by the application to hold data.

**RAM:** Byte addressable memory normally used for volatile storage.

Real Profiling: The act of profiling a real world application.

**Spintec:** The research laboratory which founded Evaderis.

**Synthetic Profiling:** The act of profiling a synthetic program designed to exhibit certain desired characteristics.

**Word:** 4 bytes.

# Introduction

With the evolution of technologies and the increasing demand for technical products, more and more computing devices are produced and deployed. In our everyday life, we may stumble across several different embedded systems such as air conditioning systems, automotive devices as well as automatic teller machines without reflecting to much about the complexity of the intrinsic systems. The architectures of these embedded systems must be designed, manufactured, programmed and finally deployed. Their software must then be capable of surviving with the limited hardware under various constrains. As time passes and technology evolves, the devices become outdated. The legacy code is not always able to run on newer hardware and thus additional software must be developed.

With newly introduced architecture designs, the corresponding software must preferably be able to take advantage of various hardware properties. This is the case for most of the system’s subunits. One of those subsystems is the memory which is primarily managed by a memory allocator. Furthermore, different allocator strategies exist, helping the allocator in its task of optimizing the memory usage. These strategies should be formed with the memory properties in mind. This thesis attempts to take a step towards solving these strategic memory allocation issues by trying to identify different trade-offs existing in a simulated embedded system with different memory properties.

This chapter introduces the thesis, in it’s whole, and gives an overview over the different aspects implicitly associated with the thesis-related work. The second chapter gives an in-depth background and the third presents the different methods used in the degree project as well as in the thesis report writing. Following these, chapter four describes the degree project and the fifth chapter exposes the results. Finally, in the sixth chapter, conclusions are drawn and thereafter a summary concludes the thesis.

## Background

This section provides a background for understanding the thesis problem. The first subsection of this section explains how the project is linked to the company Evaderis and the Dycton project. The second subsection describes the thesis problem.

### Evaderis and Dycton

Evaderis is a company in Grenoble, a city located in the south-eastern part of France. Evaderis was founded by Spintec in 2014, a research lab which is also located in Grenoble. This research lab specializes in Non-Volatile Random-Access Memory, NVRAM for short. Spintec’s goal is to evaluate and analyze new emerging NVRAM Technologies. They do not only publish research papers concerning the subject, but also contributes to a coherent patents portfolio. Additionally, they create relevant functional demonstrators used for demonstrations of their technologies. [1]

Evaderis is the first company worldwide offering intellectual property solutions based on new embedded NVRAM architectures. Evaderis’ goal is to bring system on chips using NVRAM technologies to the industrial market. By developing their own architectures, they create products with new possibilities and features in terms of cost efficiency, performance, density and endurance. Because of this they are able to compete with their competitors. By implementing these NVRAM technologies the company hopes to be pioneers in a new paradigm regarding architecture design for embedded devices. [2]

The Dycton project is a PHD degree project conducted at INSA-Lyon which addresses the problem of dynamic memory management in novel architectures for NVRAM based systems on chips. It aims at providing an easy-to-use memory manager for embedded software developers working on the Evaderis architecture. As the hardware chip is still being developed and not yet available for study the Dycton project implements a simulator reproducing the main characteristics of the platform. This not only enables parallel software development but also enables architecture exploration. Meaning that simulating different types of memory and study variants of the architecture is a possibility.

The Dycton project’s system on chip hardware is simulated using SystemC. The simulator is keeping track of data related to memory access patterns such as cycles spent freeing and allocating memory from the heap. This data is stored in log files which later undergoes analysis applied by Python scripts. As Python is a high-level programming language it is only used for data analysis and is not a base for the software running on the simulator. Instead the simulator is running software coded in C which is closer to the hardware.

The Dycton project can be divided into three main phases. In the first phase a basic instruction set simulator(ISS) as well as benchmark programs are developed. The basis for the benchmark programs are decided by surveying different research papers and working in close cooperation with Evaderis. In the second phase memory allocationmechanisms are designed. In the third and last phase the simulator becomes more detailed and the different properties of various NVRAMs are comprised. This thesis contributes to the third phase of the Dycton project by evaluating various allocators engaging on memory with different properties. One could postulate that without the Dycton project the thesis problem of this report wouldn’t have been brought to light.

### Problem Background

The company Evaderis desires an embedded device which can operate with low power consumption. The company aims to achieve this by using checkpointing, normally-off technique and non-volatile memory. Furthermore, the device shall be off by default, meaning that it will only be turned on for a short amount of time when triggered by an interrupt. During this short timespan the device is expected to complete all its computations before going back to its normally-off state. Checkpointing is achieved by keeping necessary data in non-volatile memory. More info about normally-off technique and checkpointing can be found in the background chapter on page 8. [3]

A difference between Evaderis proposed architecture compared to the common architectures is the fact that all caches have been substituted for scratchpads. A scratchpad is an extension of memory which can be accessed faster than the ordinary ram. These scratchpads use STT-MRAM which is non-volatile and thus permits checkpointing. These scratchpads will enlarge the memory address space and indeed have different properties compared to DRAM memory. The addition of another memory unit results in a new set of addressable addresses. By using multiple memory allocators with various strategies, one can tune the strategies to optimize memory allocation. Likewise, a possibility to reach the same goal, is to combine different memory types with different allocators which implement different strategies. [3]

## Problem

Legacy software acting on embedded devices will not work well on an architecture consisting of several memory regions with different properties. Thus, new memory allocators must be developed to accommodate for those properties. This problem can generally be summed as the sentence “What is the best strategy of a memory allocator acting on heterogenous memory?”. This problem statement is complex and won’t be the main focus of this report. The main focus of this report will instead be to identify the different trade-offs that must be made when allocating heterogenous memory. The specific problem this thesis addresses can be expressed as “What trade-offs exist concerning the memory allocation of heterogenous memory?”.

## Purpose

The purpose of the thesis is to evaluate different strategies for the memory allocator as well as discovering trade-offs related to the properties of the heterogeneous memory they will be operating on. The purpose of the work is to create a basic memory manager for an embedded architecture where caches have been substituted for non-volatile units termed scratchpads. As already mentioned, this memory manager ought to take memory properties into considerations when making allocation decisions. This memory properties varies with the memory types.

## Goal

The primary goal of the project is for the Dycton Project to have a working bare bones memory allocator which can be adjusted to different memory properties. This will help the Dycton Project achieve its goal of providing a complete runtime system for Evaderis new device architectures. By doing so, future programmers of Evaderis new architectures won’t have to take the memory properties into account when allocating and deallocating memory blocks to the heap.

The project will favor the Dycton project by helping in the process of its completion. Likewise, the company will also benefit by getting closer to the goal of having a working software layer between the hardware and future programmers. This might in turn create new possibilities for low power embedded devices and improve many products. For example, security cameras could run with low power consumption and thus be cheaper to maintain. By limiting the energy consumption of mass produced electronic devices the total energy consumption of society may lower. A lower energy consumption may also facilitate for electronics to reach underdeveloped third world countries where electricity is expensive. This is thus a step in the right direction when talking about sustainable development.

The ethical aspects of the project are more complicated than the general benefits mentioned earlier. When releasing new technology, one cannot know how it will be used or misused by the rest of the world. New, more energy efficient, embedded devices might be used in contexts were ethical aspects may be more or less important. For example, as security cameras becomes cheaper they may be used in illegal and unethical manners. This might be farfetched and since the future is hard to predict, we might as well discuss more interesting things like memory allocators. With that said, ethical issues are not a major concern of this project.

## Methodology

There were three persons involved in the degree project work. A PHD student from INSA-Lyon was the author of the simulator and the primary person to work on the Dycton project. He was responsible for the creation and maintenance of the simulator. Secondly, I was doing the degree work by developing a bare bones memory allocator for the Dycton project. The third person involved was an associate professor at INSA-Lyon acting as a supervisor and product owner. We worked in an agile-like manner with weekly meetings but with our own different developer roles. The work was executed in an inductive manner by attempting to sample enough data to later draw conclusions.

## Delimitations

The aim of the delimitations of the degree project is not to prevent discoveries. Rather, the delimitations exist to improve focus which in turn permits us to go deeper. Possible delimitations are, for example, which data structures to use. This project work will be delimited to three data structures. These are a Bitmap, a Buddy System and a Free list. These data structures will be matched with different memory types and be evaluated with the help of different evaluation criteria.

The evaluation criteria are access pressure, execution time and heap fragmentation. The criteria are described more in depth in section Evaluation. The data structures will all work with a first fit policy when a policy is relevant. The memory properties, which the allocators are working with, will be based on properties of NVRAM memories.The memory types we are particularly interested in, and thus limit our research to, are EEPROM, NOR Flash, FRAM, MRAM, PCM and RRAM. For the interested reader, these memory types will be further explained in chapter 2.

## Outline

In chapter 2 a theoretical background is presented. In chapter 3 the degree project is described and in chapter 4 the results are shown. Thereafter, conclusions are drawn and discussions are made regarding further possible work in chapter 5 .

# 

# Background

This chapter contains a theoretical background. The first section describes different types of profiling and their pros and cons. The second section introduces the dynamic data structures used in the project. The third section describes different characteristics of program behaviour and the forth section provides an overview over the different types of non-volatile memory types which could be used in Evaderis architectures. The following last two sections are not required to understand the main points of the thesis but are provided for the interested reader.

## Embedded Systems

What is an embedded system

No user interface

Little memory space

Energy constrains

(Inspiration from Wikipedia)

No virtual memory

No cache

## NVRAM

Non-volatile random-access memory(NVRAM) are the type of RAM with the property of being non-volatile. The non-volatility of memory significates that the memory content is kept even when the memory is no longer provided with power. There are many different types of NVRAM currently in existence. Some include EEPROM, NOR Flash, FRAM, MRAM, PCM and RRAM. Different read and write latencies for these memory types can be found in Table 1.

Charge storage is a technique used in EEPROM, NOR Flash and FRAM. The term charge storage refers to these memories’ way of storing information by storing a charge. This charge represents a bit state that can be read later when information retrieval is required.

EEPROM stands for electrically erasable programmable read-only memory and consists of arrays of field and control gates. Field gates contain different charges while control gates regulate which field gates to program. EEPROM can be programmed using field electron emission (Also known as Fowler–Nordheim tunneling). Field electron emission can either inject or remove electrons from field gates. This abundance or lack of charge significates a bit state of 1 or 0.

http://www.freepatentsonline.com/4115914.pdf

NOR Flash is a type of flash memory working with arrays of MOSFET transistors similarly to EEPROM and regular flash memory but using nor gates. These nor gates are instantiated by connecting the cells’ floating gate’s legs to ground and a bit line. NOR Flash is known for having a low read latency but is, on the other hand, not dense compared to other memory types.

https://www.eetimes.com/document.asp?doc\_id=1278751

Ferromagnetic RAM FRAM

Resistance Switching is a technique used in MRAM, PCM and RRAM. The basic idea behind the technique is to store information by transforming the state of a dielectric solid-state material. The state of the dielectric is defined by its conductivity which can be changed by applying different voltages over the dielectric material. Since the conductivity of the dialectic material is dependent on past events of current flow, it is sometimes referred to as a memristor.

