

Dynamic Allocation for Embedded Heterogenous Memory

An Empirical 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](#_Toc502578655)

[List of Figures ii](#_Toc502578656)

[Abbreviations iii](#_Toc502578657)

[Glossary iv](#_Toc502578658)

[1 Introduction 1](#_Toc502578659)

[1.1 Background 1](#_Toc502578660)

[1.1.1 Evaderis and Dycton 1](#_Toc502578661)

[1.1.2 Problem Background 2](#_Toc502578662)

[1.2 Problem 3](#_Toc502578663)

[1.3 Purpose 3](#_Toc502578664)

[1.4 Goal 3](#_Toc502578665)

[1.5 Methodology 4](#_Toc502578666)

[1.6 Delimitations 4](#_Toc502578667)

[1.7 Outline 5](#_Toc502578668)

[2 Background 6](#_Toc502578669)

[2.1 A Note on Embedded Systems 6](#_Toc502578670)

[2.2 A Multitude of NVRAMs 6](#_Toc502578671)

[2.2.1 Charge storage 7](#_Toc502578672)

[2.2.2 Resistance Switching 8](#_Toc502578673)

[2.3 Checkpointing in Embedded Systems 8](#_Toc502578674)

[2.4 Memory Management 9](#_Toc502578675)

[2.5 Single-Bank Memory Allocators 11](#_Toc502578676)

[2.5.1 Free List Allocator 11](#_Toc502578677)

[2.5.2 Bitmap Allocator 12](#_Toc502578678)

[2.5.3 Buddy Allocator 13](#_Toc502578679)

[2.6 Profiling Techniques for Evaluating Allocators 14](#_Toc502578680)

[2.7 Typical Allocation Patterns Found in Programs 15](#_Toc502578681)

[2.8 SystemC 15](#_Toc502578682)

[3 Methodologies 17](#_Toc502578683)

[3.1 Approach 17](#_Toc502578684)

[3.2 Phases 17](#_Toc502578685)

[3.3 Evaluation criteria 18](#_Toc502578686)

[3.3.1 Execution Time Evaluation 18](#_Toc502578687)

[3.3.2 Memory Usage Evaluation 19](#_Toc502578688)

[3.4 Implementation of the Allocators 20](#_Toc502578689)

[3.5 Synthetic Profiling 21](#_Toc502578690)

[3.6 Simulating Real Program Behavior 21](#_Toc502578691)

[3.7 Managing Multiple Banks Using a Memory Manager 22](#_Toc502578692)

[3.7.1 Emulating NVRAM Using Memory Architectures 22](#_Toc502578693)

[3.7.2 Applying Various Strategies 23](#_Toc502578694)

[4 Results 25](#_Toc502578695)

[4.1 Single-Bank Allocation Using Various Allocators 25](#_Toc502578696)

[4.1.1 One Large Object - A Plateau 26](#_Toc502578697)

[4.1.2 Increasing the Memory – A Ramp 27](#_Toc502578698)

[4.1.3 Studying Intense Memory Usage - Peaks 28](#_Toc502578699)

[4.1.4 Facing the Real World – A JSON Parser 29](#_Toc502578700)

[4.1.5 Discussion 31](#_Toc502578701)

[4.2 Multi-Bank Memory Management Applying Various Strategies 34](#_Toc502578702)

[4.2.1 A Two Bank Architecture 35](#_Toc502578703)

[4.2.2 Modifying the Latencies 36](#_Toc502578704)

[4.2.3 Adding an Additional Memory Bank 37](#_Toc502578705)

[4.2.4 Discussion 38](#_Toc502578706)

[5 Conclusions and Future Work 40](#_Toc502578707)

[5.1 Conclusion 40](#_Toc502578708)

[5.2 Future Work 41](#_Toc502578709)

[References 42](#_Toc502578710)

[Appendix A 1](#_Toc502578711)

# List of Tables

[Table 1: The sizes and latencies of different memory banks in different memory architectures. 23](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578712)

[Table 2: Results from synthetic and application profiling. 33](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578713)

[Table 3: Results from memory manager’s different strategies ran on 3 different architectures. 39](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578714)

# List of Figures

[Figure 1: Evaderis’ architecture based on an Open-RISC processor. [3] 7](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578715)

[Figure 2: The RAM’s data segments. 9](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578716)

[Figure 3: An example of heap-allocated objects and the fragmentation resulting from their placement. 10](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578717)

[Figure 4: A 60KB heap with one allocated block whose placing results in external fragmentation. 11](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578718)

[Figure 5: Illustration of synthetically generated memory allocation patterns. The left most graph illustrates a plateau, the middle graph a ramp and the right most graph a peak. 14](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578719)

[Figure 6: The language architecture of SystemC. 16](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578720)

[Figure 7: The phases of the study presented in this paper. 18](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578721)

[Figure 8: Illustration of the memory footprint. 19](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578722)

[Figure 9: The free list, bitmap and buddy allocators’ behaviours during the plateau. 26](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578723)

[Figure 10: The free list, bitmap and buddy allocators’ behaviours during the ramp. 27](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578724)

[Figure 11: The free list, bitmap and buddy allocators’ behaviours during the peaks. 28](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578725)

[Figure 12: The free list, bitmap and buddy allocators’ behaviours during the JSON parsing application. 29](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578726)

[Figure 13: The execution time of the free list, bitmap and buddy system when combined with the plateau, ramp, peaks and JSON Parser. 30](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578727)

[Figure 14: The memory usage of the free list, bitmap and buddy system when combined with the plateau, ramp, peaks and JSON Parser. 30](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578728)

[Figure 15: The Threshold, Fastest First and Occupancy strategies’ behaviours on memory architecture A. 35](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578729)

[Figure 16: The Threshold, Fastest First and Occupancy strategies’ behaviours on memory architecture B. 36](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578730)

[Figure 17: The Threshold, Fastest First and Occupancy strategies’ behaviours on memory architecture C. 37](file:///C:\Users\thoma\Documents\KandidatExamensarbeteDynamicMemoryAllocation.docx#_Toc502578731)

# Abbreviations

**CPU:** Central Processing Unit

**EEPROM:** Electrically Erasable Programmable Read-Only Memory

**FRAM:** Ferroelectric Random Access Memory

**INSA:** Institute National Des Sciences Appliquées

**JSON:** JavaScript Object Notation

**LFB:** Largest Free Block

**LIFO:** Last In First Out

**LMF:** Largest Memory Footprint

**MRAM:** Magnetoresistive Random Access Memory

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

**PCM:** Phase-Change Memory

**RAM:** Random Access Memory

**RRAM:** Resistive Random Access Memory

**SOC:** System On Chip

# Glossary

**Allocator:** See Memory Allocator.

**Application:** The program code that is not a part of the memory allocation logic.

**Application Cycles:** Cycles not spent in the malloc or free routine.

**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.

**Clock Cycle:** The time required for the execution of one fundamental processor operation.

**Compute Intensive application**: An application reading and writing to objects intensively.

**Cycle:** See Clock Cycle.

**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:** A portion of memory having a size that is a multiple of a word size

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

**Heap Overflow:** A buffer overflow occurring in the heap segment when the segment runs out of memory and software attempts to overwrite data outside of the segment’s boundaries.

**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 continuous portion of memory with a certain type of RAM.

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

**Memory intensive application**: An application issuing numerous malloc and free calls.

**Metadata:** The data used by the allocator 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 a variety of 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 provides a theoretical background and the third presents the different methods used as well as implementation details. Following these, chapter four exposes the results of the degree project as well as the conclusions derived from these. Finally, in the fifth chapter, a summary is provided and thereafter potential future work is discussed.

## Background

This section provides a background for understanding the thesis problem. The first subsection 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 located in Grenoble, a city in the south-eastern part of France. Evaderis was founded by Spintec in 2014, a research lab located in Grenoble as well. This research lab specializes in Non-Volatile Random-Access Memory, NVRAM for short. Their 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. Their 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 system on chips. It aims at providing an easy-to-use memory manager for embedded software developers working with Evaderis’ architectures. 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 foundation 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 memory elements. 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 will be the main problem addressed in the paper and can be summed as the sentence “What is the best strategy of a memory allocator acting on heterogenous memory?”. Since this problem statement is not of trivial nature, it won’t be fully answered by this paper. The paper will instead serve to indicate what techniques and strategies are useful and which ones should be avoided in the context of Evaderis’ architectures.

## 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. These memory properties vary 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 concerning sustainable development.

The ethical aspects of the project are more complicated than the general benefits mentioned earlier. When releasing new technology, one cannot fully comprehend 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 except for having individual tasks. 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 will be based on execution time and memory usage. They will be further described Section 3.3Evaluation.