MRAM

PCM

RRAM

Describe memory types

We descrbed principels of these technology

Read faster than wrtinging of FRAM,MRAM,PCM,RRAM.(See magnus master thesis) Read mail from 31:st October

NVRAMs are more energy costly than normal ram and slower.

SRAM used in 1:1 memory banks since companies could use this RAM-type.

## Checkpointing

Checkpointing is a technique used in embedded devices for saving energy. By not always being turned on, the device applies, what is called a normally off technique. The idea behind this technique is that the device is turned off until it’s required to do an action. When time comes for the device to act, it wakes up, executes for a short period of time and goes back to its off state. Checkpointing refers to the idea of saving the device’s current state to enable it to continue from that checkpoint in the future. [3]

When a normally off device stops executing it saves all of its register values to a non-volatile memory. If the device is operating on volatile RAM it will also need to save its RAM-data to some other memory location containing non-volatile memory, such as the hard drive. Following these memory adjustments, the device activates a timer before finally turning off. This timer is set to a certain amount of time. When this timer finishes an interrupt is sent to the kernel which turns the device back on. As soon as the device is powered on it runs its usual start up routines followed by the device specific code. After the device has stopped executing, it goes back to the first step of the checkpointing cycle to start a new checkpointing iteration. [3]

## Memory Management

Describe binary sections

Static data, stack, heap,text

Memory management is the act of deciding where in memory to put data giving address to each object.

Define memory block

Define interface of malloc and free

Fragmentation

Data structures

What is the job of a memory allocator?

## Dynamic Data Structures

Memory allocators use different data structures to keep track of memory. The goal of the allocators is to allocate memory while minimizing wasted space without affecting performance significantly. This chapter presents different dynamic data structures commonly used in memory allocators [4]. In section 2.5.1, free lists are presented, thereafter bitmaps are introduced in section 2.5.2. Finally buddy systems are explained in section 2.5.3.

### Free Lists

A free list is a linked list keeping track of all the free blocks in the heap. In a free list each allocated and unallocated memory block is given a header. This header contains the necessary data for the free list to operate. The header of an allocated block contains information about the allocated block size. The header of a free block contains both information about the free block’s size as well as a pointer to the next free block. For allocated blocks, the header is instead only used when freeing blocks. The memory allocator code can thus limit itself to one pointer to the free list head to keep track of the heap status.

To allocate a block the memory allocator accesses the head of the list using its head pointer. It then reads the free list and makes decisions based on a policy. Depending on the policy used the memory allocator might have to read the whole free list or only a couple of nodes before allocating a memory block. Once a block has been successfully allocated, the memory allocator returns a pointer pointing to the start address of the block to the caller.

When freeing in a free list the size of the memory block to be freed can be found in the header of the allocated block. Using the pointer acquired from the function call, the memory allocator is able to calculate the starting address of the block’s header. It can then extract the block size and use this information to add a new free block into the free list.

To keep the free list short, coalescing is normally attempted after freeing. After freeing a block, the memory allocator checks if nearby blocks are adjacent. If so, the freed block can be coalesced with its free neighbor into a bigger free block. This allows two free list nodes to combine into one which in turn results in quicker access times for the nodes behind this node. Additionally, it enables allocation of larger blocks since they might fit into the newly coalesced block.

For free lists there exists many different policies for choosing which free blocks to allocate and return. Commonly used policies are first fit, best fit and next fit. First fit traverses the list returning the first free block large enough for the requested size. The best fit policy finds and returns the smallest available memory block large enough for the requested size. The next fit policy usually operates in the same manner except that it moves its head pointer along the free list.

### Bitmaps

Bitmap implementations keeps track of memory se of a specified memory range using two bits per segment. The state of the bits provides information about whether its corresponding segment is occupied or not and whether the bit is a boundary bit or not. The size of a segment may vary depending on the implementation. This segment size affects the speed of memory allocation and deallocation and should thus be chosen carefully. The segment mapping bitmap is stored in its own memory area and maps to a memory range outside this area.

When an allocation request is received the allocator starts the allocation process by calculating the number of segments needed to satisfy the request. It then proceeds by searching for a continuous amount of non-occupied segments using the bitmap. When enough continuous non-occupied segments are found, they are marked as occupied. Additionally, the last segment is marked as a boundary segment. Finally, the address of the first segment is returned to the requestor. The bitmap search time is proportional to the number of grains in the bitmap which in turn is dependent on the bitmap’s grain size.

Freeing in a bitmap implementation is of trivial nature. Firstly, the given address is converted to a bit location in the bitmap. This bit location is set as the current bit location. This current bit location is then marked as non-occupied and the allocator checks if the bit location marks the segment as a boundary. If so, the memory allocator stops its freeing procedure. Else way the current location is set to the subsequent bit location and the allocator continuous to check the bit map recursively.

### Buddy Systems

A buddy system works using a tree like structure. The data structure keeps track of the availability of the tree nodes using a bitmap. Each node corresponds to one block in the memory and one bit in the bitmap. The size of a block is dependant of its deepness in the tree. The root node has a size equal to the heap size and its children have sizes equal to half of the parent size. This data structure only allocates blocks with sizes equal to powers of 2 since the heap is a powers of 2 and each tree node has two children.

The allocator starts off with one big block representing the whole heap size and its current level set to the highest level. When receiving a memory request, it checks if the size of the current tree-level matches. If the size matches the address of a free block on that level is returned. If not, the allocator checks if the size matches the size of the next level by setting the current level to the next level. If the current level reaches the deepest level of the tree it automatically stops searching for a perfect match and returns an address to a free block in the deepest level.

When the allocator can’t find a free block in the matching level splitting occurs. Splitting is the act of recursively moving up the tree trying to split free blocks higher up to enable the allocator to find a free block on the desired level. When splitting the allocator sets the parent node’s corresponding bit to 1 meaning allocated. It then proceeds to set its two children nodes’ corresponding bits to 0, meaning free. Finally, the allocator retries allocation on the desired level. If this fails, the allocator may try merging smaller free blocks into a sufficiently large block.

Freeing is considered to be fairly straight forward in the buddy system as it is in many cases sufficient to only set a bitmap bit to 0. However, in some cases, coalescing is required. This depends on if the freed bit’s, so-called buddy, is also free. The buddy is defined as the bit that has the same parent bit in the bitmap as the freed bit. If both bits are free they may be coalesced. The act of coalescing is the act of marking both children nodes as allocated and the parent node as free. Coalescing allows the buddy system to handle large requests after small requests have been allocated and freed.

Since there is usually no space between allocated blocks in a buddy system, this data structure does only suffer minimally from external fragmentation. Instead, blocks risk being too large for the requested size and internal fragmentation may thus instead be common. It is also common that the allocator is rather fast compared to other allocators but instead suffers from fragmentation issues.

## Profiling

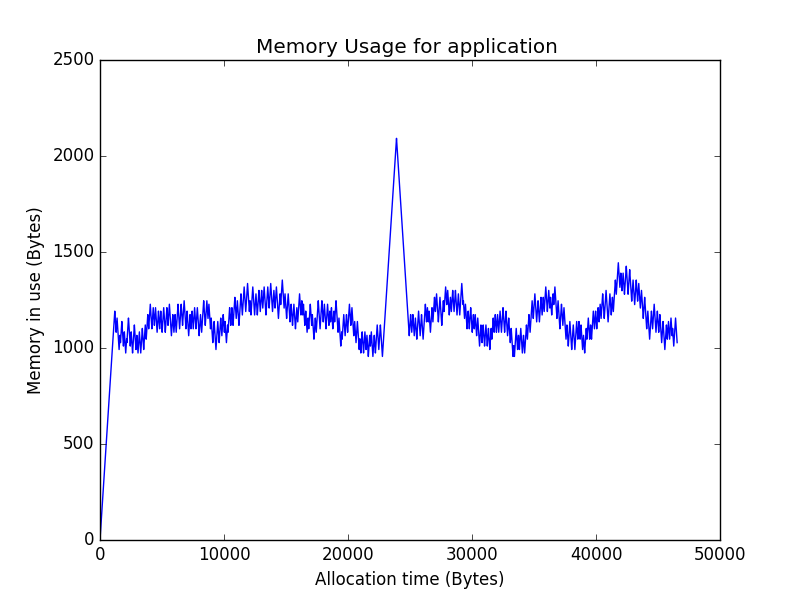
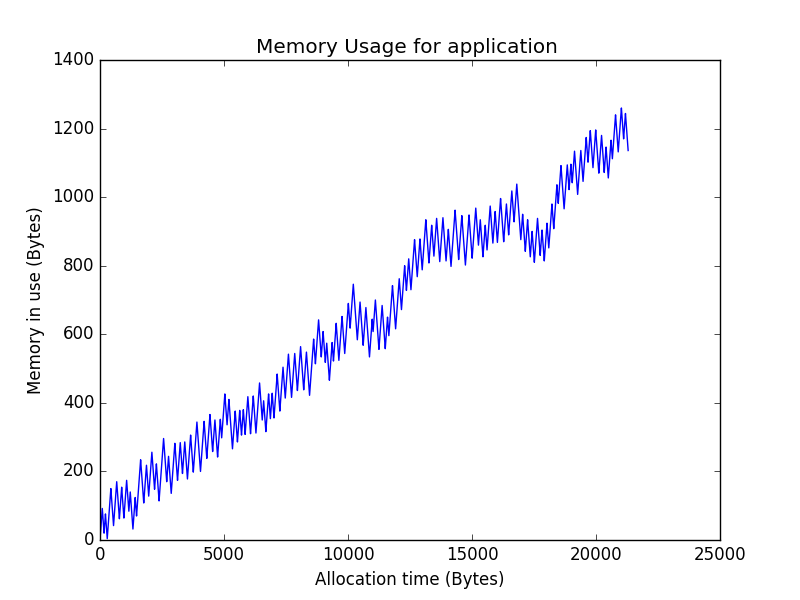
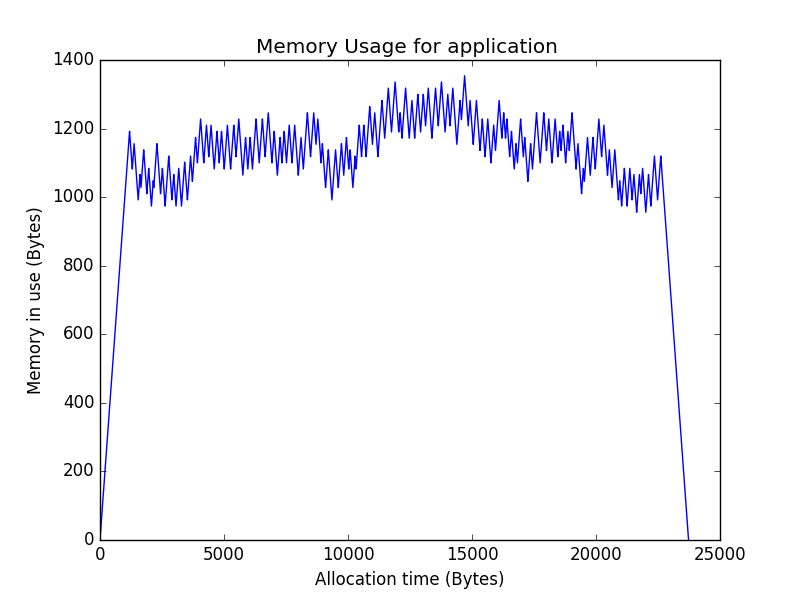
When evaluating memory allocators there are two common practices for profiling. The first one is to draw conclusion using mathematics. This has been shown to be insufficiently accurate for general memory allocation and thus the second technique is more commonly used. The second technique is referred to as profiling. Profiling is done by tracking the behavior program and tracing its memory allocation and deallocation while keeping track of the status of the heap data structure. Profiling can be applied on either real or synthetic applications. In this report we refer to profiling of real world applications as real profiling and profiling of stochastic applications as synthetic profiling.[4]