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 provided. This background serves as a foundation for understanding the remaining parts of the report. In chapter 3 the degree project methodology is presented and in chapter 4 the results are shown and conclusions are derived. Thereafter, a summary of the paper is provided and discussions are made regarding further possible work in chapter 5 .

# 

# Background

This chapter contains a theoretical background providing the theory needed to understand the remaining parts of the thesis. The chapter starts by giving the reader an understanding of what embedded systems are, followed by a brief overview of novel non-volatile memory technologies as well as an introduction to checkpointing in embedded devices. Thereafter the fundamentals of memory management are described and various memory allocation techniques are presented. Additionally, two sections are dedicated to provide a background of common evaluation techniques for allocators as well as typical program characteristics. Finally, a section about the simulator used is provided for the interested reader.

## A Note on Embedded Systems

An embedded system is a computer system within a mechanical or electrical system such as an elevator, a washing machine or a car. Because of their niched environment, embedded systems contain traits making them distinguishable from general purpose computers. Typically, embedded systems have no user interface, little memory space and are often limited by energy constrains.

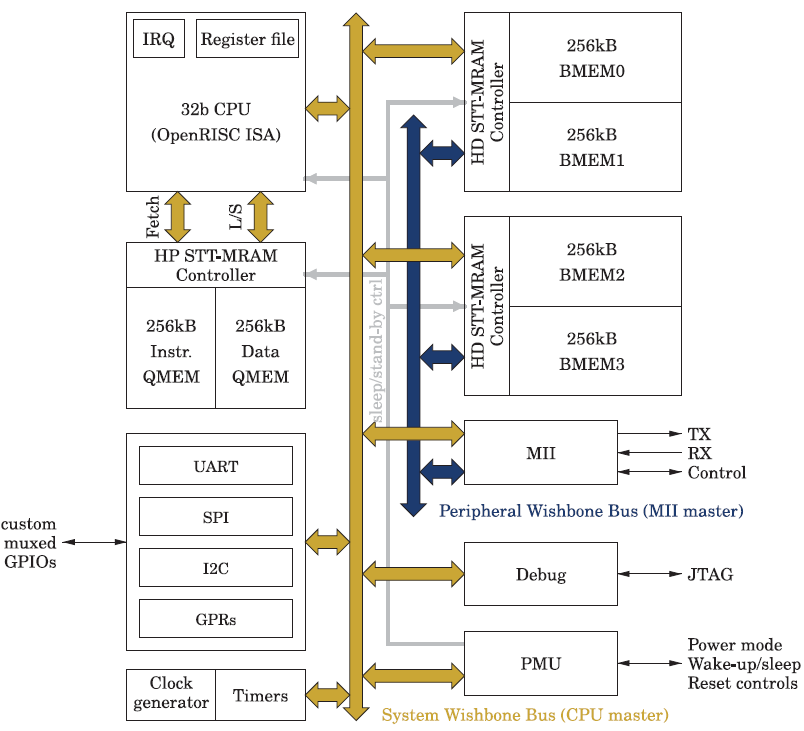
In general purpose computers, RAM is often extended using the disk as well as a technic called paging. By combining the RAM and disk, one obtains what is referred to as virtual memory. As a consequence of the lack of performance and memory in embedded systems these type of systems usually do not use virtual memory(**Reference?**). Instead, the CPU-addresses refer directly to RAM using what is called physical addresses. The Central Processing unit (CPU) is a unit executing program instructions on clock cycles. A clock cycle, or cycle for short, is defined as the time required for the execution of one simple processor operation.

In this paper the simulated architecture of our studies is fundamentally based on Evaderis’ architecture which can be seen in Figure 1. Evaderis processor architecture is based on Open-RISC processor, a 32-bit scalar reduced instruction set computer (RISC) with Harvard microarchitecture, virtual memory support and a five-stage pipeline. In the open-RISC processor, the commonly utilized cache subsystem has been replaced by direct memory interface enabling a direct access to various scratchpad memories. These scratchpad memories are tiny but fast memories extending the memory address range by substituting caches. [3]

## A Multitude of NVRAMs

Non-volatile random-access memory(NVRAM) are the type of byte addressable memory 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. This type of memory differs from regular RAM in other ways than its non-volatility. For example, many NVRAMs have limited write endurance; meaning that memory cells only can change values a limited number of times. An additional difference from regular RAM is that NVRAM tends to have asymmetrical read and write latencies with write latencies being costlier than the former. [4]

Figure 1: Evaderis’ architecture based on an Open-RISC processor. [3]

There are many different types of NVRAM currently in existence. Some commonly used include EEPROM, NOR Flash, FRAM, MRAM, PCM and RRAM. These varies in many aspects such as density per cell, write and read energy as well as manufacturing costs [4]. These types of NVRAMs can be divided into two different categories based on the technic they use. These two technics are charge storage and resistance switching which are presented in the following two subsections.