In stochastic profiling memory allocations, deallocations, block lifetimes and block sizes are randomized using stochastic distributions. Blocks are then allocated and unallocated pseudo randomly. An advantage of stochastic profiling is that stochastic allocation most likely won’t randomly exploit advantages in the memory allocator being evaluated. And thus, won’t randomly yield good but incorrect results. This is something which is more likely to happen with real profiling since the allocator may perform very well with the profiling program but miserably when used by real programs in practice. [4]

Figure 1: Synthetically generated memory allocation patterns illustrated. The left most graph illustrates a plateau, the middle graph a ramp and the right most graph a peak.

In real profiling a program is chosen with the purpose of mimicking the behaviors of larger real programs. By verifying that the allocator can handle the stress from the chosen program one expects its strategies and policies to generally work on a larger scale, with other programs. Since real profiling is using an actual program for profiling rather than a synthesized model, it is often seen as a rather good way of profiling. The difficulty with real profiling is to acquire a real application representative of other programs. Many questions arise about what characteristics real programs express and how one could exploit them strategically. As may be expected, these questions are hard to answer since there are no absolute characteristics expressed by all types of programs. [4]

In this degree project both stochastic and real profiling, with some stochastic behaviors, were chosen for evaluating different memory allocators. Synthetic profiling was used to study the characteristics of various memory allocators while real profiling was used to validate the behaviors discovered.



For real profiling, a small program parsing and allocating JSON objects was taken of the shelf rather than synthesized to represent real programs as well as possible. The idea behind using a JSON parser was that JSON objects, to some extent, should reflect the size of commonly used objects by real programs. These JSON objects were then chosen carefully to imitate real program behaviors as well as possible. The JSON Objects fetched were not synthetized but instead fetched from real life applications.

## Plateaus, Ramps and Peaks

The typical heap allocation patterns can be observed as combinations of three different fundamental allocation patterns. These patterns are plateaus, ramps and peaks. The names are derived from the form being observed when program behavior is plotted in a graph illustrating memory in use plotted against time. In figure x these fundamental allocation patterns are illustrated. In this thesis the allocation patterns will be referred to as access profiles.

A plateau is characterized by long lived memory allocations. When a program is initializing it is common that it allocates a distinguishable amount of data to the heap. This data is often big data structures. This data is then only deallocated when the program stops. This behavior is what we call a plateau. An example of a synthetic plateau can be seen in Figure 1.a.

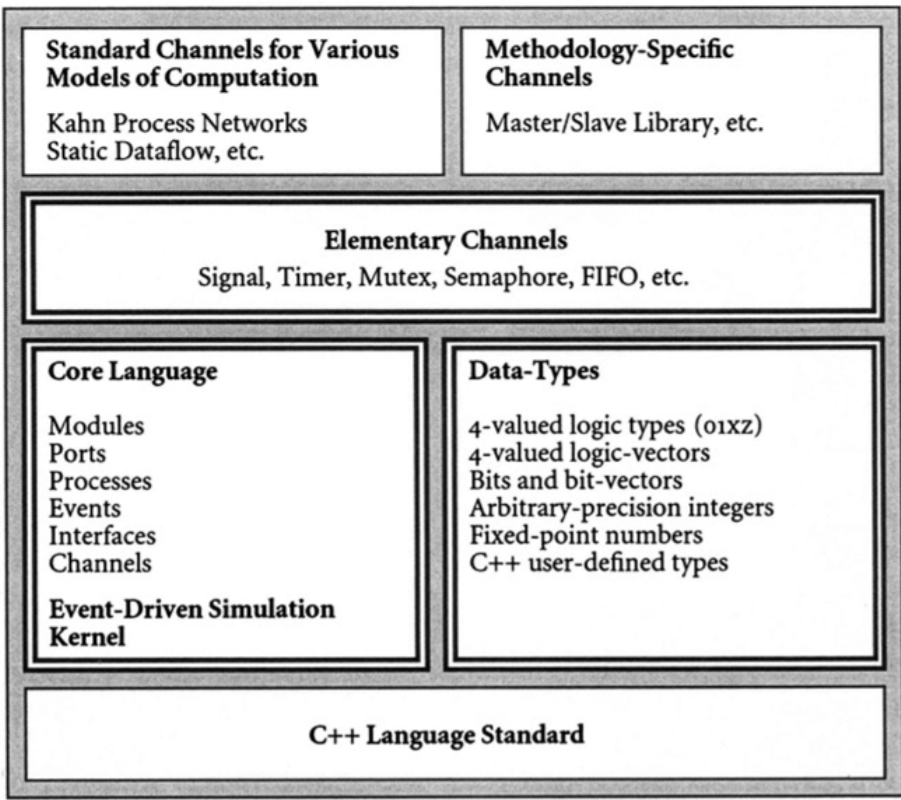
A ramp is characterized by an increase in memory usage over a program’s lifetime. This pattern is distinguishable in programs that allocate more and more data as time elapses. An example of a synthetic ramp can be seen in Figure 1.b.

Figure 2: The language architecture of SystemC

Peaks is characterized by a steep and short-lived increase in memory usage. This pattern can be observed in programs requiring a lot of memory for doing something temporarily. To study peaks is of special interest when studying memory allocation since the memory allocator is typically extra stressed when influenced by this type of patterns. An example of a synthetic ramp can be seen in Figure 1.c.

## SystemC

SystemC is a discrete event-driven simulation interface. A primary goal of the interface is to enable system-level modeling. It is used for modelling both hardware and software over the register-transfer abstraction layer. The models’ specifications are described using plain C++ syntax which increases the ease of use of the interface compared to interfaces using their own languages. Interestingly, since the fundamental base of SystemC is created in standard C++, SystemC programs can be complied into executables using a standard C++ compiler. [6]

There are various challenges regarding the creation of a system-level design language. Firstly, one must be able to model the wide range of computation and communication models in existence. Additionally, one must also be able to model the different abstraction levels and system design methodologies used in systems as well as doing this efficiently enough to give larger systems enough support. SystemC approaches these challenges using a layered architecture. [6]

The SystemC layered language architecture can be found in figure ??. The base layer of SystemC’s language architecture contains the simulation kernel built on top of C++. This kernel serves as the foundation for higher level structures. By combining the kernel with modules, ports, processes, events, interfaces and channels we form something called the core language. This core language is then combined with different data-types. Together they form the foundation for what we call the Elementary Channels containing signals, timers mutexes, semaphores, FIFO buffers and so on. These Elementary channels then, in turn, form the basis for further channels of higher complexity. [6]

# Methodologies

This chapter describes the degree work and its strategies for attacking the thesis problem. It contains a brief description of our way of working and a description of the implementation and the structure of the memory manager. It also describes our attempts at simulating real program behavior.

## Approach

We addressed the thesis problem with the help of a simulator based on SystemC. This simulator simulated a system on chip architecture with different memory properties. The simulator ran C code and would output logs during the execution of the code.

These log files contained information about heap requests, the most thesis relevant log revolving around malloc and free calls. The malloc logs included information about the requested block size, the block size returned to the caller, start address of the returned block, time of the request and cycles spent in the malloc routine. The free logs included information regarding the size of the block being freed, its address, a timestamp and the amount of cycles spent in the free routine.

The logs and data acquired from the simulations were analysed using Python scripts running outside of the simulation. These scripts were able to plot the heap memory usage over time as well as calculating the total time spent in and outside of various functions. Additionally, the scripts could calculate the memory usage in terms of total fragmentation as well as worst memory footprint. This different metrics are explained in section 3.3.

## Phases

The study conducted can be divided in five different phases. These phases all have dissimilar goals and contribute to the project in various manners. Each consecutive phase builds and depends on the previous phase while simultaneously creating a foundation for the next phase. These phases are presented in figure x.

The first phase was the literature study. In this phase literature relevant to the thesis problem was studied. Some of the relevant literature was provided by INSA’s associate professor and others were acquired using databases such as Scopus and libraries such as ACM digital library.

The second phase was the data structure implementation phase. During this phase the three data structures were implemented. The free list was implemented as an address ordered linked list with a first fit policy. The bitmap was implemented by placing two bitmaps in the start of the heap and a grainsize of 256 bytes. Likewise, the buddy system was implemented using a bitmap, at the start of the heap, and a minimal grainsize of 256 bytes. Implementation details are further detailed in section Data Structures3.4.

The third phase, referred to as synthetic profiling phase, had the goal of implementing and utilizing 3 synthetic profiles for evaluating the 3 data structures’ performances during different types of program behavior. These synthetic profiles reflected the characteristics of a plateau, a ramp and consecutive peaks. These program characteristics are described in chapter x section x.

The forth phase, the real profiling phase, aimed at determining which data structures were the most relevant for a memory manager operating with the common memory and allocation sizes found in SOC:s. To address this issue, a small real program was created. This program was run in combinations with the 3 different data structures implemented in the third phase. By analyzing the results, decisions regarding which data structures to keep and which ones to eliminate could be made.

The fifth and last phase was the strategy phase. During this phase 3 different memory architectures were defined. These architectures were designed with the goal of representing the most common memory properties in the field. Additionally, 3 different memory manager strategies were designed and implemented. These strategies defined various decision based models determining which memory bank should be given which memory request.

## Evaluation criteria

To evaluate the data structures, we used various evaluation criteria. To evaluate execution time, we used cycle counts. For evaluation of system reliability, we focused on fragmentation and what we call memory footprint. Section x presents how we evaluated execution time and section x how we evaluated memory usage.

### Execution Time Evaluation

By using the simulator and its log-files, we could acquire three types of cycle counts. These are malloc cycles, free cycles and outside cycles. By summing these metrics, we acquire an additional metric referred to as the total cycle count which can be used for evaluating total execution performances.

Malloc cycles are defined as the total amount of cycles spent in the malloc routine. In other words, the total amount of cycles spent finding a free block and returning it’s address to the client program, referred to as the “application”.

Free cycles are defined as the total amount of cycles spent inside the free routine. This can be expressed as the total amount of cycles spent freeing blocks and adjusting metadata.

Outside cycles are cycles which were not spent in the malloc or free routine. This type of cycles represents the amount of cycles executed by the application. This metric is of particular interest when discussing compute bound applications. Similarly, malloc and free cycles are of particular interest when discussing memory bound applications.

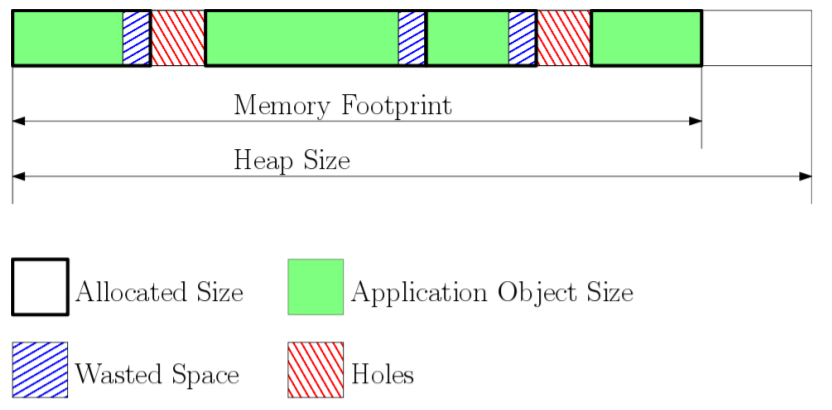


Figure 3: The heap and its elements illustrated.

During the strategy evaluation of our experimental study we used write and read latencies to simulate NVRAM properties. These latencies were included in the form of additional cycle counts when reading and writing to memory.

### Memory Usage Evaluation

The memory footprint corresponds to the size of the heap that is currently in use. It is defined as the highest address in use subtracted by the base of the heap. The memory footprint is illustrated in figure x. Blue boxes and red boxes illustrate internal respective external fragmentation. For multiple heaps we define the total memory footprint as the sum of the individual footprints calculated at the different heaps.

The memory footprint is of special interest since we want to avoid heap overflows which are caused when the memory footprint exceeds the heap size. For evaluation of a data structure’s or allocation strategy’s reliability we used a notion refereed to as Largest Memory Footprint or LMF. This metric is relevant since it is equivalent to the smallest memory required to run an application.

We define fragmentation as the sum of the external and internal fragmentation found inside of the memory footprint. Fragmentation is of interest to study when the LMF is similar for different data structures or allocation strategies. It indicates whether the LMF would likely continue to grow or not.

## Data Structure Implementation

The data structures were implemented in c inside of the simulation environment. They were implemented naively without optimization. Doing this rather than using off the shelf data structures enabled us to have a thorough understanding of their behaviors and decisions.

The free list was implemented in place as a singly linked list ordered by address. It used a first fit policy and aggressive coalescing upon freeing. For free blocks, it kept a header containing a size integer and a next header pointer pointing at the next unallocated block. For taken blocks, the pointer was omitted and the header only contained size information. This header was stored at the top of every block.

The bitmap data structure used two bitmaps. The first bitmap mapping whether a grain was allocated or not while the second bitmap contained information regarding if a grain was a boundary grain. The second bitmap was needed for freeing since the freeing function needed to know when to stop freeing.

The buddy system was implemented with one bitmap mapping different levels containing different grain sizes. Since implementations were made naively it was made to always merge freed blocks when possible.

The naive implementations were thought to be fair but proved to be a catastrophe for the buddy system. The buddy system therefore underwent a change of strategy from aggressive merging and splitting to what is known as deferred coalescing. This simply meant that it would not merge free blocks until it had no other choice.

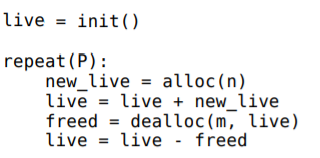
For the strategies sake, the relevant data structures were implemented with logging of their free space. This logging constituted of information about the data structure’s largest free block (LFB) and the remaining free space in its memory bank.

## Synthetic profiling

The different memory allocators were combined with different access profiles enabling a pre-analysis of the data structures’ behaviors. This pre-analysis was conducted with the goal of discovering more of the data structures’ strengths and weaknesses under varying access profiles. Since three synthetic profiles and 3 data structures were used 9 combinations were analyzed.

To synthesize the access profiles, described in section 2.3 we utilized three different block categories. These categories were small sized blocks, medium sized blocks and large sized blocks. Regarding the small and medium category, the block sizes were uniformly distributed in two different size ranges. The block sizes aimed at replicating the block sizes used in practice and were chosen based on the available expertise at INSA.

The small block sizes varied from 12 bytes to 24 bytes representing mainly small local variables used temporally. The medium block sizes varied from 128 to 256 bytes, representing larger allocation such as arrays and data structures. Large blocks were of size 4096 bytes. The large blocks were not used to mimic a certain type of variable size but rather the sum of all the memory requests done during the start of a program. I.e. the steep starting phase of a plateau.

To simulate the access profile a generic function was created. The pseudo code for this generic function can be seen in figure ??. The function used a global array, referred to as live, to keep track of allocated blocks. To generate the access profiles the function invoked the alloc and dealloc functions p times.

When calling the alloc(n) function, n blocks would be allocated having a size stochastically chosen from either the medium or small size category. The returned block would then be stored in the live array. To allocate blocks, the alloc(n) function simply call the malloc function with the desired size.

By calling dealloc(m, live) the repeat function could free blocks. In the dealloc function, m target blocks in the live array would be stochastically chosen. These m blocks would then be deallocated from memory and removed from the live array. The live array would then be reduced in size to eliminate holes created by the absence of the removed blocks. To free a block the dealloc function simply called the free function containing the address of the desired block to free.

To simulate a plateau the preliminary step was to allocate a large sized block. This would create a steep increase in the memory usage as seen in the left most part of Figure 1.a. This allocation was simply done by calling the malloc function with the large block size as the parameter and storing the pointer. After this steep increase, the repeat function was called with p = 1000, n = 10 and m = 10. This would result in the creation a nearly flat memory usage as seen in the middle of Figure 1.a After the allocation and deallocation of these 10000 blocks the large sized block would be de allocated.

The ramp was created similarly to the plateau except that it didn’t use a large block and used the repeat function with different parameters. To generate the ramp, one only had to call the repeat function with the values p = 1000, n = 10 and m = 9. By having n = m + 1 the amount of live objects would increase by 1 after each iteration creating the ramp that can be seen in Figure 1.b. After the p iterations, the ramp would reach it end having a live array of 1000 objects.

The peaks were designed to mimic the intense memory usage commonly seen in programs [4]. They were implemented in the same way as the plateau except for the arguments to the repeat function. After the allocation of the large sized block, the repeat function was instead called with p = 10, n = 1000 and n=1000. This would result in 10 peaks being created before the end of the synthetic plateau. Each peak was created by first allocating 1000 blocks of stochastic sizes and then deallocating these blocks in a stochastic order.

## Simulating Real Program Behavior

To simulate real program behavior rather than using synthetical profiles we chose to implement a JSON Parser and a small C application. The idea was to parse JSON objects using a third-party library. This third-party library synthesizes a JSON tree out of a JSON object. During the creation and modification of the JSON tree the third-party library executed several memory allocation requests and memory freeing calls. This third-party library, in combination with a JSON object, should be a satisfactory replication of real program behavior.

Our approach to simulating real program behavior can be divided into three different phases. The first phase is the parsing phase where the tree is created from the JSON object. The second phase is the selection phase where the C application modifies the JSON tree. During the last phase the tree is serialized to, in theory, be sent back to the original sender of the JSON object.

During all these three phases, all of the tree manipulation is handled by the third-party code. This is favorable since this code’s memory requests will be of realistic nature since the library is normally used in practice and not just in simulations. To avoid synthetic behavior the used JSON object was fetched from a movie database rather than being synthesized. The JSON object can be found online hosted on KTH:s webservers[[1]](#footnote-1).

## Memory Manager

To address the problem of memory allocation on heterogenous memory we chose to use a memory manager. The memory manager works on top of one or many memory allocators which in turn work with different memory banks. Each allocator implements a data structure such as a free lists, bitmap or buddy system.

### Memory Architectures

The diverse address spaces represent different memory units with various memory properties. For example, one of the memories might represent a scratchpad memory using spin transfer torque magnetic random-access memory (STT-MRAM) while another memory might be imitating NOR flash. The number of memories are dependent on the architecture design and can thus, in theory, be considered as arbitrary. See figure x for an illustration of the memory manager’s constituent parts.

In this study we limited ourselves to 3 different memory architectures with a memory of 384 KB. The memory architectures were constituted by several memory banks. Each memory bank had different read latencies, write latencies and capacities. The latency values and memory bank sizes are representatives of the target industrial domains. They were acquired from discussion with the industrial project partner Evaderis. The latencies aim at replicating the real latencies found in different types of NVRAM memories.

The first memory architecture, A, contained two memory banks of 128 and 256 bytes respectively. The first bank had a read and write latency of 1 and 10 cycles respectively. The second bank had a read latency of 10 and a write latency of 100 cycles.

The second memory architecture, B, also contained only 2 memory banks. The first bank identical to the first bank of architecture A. And the second bank with a read and write latency of 50 cycles. The bank sizes were retained from the first architecture.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Memory architecture** | **Memory bank** | **Size (KB)** | **Read latency** | **Write Latency** |
| A | 1 | 128 | 1 | 10 |
| A | 2 | 256 | 10 | 100 |
| B | 1 | 128 | 1 | 10 |
| B | 2 | 256 | 50 | 50 |
| C | 1 | 32 | 1 | 1 |
| C | 2 | 96 | 1 | 10 |
| C | 3 | 256 | 10 | 100 |

The last memory architecture, C, contained 3 memory banks. The first one had a read and write latency of 1 cycle, imitating SRAM behavior. The second one had a read latency of 1 and a write latency of 10. Finally, the last bank had a read and write latency of 10 and 100 respectively. The memory bank sizes were 32, 96 and 256 KB ordered from fastest to slowest. This architecture can be seen as architecture A were some of the fast memory has been substituted for even faster, but volatile, SRAM.

An additional memory architecture was also added for testing purposes. The testing architecture had a single memory bank with a capacity of 384 kb as well as a read and write latency of 1 cycle.