### Charge storage

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 presence or absence of a 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 electrons/charge significates a bit state of 1 or 0. [5]**(New Source for further reading?)**

Flash memory

**NOR or NAND flash???**

NOR Flash is a variation of EEPROM memory working with arrays of floating gate transistors but using NOR gates as control gates. These nor gates are instantiated by connecting the cells’ floating gate’s legs to ground and a bit line.

FRAM stands for ferromagnetic RAM and works very similarly to, the well known, DRAM with the exception of using a ferromagnetic material instead of the common insulator. This new material has an ability to remember electric fields and ca, as a consequence, be used to store data without a power supply. Compared to EEPROM and flash, FFRAM is better in terms of speed and power usage. On the other hand, it is worse in terms of density using about five times as much space as flash.

### Resistance Switching

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 stands for Magnetoresistive Random-Access Memory. Unlike many other NVRAMs, MRAMs’ read and write latencies are symmetrical. In this type of NVRAM data is stored using grids of magnetic tunnel junctions. These junctions are created by placing a thin insulator between two ferromagnetic plates. These ferromagnets can hold a magnetic polarization. [6]

One of the plates’ magnetic polarization is fixed, serving as a reference, while the other one’s polarization is changeable by applying an external magnetic field. When the magnetic polarization of the ferromagnets is directed in the same direction the resistance is low, otherwise it’s high. Thus, by changing polarization one can store a 0 or 1 in the magnetic tunnel junction. [6]

PCM stand for Phase Change Memory and utilizes chalcogenide glass to store data. The chalcogenide glass has two different state. These are the low resistance crystalized state and the high resistance non-crystalized state. By heating or cooling the glass changes state and thus resistance. As a consequence, since the resistance represents its bit state, PCM can store bits. [7]

RRAM stands for resistive Random-Access Memory. Unlike MRAM and PCM, this technology does not use magnetic polarization or heat for storing data. Instead the technology uses two electrodes at which ions dissolve and precipitate. This procedure affects the resistance and can thus be exploited for dadata storage. RRAM is believed to potentially be able to increase modernt IT systems’ performance while simultaneously reducing their energy consumption

<http://www.fz-juelich.de/SharedDocs/Pressemitteilungen/UK/EN/2013/13-04-23batterie.html>

## Checkpointing in Embedded Systems

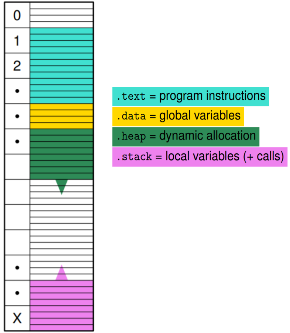
Checkpointing is a technique used in embedded devices for saving energy. By not always being powered, the device applies, what is called a normally off technique. The idea behind this technique is that the device is unpowered until it’s required to execute an action. When time comes for the device to 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]

Figure 2: The RAM’s data segments.

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. 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

Memory management is the act of deciding where in RAM to put what data. The memory is, in most cases, divided into data segments containing different types of data. These are the text, data, stack and heap segments. The first two have fixed sizes specified at compile time while the second two are allowed to grow and shrink as the application uses them [8]. In this paper the notion application is defined as the program code that is not a part of the memory allocation procedure.

The text segment contains program instructions and the data segment contains global variables. The heap segment is used for allocation of variables whose sizes are not known in advance by the compiler. Finally, the stack is a LIFO structure containing local variables as well as registers values from function calls. The described data segments are illustrated in Figure 2.