### Strategies

The goal of the memory manager is to optimize memory usage by deciding in which memory to put what data. To do so, the memory manager uses various strategies. These strategies are referred to as Occupancy rate, Threshold and Fastest first. These strategies were, as the data structures, implemented naively to enable profound understanding of their behaviors.

The occupancy rate strategy strived to always allocate blocks in the memory with the lowest occupancy. To keep track of the memory occupancy in various memory banks, the data structures kept track of the amount of free memory in their corresponding memory banks (See section 3.4). By using this information, the memory manager could calculate the occupancy of each memory.

The Threshold strategy attempted to exploit the idea that small allocated objects should be accessed more times per byte than larger allocated objects. Thus, when this strategy is applied, the memory manager checks if the size requested is less than a specified threshold. If that is the case, the memory manager delegates the allocation to the memory allocator responsible of the fastest memory. If the size is larger than the threshold the request is passed to the memory allocator of a slower and bigger memory. An exception was made for the C architecture where this strategy instead used 2 thresholds.

The Fastest first strategy was supposed to take advantage of the fastest memory bank as much as possible. By ensuring that the fastest memory bank would always be as full as possible, this greedy strategy was thought to implicate high execution speed at the expense of fragmentation and memory footprint size. Counter-intuitively, this was not the case.

The implementations of the relevant data structures kept track of the largest available memory block in their corresponding memory banks. By taking this information into account the memory manager could avoid following the strategies when they would result in a heap overflow. In a scenario where the memory manager used the threshold strategy resulting in the decision to allocate into the smallest memory bank, and that chosen bank was full, it would thus continue with the next bank. An illustration of this procedure can be seen in figure. Add UML figure

# Results

This section contains the results of the experiments described in the Background chapter. In the first section, Profiling, we present and evaluate the results from the synthetic profiling and real profiling phase of the thesis. In the second section, Memory Manager, present and analyze the results from the strategy phase of the thesis.

## Profiling

This chapter presents the results of the profiling. The section starts

The collected data from the synthetic and real profiling is shown in Table 1. As can be seen in the left most column the LMF (Largest Memory footprint) varied significantly between the different allocators. As a consequence of this the fragmentation at LMF is, in some cases, not of an interesting nature.

### Plateau

The plateau was handled well by all of the data structures. Compared to the upcoming access profiles the plateau was less of a challenge for the data structures.

The free list handled the plateaus in 2260 thousand malloc cycles and 1640 thousand free cycles with a largest memory footprint (LMF) of only 5900 bytes. This LMF was the lowest observed LMF out of all the LMF values obtained from the access-profile data-structure combinations. The free list behavior during the Plateau can be observed in figure ??. In this figure, every blue box symbolizes an allocation where its height illustrates the size of the allocated object and its width the accumulated time.

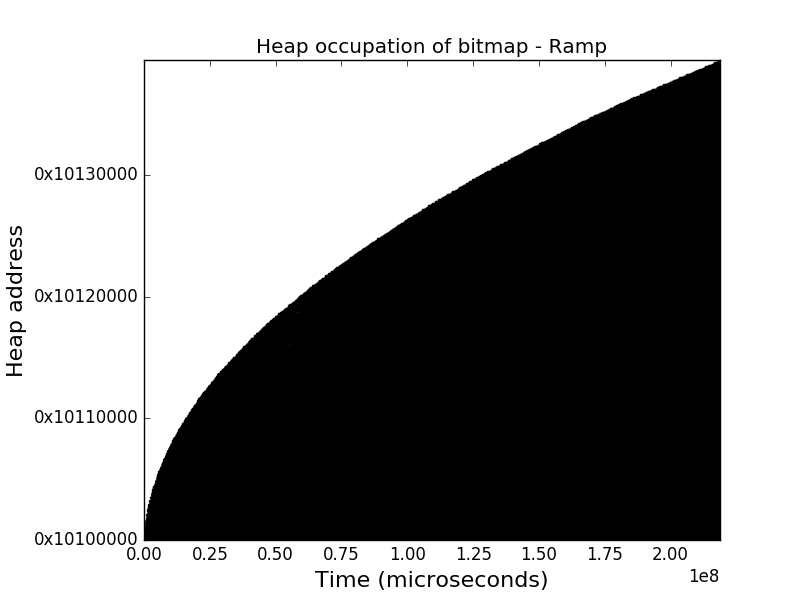
The bitmap, despite being slower than the free list, did well in comparison to other of its performances during other profiles. The bitmap allocator spent a total of 20000 thousand cycles handling memory allocation requests and a total of 2610 thousand cycles handling free requests. Its largest memory footprint was around 8100 bytes. The behavior of the bitmap allocator can be seen in figure ??. Interestingly to note is the white space at the start of the heap at address which is due to the storage of the intrinsic bitmaps.

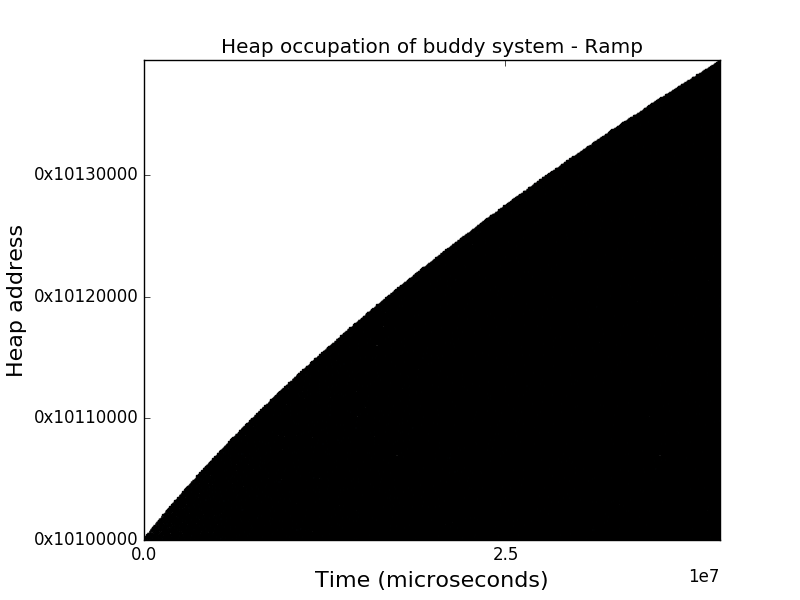
The buddy system did, as the other data structures, have its best performance during the plateau. In total it spent 7200 and 2930 thousand cycles in the malloc respective free functions having a LMF of 7200 bytes. The buddy system did worse than the free list in terms of total cycle count and memory usage. But on the other hand, had a better total cycle count than the bitmap.

The buddy system could, at a first glance, seem to act in a strange fashion. Compared to the other two data structures it chose to allocate the large block at an offset of 0x0000100 from the heap base. This was due to the first required right sized grain already being split into smaller grains and thus not available for the large block. The large block was instead forced to populate the second grain matching its size. The remainders of the first grain, that was split, were then utilized for smaller requests.

### Ramp

The ramps were in general tougher for the three data structures than the plateaus. The behaviors of the free list, bitmap and buddy system during the ramp can be seen in figure x,x and x.

During the ramp, the free list started struggling as a consequence of all the free blocks requiring traversal when searching for a large enough free block to satisfy the received request. The free list spent 12600 thousand cycles freeing blocks which was worse than the bitmap’s and buddy system’s performances. On the other hand, the free list only spent 17600 thousand cycles allocating blocks and had a LMF of 119 thousand bytes. The last two being significantly better than the bitmap’s and buddy’s values.

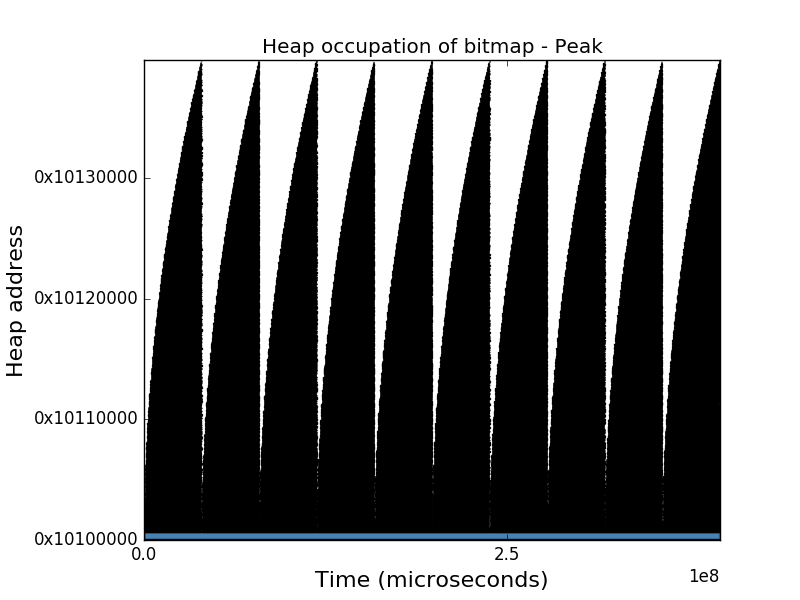
The bitmap suffered a rather large slowdown in comparison to it’s the prestation during the plateau. The bitmap spent 213000 thousand cycles allocating and 2340 thousand cycles freeing while having a large LMF of 258 thousand bytes. The number of allocation cycles, compared with the free list’s and buddy system’s number of allocation cycles, were of catastrophic nature.

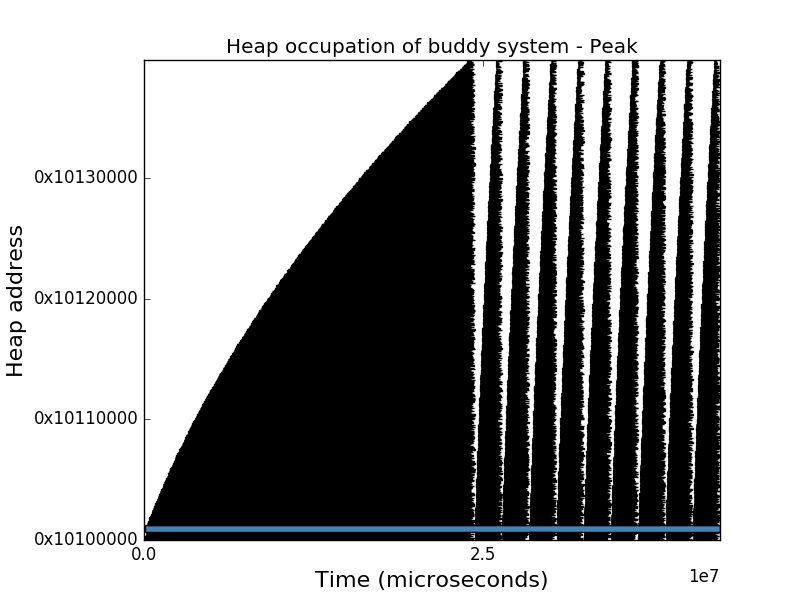
The buddy system did, as the bitmap, not suffer much in terms of freeing cycles but rather in terms of allocation cycles and fragmentation. The buddy system spent 34000 thousand cycles allocating memory and had a LMF of similar size as the bitmap. On the other hand, the amount of cycles spent freeing were close to only 2640 thousand cycles.