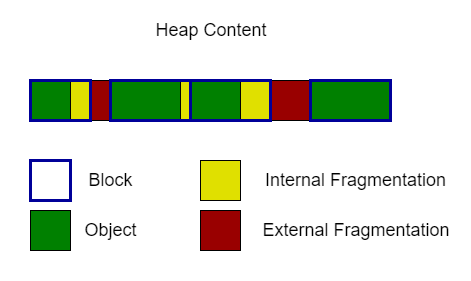
In this paper we focus on the allocation and deallocation of heap data aiming at withholding high performance while avoiding heap overflows to ensure high reliability. A heap overflow is a type of buffer overflow occurring in the heap segment when the segment runs out of memory and software attempts to overwrite data outside of the segment’s boundaries.

Figure 3: An example of heap-allocated objects and the fragmentation resulting from their placement.

To allocate data to the heap one normally uses a piece of software referred to as a memory allocator. This memory allocator is responsible for preventing heap overflows. To communicate with the programmer, the allocator has an interface of two functions called malloc and free. These functions are used to allocate and deallocate blocks of memory. In this paper, a block of memory is defined as one contiguous piece of memory used by the allocator to store data. Additionally, an object is defined as one contiguous piece of memory used by the application to hold data.

The malloc function is invoked with the requested size as a parameter. The function finds a block in memory and returns the address of an object placed inside the block to the caller. The free function, takes an object address as an input argument and is invoked to free an object’s corresponding block. By freeing a block, that particular memory space can then be reallocated later by the malloc function.

A common issue when using these functions is the appearance of numerous small free blocks in the heap. These free blocks appear either because the allocator returns blocks of too large size in comparison to the requested object size or because of freeing of non-consecutive blocks. In the first case, referred to as internal fragmentation, every byte in the block that is not a part of the object is wasted. The second case is what normally is referred to as external fragmentation which is the reason why we sometimes can’t allocate a 20 KB block in a heap with a total free space of 40 KB. External fragmentation as well as internal fragmentation are both measured in percentage. An illustration of external and internal fragmentation can be seen in Figure 3.

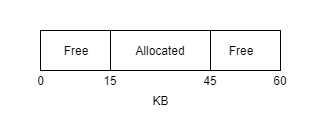
To comprehend external fragmentation, imagine a 60KB heap consisting of 2 free blocks of size 15 KB and 1 allocated block of size 30KB. Thereafter, imagine that the allocated block was placed in-between the two free blocks as in Figure 4. If the allocator then receives a request of size 20 KB, it will not be able to find a large enough block since it only has two free slots of size 15 KB.

Figure 4: A 60KB heap with one allocated block whose placing results in external fragmentation.

To combat internal and external fragmentation various data structures can be utilized. Additionally, some data structures can apply various policies. These policies can, for example, be to always allocate the first block of enough size or to always allocate in the most fitting block for the request.

## Single-Bank Memory Allocators

Memory allocators use different dynamic 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 [9]. In section 2.5.1, free lists are presented, thereafter section 2.5.2 treats bitmaps. Finally buddy allocators are explained in section 2.5.3. These sections are not meant to cover these data structures in detail, as this would be too comprehensive, but rather serve as a reminder. Unexperienced readers are recommended to read the “free space management” section in the book “Operating Systems: Three easy pieces” [9] as well as the Wilson survey [10].

### Free List Allocator

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 a free block contains both information about the free block’s size as well as a pointer to the next free block. The memory allocator code can thus limit itself to one pointer to the free list head to be able to access all free blocks in the heap.

Unlike the unallocated block header, headers of allocated blocks only contain information about the allocated block’s size. This size is read to know how much memory to free when freeing a block.

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 adds its headers to the block and returns an address to the remaining part of the block.

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 object 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 free 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 objects since they might fit into the newly coalesced block.

For free lists there exists many different policies for selecting a free block to allocate and return. Commonly used policies are first fit, best fit and next fit. In this paper we limit ourselves to first fit for simplicity. In first fit the allocator traverses the list returning the first free block large enough for the requested size. When using the best fit policy, the allocator 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 first-block pointer along the free list. This is done in an attempt to combat fragmentation.

### Bitmap Allocator

The bitmap allocator uses a bitmap to keep track of which parts of memory are in use and which parts are not. The bitmap is a data structure that maps a bit to a, so called, grain of memory. A grain is a portion of memory having a size that is a multiple of a word size. The state of the bit provides information about whether its corresponding grain is occupied or not. The size of a grain may vary depending on the implementation. This grain size affects the speed of memory allocation and deallocation and should thus, be chosen carefully. The bitmap is stored in its own memory area and maps to a memory range outside this area or in some cases to its own area as well.