Interesting to note is the smoothness of the bitmap’s and the buddy system’s curves in figure and . This is due to the grainsize of 256 bytes being larger than required. The extra space in each block contributes to the total internal fragmentation which in turn contributes to the total fragmentation.

### Peak

The peaks were the toughest of the access profiles for the data structures. The typical intense memory usage associated with peaks meant numerous of allocation and freeing request. The behaviors of the free list, bitmap and buddy system during the peaks can be observed in figure x,x and x.

When facing the peaks, the free list kept a satisfactory low amount of 2250 thousand allocation cycles similar to its performance during the plateau. On the other hand, the free list struggled with the freeing, reaching a remarkably number of cycles spent freeing of 19 100 thousand cycles. During the peaks, the free list had a LMF of 111 thousand, which was the lowest LMF of our three candidate data structures’ LMFs during this access profile.



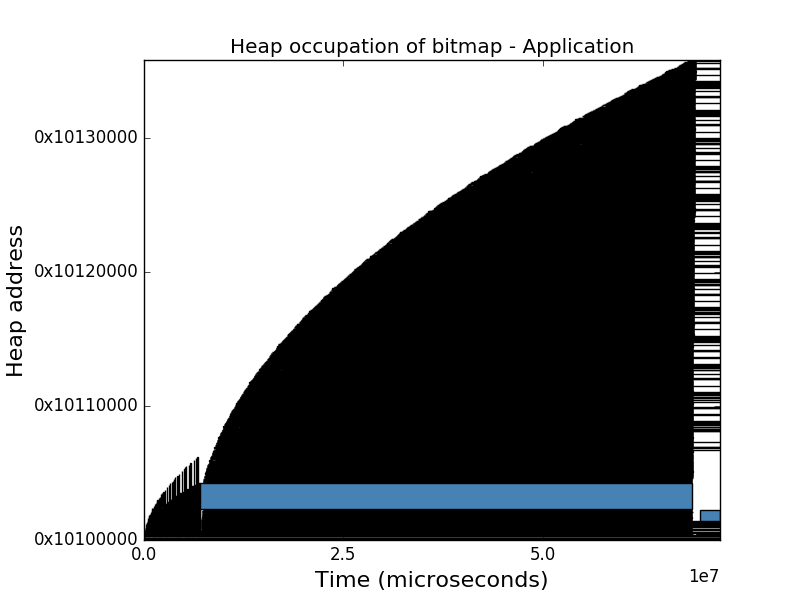
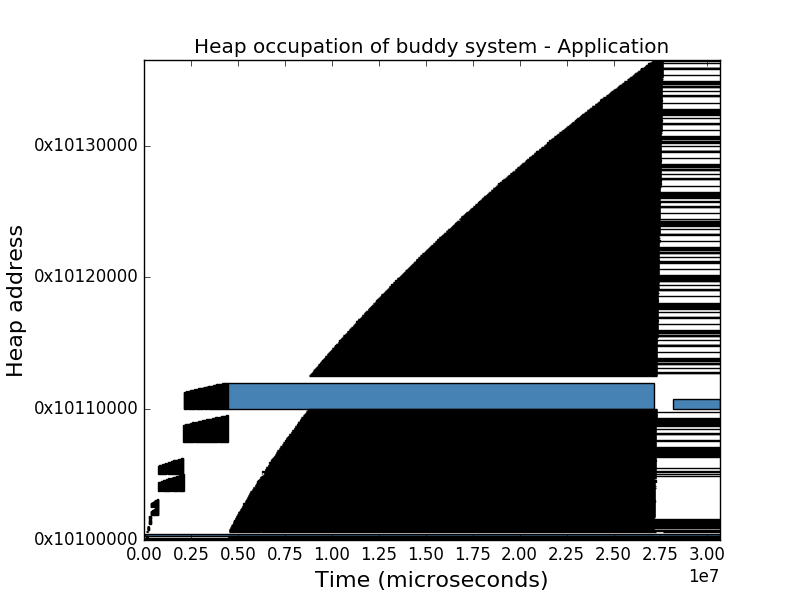
During the peaks the bitmap kept struggling with its memory allocation cycles and memory footprint. The largest memory footprint reached a value of 261 thousand bytes, similar to the value of the buddy system’s average fragmentation. The total amount of cycles spent allocating data was 391 million cycles, more than a factor of 10 times the buddy system allocation cycles. On the contrary, the bitmap spent a slightly lower amount of cycles freeing data than the buddy system.

When stressed by the peaks, the buddy system struggled more than during earlier access profiles. The buddy system spent 36300 and 2940 thousand cycles allocating respectively freeing. Its average fragmentation kept being noticeably high, almost reaching 69%. In other words, it performed slightly worse than the free list in terms of cycles but were still far behind in terms of fragmentation.

An interesting remark is the buddy system’s difference in time when handling various peaks. The first peak took markedly more time than the following peaks which had a constant time of ??. The reason behind the slowness of the first peak is the splitting of the largest grain. This grain has to be recursively split until the acquired grainsize matches the requested block size.

### Application

The application, which aimed at replicating a real-life application, yielded interesting results. The data structures’ responses to the applications requests can be observed in figure ??. By observing these graphs, one can see that the application has characteristics similar to that of the synthetic ramp.

The free list handled the application’s allocation requests with ease. The data structure only needed a total of 460 thousand cycles in the malloc routine. The free list also showed a favorably low LMF barely reaching 41 thousand bytes. On the other hand, the free list spent 1500 thousand cycles freeing, being about three times as much as the buddy system did. This was expected since the application exhibited ramp characteristics and thus should show similarities between its malloc-free cycles ratio and that observed during the synthetic ramp.

The bitmap had, as earlier, a very poor performance in comparison to its competitors. It spent 62 000 thousand cycles executing inside the malloc function but only 1600 thousand cycles in the free function. In total it was the slowest in handling the requests of the application compared to the free list and buddy system. On top of this, it had a large LMF of 235 thousand bytes compared to the free list’s LMF of only 40 thousand bytes.

The buddy system used 21 000 thousand cycles allocating blocks and only 600 thousand cycles freeing blocks. This low amount of freeing cycles is believed to be a result of the buddy system having a best freeing time complexity of O(1). This best-case complexity appears when the buddy system does not need to merge or split since it then only needs to mark a grain as free in the bitmap. Contrary to this good result, the buddy system had a large LMF similar in size to that of the bitmap.

Figure 4: The execution time of the free list, bitmap and buddy system when combined with the plateau, ramp, peaks and application.

Figure 5: The memory usage of the free list, bitmap and buddy system when combined with the plateau, ramp, peaks and application

### Conclusions

By looking at figure ?? we can clearly see that the free list is faster than the bitmap and buddy system. From a performance perspective the free list was thus clearly better in all cases except during the peaks. During the peaks, the buddy system could be a competitor because of its faster handling of the peaks after the first peak. This significates that an application showing mainly peak characteristics could, in theory, benefit in execution time using the buddy system instead of the free list. However, this would require the grainsize to be larger than most block sizes to avoid the split and merging delays.

From a perspective regarding effectiveness of memory usage the free list was an obvious winner. This can be seen in figure ?? where the free list’s largest memory footprint was always smaller than that of the bitmap and the buddy system. In other words, the experiments imply that the free list’s critical points will be of less danger than those of the bitmap and buddy system. In conclusion, usage of free lists, rather than other candidates, will contribute to a more reliable system since free list proved to be the best at avoiding heap overflows and generally were faster.

A big reason for the buddy system’s and bitmap’s large memory footprints were the relationships between their grainsize and the requested block sizes. The block sizes of the synthetic profiles were set with respect to what was thought to be reasonable for an application running on an embedded device. The minimum grainsize of the buddy system and the bitmap were set to 256 bytes while the blocks of medium and small sizes never would exceed that value.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Data Structure** | **Profile** | **Malloc cycles** | **Free cycles** | **Outside cycles** | **Total cycles** | **LMF Fragmentation** | **LMF** |
| **Free List** | **Plateau** | 2 260 472 | 1 641 896 | 3 294 064 | 7 196 432 | 27.649% | 5 888 |
| **Bitmap** | **Plateau** | 19 964 032 | 2 609 494 | 3 294 489 | 25 868 015 | 50.493% | 8 120 |
| **Buddy System** | **Plateau** | 7 200 209 | 2 933 358 | 3 298 701 | 13 432 268 | 40.436% | 7 152 |
| **Free List** | **Ramp** | 17 359 953 | 12 612 828 | 3 166 199 | 33 138 980 | 17.164% | 119 894 |
| **Bitmap** | **Ramp** | 213 226 118 | 2 346 166 | 3 164 519 | 218 736 803 | 61.493% | 258 586 |
| **Buddy System** | **Ramp** | 34 028 366 | 2 642 408 | 3 165 352 | 39 836 126 | 61.492% | 258 582 |
| **Free List** | **Peaks** | 2 257 091 | 19 085 922 | 3 284 664 | 24 627 677 | 65.373% | 111 484 |
| **Bitmap** | **Peaks** | 391 214 122 | 2 606 520 | 3 282 325 | 397 102 967 | 67.744% | 260 577 |
| **Buddy System** | **Peaks** | 36 298 554 | 2 938 646 | 3 280 224 | 42 517 424 | 67.743% | 260 573 |
| **Free List** | **Application** | 476 296 | 1 506 849 | 8 972 302 | 10 955 447 | 64.299% | 40 598 |
| **Bitmap** | **Application** | 61 671 154 | 1 590 591 | 8 971 744 | 72 233 489 | 96.652% | 235 012 |
| **Buddy System** | **Application** | 21 112 853 | 598 198 | 8 971 868 | 30 682 919 | 96.716% | 239 624 |

By adjusting the minimal grain size of the bitmap and buddy system we discovered a trade-off. This trade-off stated that the cycles spent in malloc and free increased when the minimal grainsize was reduced and vice versa. The reason for not decreasing the minimum grain size was thus to keep the execution time of the bitmap and buddy system reasonable.

Table 2: Results from synthetic and application profiling

Since these 2 allocators only allocated in multiples of 256 this meant that a small request would result in a large internal fragmentation. To illustrate this, one could imagine a request of 32 bytes. The requestor would receive a full block of 256 bytes and 224 bytes would never be used. These wasted 224 bytes does, in turn, increase the memory footprint by 224 bytes.

Another factor to take into consideration when evaluating the 3 different data structures is that the buddy system would use deferred coalescing while the free list wouldn’t. This buddy system optimization was done since we discovered that the buddy system was completely unusable, in terms of execution time, when splitting and merging aggressively.

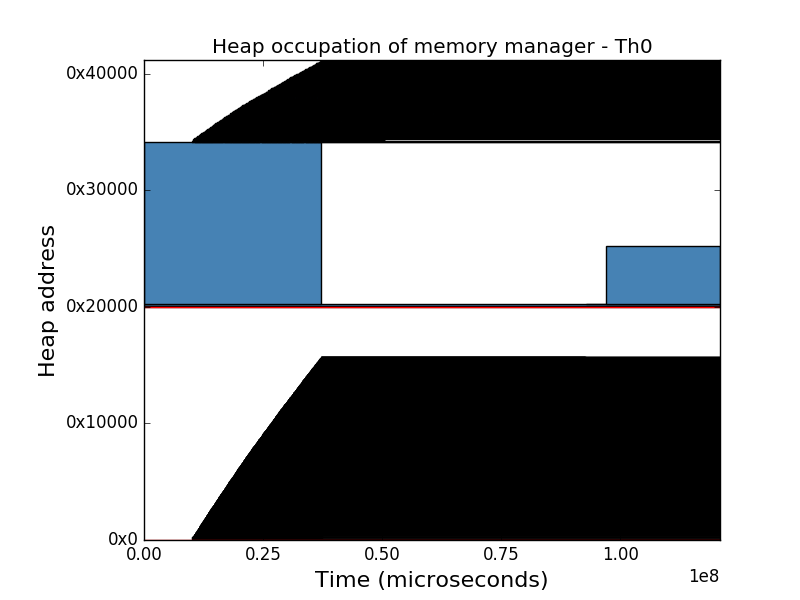
By changing the buddy system’s strategy, it was given a chance to compete with the free list and bitmap. Since the free list didn’t benefit by this optimization, this further strengthens our decision to choose the free list as the most favourable data structure. By implementing deferred coalescing in the free list as well, it could potentially result in the free cycles decreasing to a value only slightly worse than the malloc cycles.

## Memory Manager

In the previous section we concluded that the free list would be the most suitable data structure for Evaderis architectures. During the evaluation of the memory manager’s potential strategies we thus chose to limit ourselves to this data structure.

After simulation of all the strategy architecture-combinations we discovered that they expressed very similar LMF. The fragmentation is thus of greater importance and what will rather focus on during this section. All the results acquired can be observed in table and will be analyzed in section and .

### Architecture A

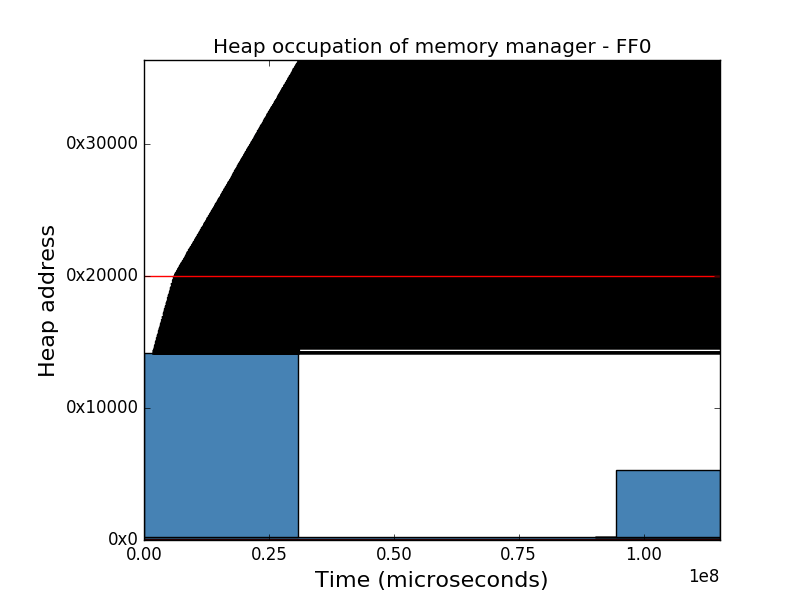
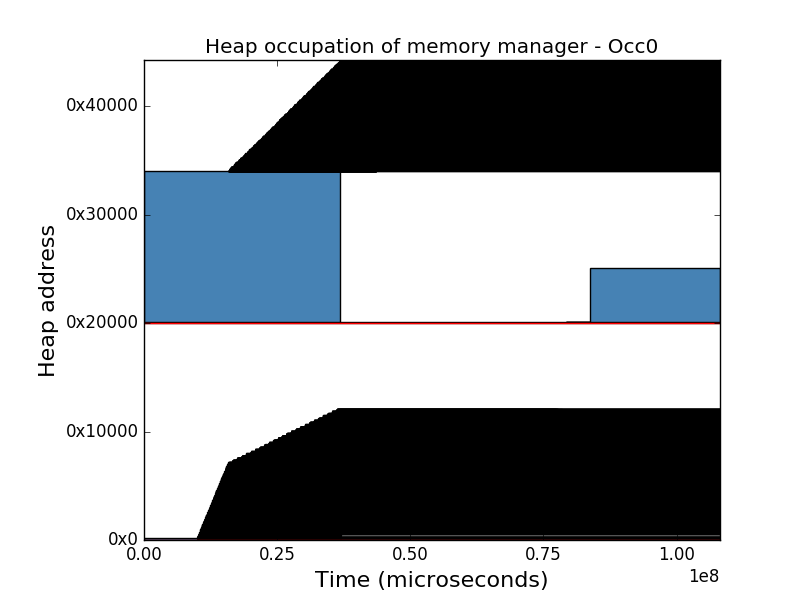
The first architecture had two banks. The first bank was fast while the second bank was slow and larger. When matching the architecture with the threshold, fastest first and occupancy strategies we acquired different access profiles. These access profiles can be seen in figure x. The red line is the border between the first and second memory bank.

The threshold strategy spent 7 million and 56 million cycles in the malloc respectively free routine.

The large amount of freeing cycles is a sign of a high amount of coalescing taking place. It had a relatively high fragmentation of 82.0% similar to the 81.8% occupancy but significantly different from the 59.8% of fastest first. Since they have similarly sized LMF:s, this shows that the fastest first strategy utilized the memory most effectively.

The fastest first strategy had similar cycle counts for malloc and free as the threshold strategy. On the other hand, it only spent 48 million cycles outside of these functions compared to the 58 million and 59 million cycles of the threshold respective occupancy strategy. This significates that the fastest first strategy might benefit from applications being more compute bound than memory bound.

The occupancy strategy had a markedly better performance than the competing strategies. While the threshold and fastest first strategy reached 121 and 115 million total execution cycles, the occupancy strategy merely reached 108 million. Thus, for this architecture, the occupancy strategy was the fastest.



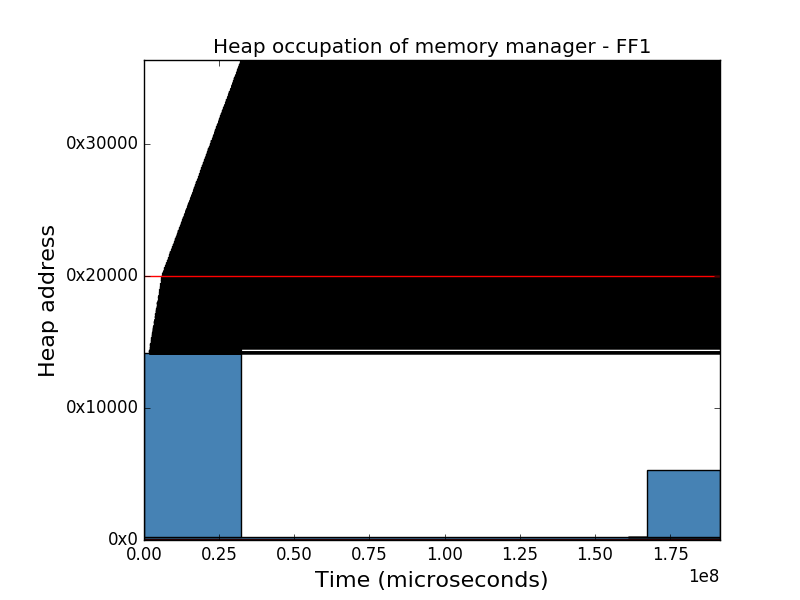
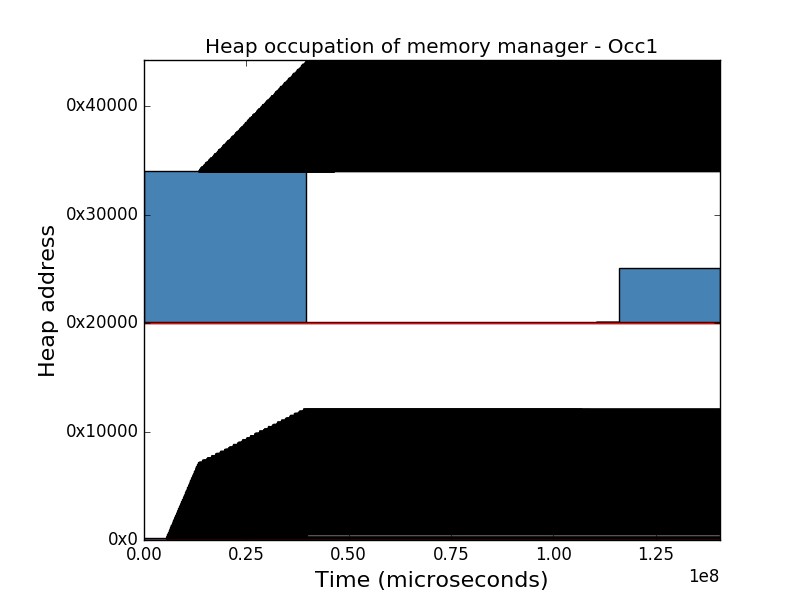
### Architecture B

The second architecture was very similar to the first. The only difference being the different read and write latencies. As a consequence of this, the graphs in figure x and y are very similar, only differing with a stretch in x-direction.

After switching to the second architecture, The threshold’s total cycles were significantly better compared with other strategies. The threshold strategy survived on 130 million cycles while the fastest first and occupancy needed 130 respective 190 million cycles. This shows that the latter two were more negatively affected by the increased write latency of the second architecture.

The fastest first strategy kept having a lower fragmentation and number of outside cycles than its competitors. The lower fragmentation came with a cost in terms of freeing cycles since they were almost twice as high as the competitors’. In conclusion, the longer heap of fastest first required more time to search but could keep a lower fragmentation.

The relative performance of the occupancy strategy became worse after the switch of latency. Its total cycle count increased from 108 million cycles to 140 million cycles. Meanwhile the threshold strategy only increased its total cycle count from 120 million to 130 million cycles. Thus, we can draw the conclusion that no strategy is exclusively the best regarding the performance in terms of execution time. Rather this performance is dependent on the combination of strategy and memory architecture.