When an allocation request is received the allocator starts the allocation process by calculating the number of grains needed to satisfy the request. It then proceeds by searching for a continuous amount of non-occupied grains using the bitmap. When enough continuous non-occupied grains are found, they are marked as occupied. Thereafter, the address of the first grain 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 requires an additional bitmap marking the last grain in allocated blocks. When the free routine is invoked, the given address is converted to a bit location in the bitmap. This bit is then marked as non-occupied and the allocator checks if the grain is marked as a last grain. If so, the memory allocator stops its freeing procedure. Else way the bit location is set to the subsequent bit and the allocator continues to check the bit map recursively, using the same procedure.

An advantage of the bitmap allocator is that it allows for block searching starting at a chosen address. This is not possible in the free list or bitmap since they only allocate at the beginning of blocks with fixed addresses. An additional advantage of bitmap implementations is that they are good at handling small object requests. [10]

### Buddy Allocator

A buddy allocator uses a buddy system to keep track of free space. A buddy system is a data structure similar to a bitmap having multiple grain sizes. In a buddy system, the memory is conceptually thought of as one large block of size 2^N. This block is seen as a root block in a binary tree where leaves are grains. The data structure keeps track of the availability of tree blocks using a bitmap mapping 1 bit to 1 block.

The size of a block is dependent on its deepness in the tree, becoming half as large for each level. This allocator only allocates blocks with sizes equal to powers of 2 since the heap is a power of 2 and each tree block can only be split in 2. Upon receiving a memory request, the buddy allocator calculates the right tree level by rounding up the requested size to the nearest power of 2. The allocator then searches for a free block on size’s corresponding level.

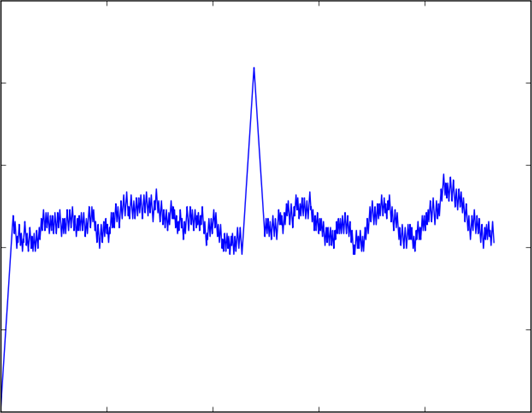
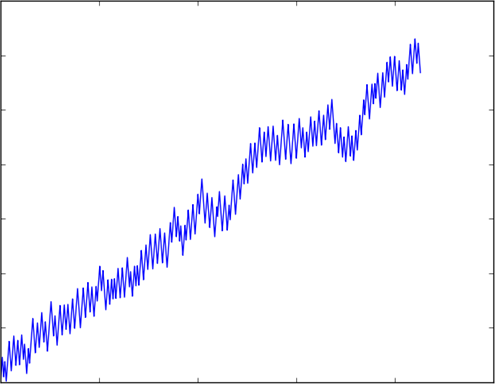
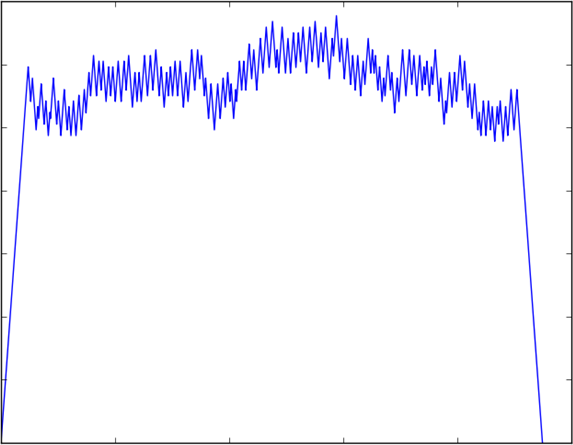
When the allocator can’t find a free block in its matching level, splitting occurs. Splitting is the act of recursively moving up the tree trying to split free blocks higher up to enable the allocator to find a free block on the desired level. When splitting the allocator sets the parent node’s corresponding 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 into a larger block. The act of coalescing is the act of marking both child nodes as allocated and the parent node as free. Coalescing allows the buddy allocator to handle large requests after small requests have been allocated and freed.

In the buddy allocator, served blocks risk being too large for the requested size and internal fragmentation may thus be common. It is also common that the allocator is rather fast compared to other allocators but instead suffers from fragmentation issues [10].

## Profiling Techniques for Evaluating Allocators

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 of a program and tracing its memory allocation and deallocation while keeping track of the status of the heap. Profiling can be applied on either real-world applications or stochastic applications. In this report we refer to profiling of real world applications as real profiling and profiling of stochastic applications as synthetic profiling. [10]



A

B

C

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. [10]

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. [10]

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

Figure 5: Illustration of synthetically generated memory allocation patterns. The left most graph illustrates a plateau, the middle graph a ramp and the right most graph a peak.

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.

## Typical Allocation Patterns Found in Programs

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 5 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 often constitutes of large 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 5.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 5.b.

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 5.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. [11]

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. [11]

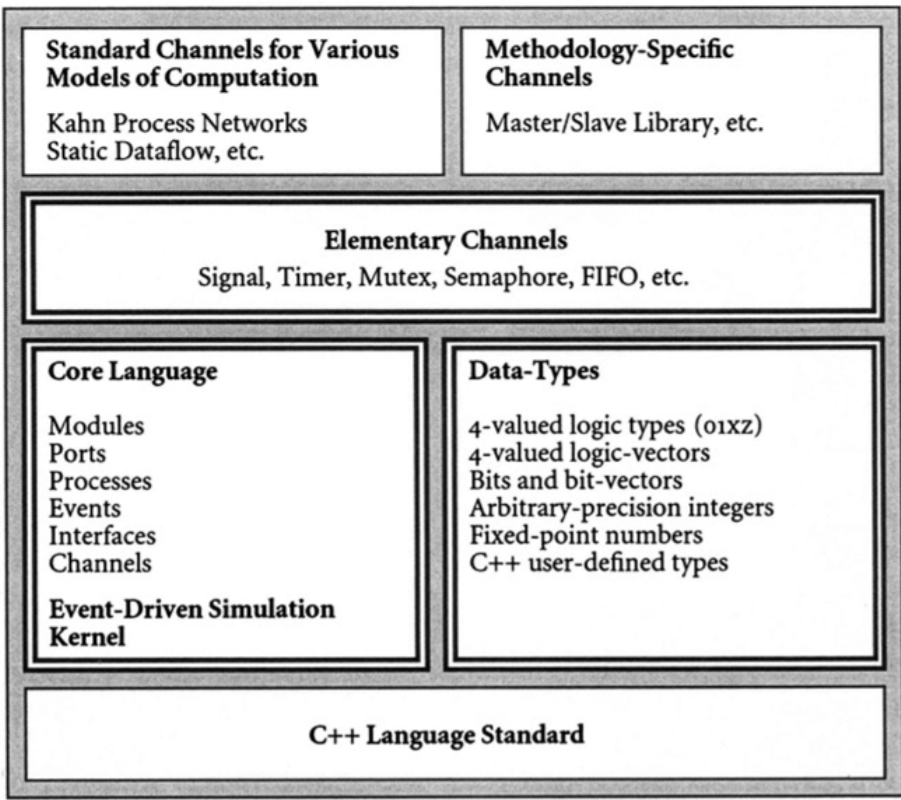
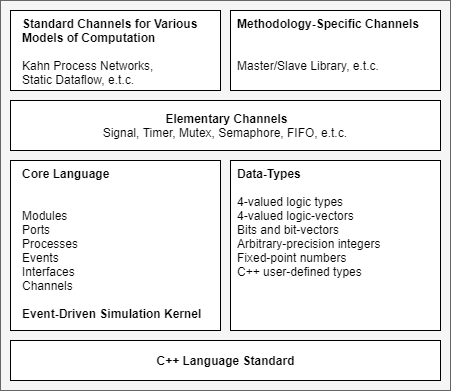
The SystemC layered language architecture can be found in Figure 6. The base layer of SystemC’s language architecture contains the simulation kernel built on top of C++. This kernel serves as the foundation for higher level structures. By combining the kernel with modules, ports, processes, events, interfaces 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. [11]

Figure 6: The language architecture of SystemC.

# 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 as well as the different phases of the work. This is followed by presentations as well as definitions of the various metrics used for evaluation. Following this, implementations of the different allocator techniques as well as synthetic profiles are described. Additionally, a section is dedicated to present the real profiling conducted. Finally, the last section provides information regarding multibank memory architectures and suitable strategies for these.

## 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 7.

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 as well as libraries such as IEEE Xplore and ACM digital library.