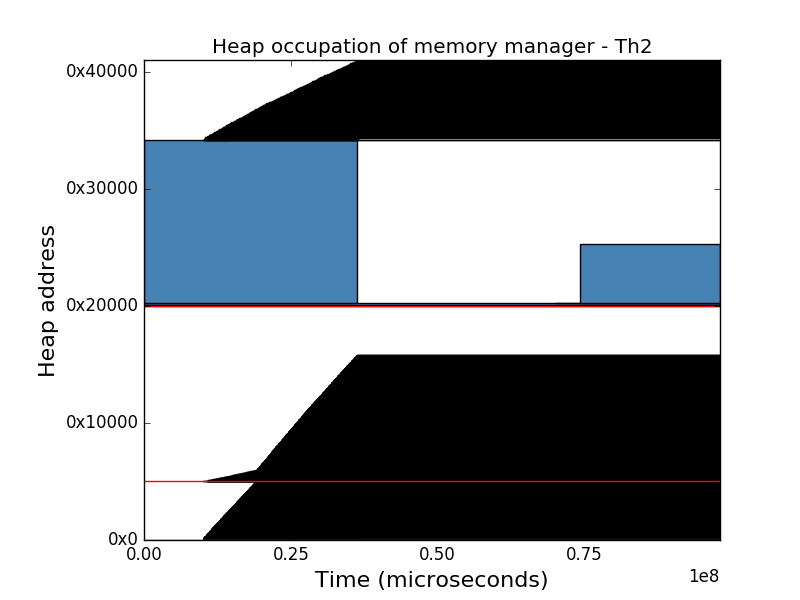
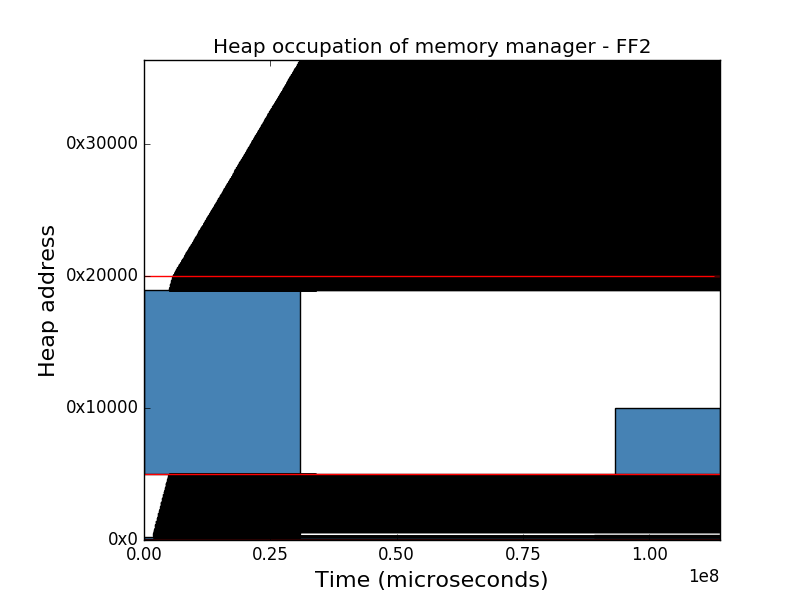
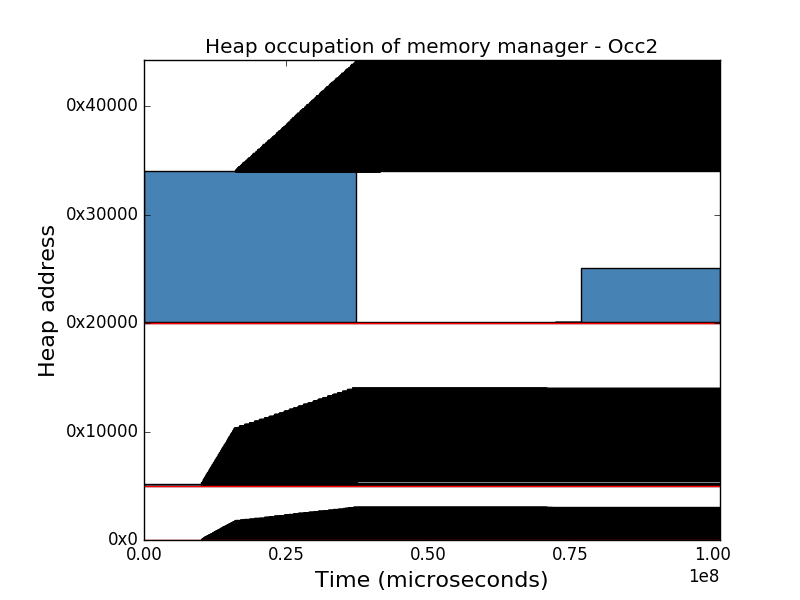
### Architecture C

The last memory architecture differed from the previous two by having 3 memory banks instead of 2. Thus, its graphs seen in figure x contain 2 line delimiters rather than 1. The memory banks are, as in figure y and z, ordered by speed starting with the fastest one at 0x0 and ending with the slowest at 0x20000.

The threshold strategy started performing better when combined with the last architecture. The total cycle count only reached 98 million while the fastest first and occupancy strategies had their total cycle count reach 113 respective 101 million cycles. The threshold strategy also benefited from a lower fragmentation of 63% compared with earlier architectures. This result suggests that layering memory in more memory banks might reduce fragmentation when using a threshold strategy.

During the last architecture, the fastest first kept yielding good results in terms of outside cycles and fragmentation. The strategy merely spent 49 million outside the memory routines compared to the two candidate strategies which spent almost 60 million outside cycles. The fastest first strategy has thus been the fastest in terms of outside cycles during every architecture. We can thus conclude that the fastest first strategy could potentially be the most suited strategy for compute bound applications.

When confronted with the last architecture, the occupancy strategy kept having a large fragmentation of around 80% while the threshold and fastest first fragmentation was approximately 60%. Interesting to note as well are the outside cycles. The strategy realized 59 million outside cycles compared to 57 million and 48 million for the threshold and fastest first strategies respectively. De facto, this strategy had the most outside cycles during every architecture rendering it the least suited for compute intense application.



### Conclusions

The conclusions acquired from the results were many. The most important being that there is no ultimate strategy performing optimally with all architectures. Instead, a good solution could posibly be to use multiple strategies and pair them with the memory types they would work best with.

Another interesting remark was that the fastest first strategy always had the least amount of fragmentation. This was a result of the fastest first strategy seeing the smaller heaps as one large heap. A consequence of this was also that the strategy was less likely to push the MF since it had more potential free blocks for allocation.

We also noted that fastest first always had a fewer amount of outside cycles than its competitors. Consequently, it might be the favourable strategy when dealing with compute-bound applications. On the other hand, it is less likely to be favourable when dealing with applications with frequent allocations and deallocations because of its large amount of freeing cycles.

Depending on the memory architectures different strategies were the best. Further studies are thus needed before deciding which strategy or strategies to apply under which circumstances in future embedded systems.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Strategy** | **Architecture** | **Malloc Cycles** | **Free Cycles** | **Outside Cycles** | **Total Cycles** | **LMF Fragmentation** | **LMF** |
| **Threshold** | **A** | 7 128 696 | 56 217 081 | 57 626 718 | 120 972 495 | 82.015% | 241 550 |
| **Fastest First** | **A** | 7 148 216 | 59 373 510 | 48 773 748 | 115 295 474 | 59.812% | 238 484 |
| **Occupancy** | **A** | 6891723 | 42 122 169 | 59 166 620 | 108 180 512 | 81.778% | 238 412 |
| **Threshold** | **B** | 12 243 797 | 58 409 346 | 59 579 448 | 130 232 591 | 82.015% | 241 550 |
| **Fastest First** | **B** | 8 076 860 | 126 816 324 | 56 706 990 | 191 600 174 | 59.812% | 238 484 |
| **Occupancy** | **B** | 7 581 377 | 69 209 245 | 63 928 516 | 140 719 138 | 81.778% | 238 412 |
| **Threshold** | **C** | 6 541 644 | 34 112 540 | 57 431 095 | 98 085 279 | 63.349% | 241 058 |
| **Fastest First** | **C** | 7 194 603 | 58 234 704 | 48 561 622 | 113 990 929 | 58.169% | 238 487 |
| **Occupancy** | **C** | 7 498 708 | 34 883 744 | 59 005 738 | 101 388 190 | 81.763% | 238 208 |

Table 3: Results from memory manager’s different strategies ran on 3 different architectures.

# Conclusions and Future Work

The purpose of this thesis was to address the issue of memory allocation in NVRAM based architectures. In this chapter we discuss the thesis, its conclusions and possible future work. The conclusions are discussed in section x and the further work is presented in section .

## Conclusion

In this thesis we have analyzed three different data structures in combination with various access profiles. We have discovered trade offs for the bitmap and buddy system regarding their grainsize and execution time. We also concluded that the free list was the most suitable data structure for Evaderis architectures in terms of memory usage and execution time.

The free list was the best data structure

Buddy system may be better than free list with some optimization

## Future Work

This thesis was fairly limited in comparison to what could have been done.

A size ordered free list versus the address ordered free list?

Other memory manager strategies?

Other applications for evaluating the strategies.?

Other JSON Objects?

Different thresholds for threshold strategy?

Segregated free list

Study more compute intensive applications

Access pressures

Request Pattern optimizations

# References

|  |  |
| --- | --- |
| [1] | Spintec, "Overview," Spintec, 2015. [Online]. Available: http://www.spintec.fr/organization/. [Accessed 11 11 2017]. |
| [2] | Evaderis, "About," Evaderis, 2017. [Online]. Available: http://www.evaderis.com/about-evaderis/. [Accessed 11 11 2017]. |
| [3] | C. Layer, L. Becker, K. Jabeur, S. Claireux, B. Dieny, G. Prenat, G. Di Pendina, S. Gros, P. Paoli, V. Javerliac, F. Bernard-Granger and L. Decloedt, "Reducing System Power Consumption Using Check-Pointing on Nonvolatile Embedded Magnetic Random Access Memories," *ACM JETS,* vol. 12, no. 4, p. 24, 2016. |
| [4] | P. R. Wilson, M. S. Johnstone, M. Neely and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," in *IWMM '95 Proceedings of the International Workshop on Memory Management*, Austin, 1995. |
| [5] | R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "Free-Space Management," in *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books, March, 2015 (Version 0.90), pp. 153-167. |
| [6] | T. Grötker, "Fundamentals of Systemc," in *System Design with SystemC*, Hingham, Kluwer Academic Publishers, 2002, pp. 11-40. |

Appendix A

1. See <https://people.kth.se/~thpeter/DynamicMemoryAllocation/walking_dead.json>

   and <https://people.kth.se/~thpeter/DynamicMemoryAllocation/walking_dead_short.json> [↑](#footnote-ref-1)