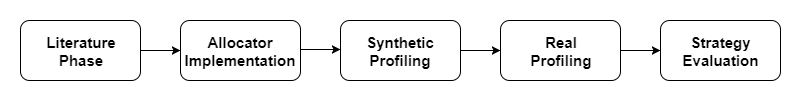
The second phase was the allocator implementation phase. During this phase the three different allocators were implemented. The free list was implemented as an address ordered linked list with a first fit policy. The bitmap was implemented by placing two bitmaps in the start of the heap and using a grain size of 256 bytes. Likewise, the buddy allocator was implemented using a bitmap at the start of the heap and a minimal grainsize of 256 bytes. Implementation details will be further described in Section 3.4.

Figure 7: The phases of the study presented in this paper.

The third phase, referred to as synthetic profiling phase, had the goal of implementing 3 synthetic profiles for evaluating the 3 allocators’ performances when facing various 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 2 in Section 2.7.

The forth phase, the real profiling phase, aimed at determining which allocators 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 taken off the shelf. This program was run in combinations with the 3 different allocators implemented in the third phase. Thereafter, data regarding performance in terms of execution time and memory usage were collected. By analyzing this data, decisions regarding which allocators to keep and which ones to eliminate could be made.

The fifth and last phase was the strategy evaluation phase. During this phase, 3 different memory architectures as well as 3 different strategies were designed and implemented. The architectures were designed with the goal of representing common memory properties in the field. The strategies acted as decision models determining which memory bank should be given which memory request. The goal of this phase was to acquire a better understanding about how NVRAMs’ asymmetricity affect performance.

## Evaluation criteria

Various evaluation criteria were used to evaluate the allocators. To evaluate execution time, cycle counts were used. For evaluation of system reliability, memory usage as well as fragmentation were studied. Section 3.3.1 presents how execution time was evaluated and Section 3.3.2 how memory usage was evaluated.

### Execution Time Evaluation

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

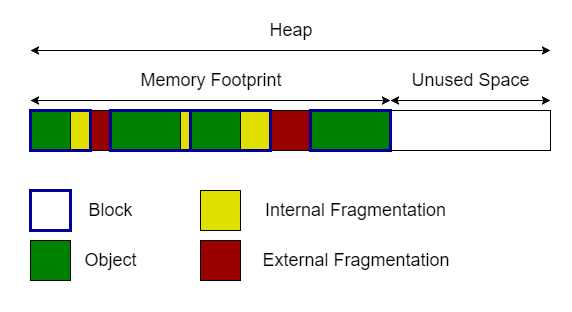
Malloc cycles are defined as the total amount of CPU-cycles spent in the malloc routine. In other words, the total amount of cycles spent finding a free block and returning its address to the calling code, referred to as the application.

Figure 8: Illustration of the memory footprint.

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.

Application cycles are cycles which were not spent in the malloc or free routine. This metric is of particular interest when discussing compute intensive applications. Similarly, malloc and free cycles are of particular interest when discussing memory intensive applications. Compute intensive applications are applications reading and writing to objects intensively while memory intensive applications are applications issuing numerous malloc and free calls.

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

To evaluate memory usage, a metric referred to as the memory footprint is utilized. The memory footprint is defined as the size of the heap that is currently in use. It is calculated by taking the highest address in use subtracted by the base of the heap. The memory footprint is illustrated in Figure 8. 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 an allocator’s or allocation strategy’s reliability we used a notion referred 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. We want to study fragmentation because it has a dramatic impact on allocator performance and in turn on program performance. Fragmentation is relevant to analyze when the LMF is similar for different allocators or allocation strategies. It indicates whether the LMF would likely continue to grow or not.

## Implementation of the Allocators

The allocators were implemented in c inside of the simulation environment. They were implemented naively without optimization. Doing this rather than using off the shelf allocators enabled a thorough understanding of the allocators behaviors and decisions. Since we did not want to give any allocator any advantage by optimization we refrained from assembly implementations. By avoiding usage of assembly, the allocators were also indirectly independent of the Instruction Set Architecture.

The free list was implemented in place as a singly linked list ordered by addresses since this simplified searching of neighboring free blocks. For simplicity, 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 free block. For taken blocks, the pointer was omitted. The header thus only contained size information. For simplicity, this header was stored at the top of every block.

The bitmap and buddy allocator were implemented according to the theory described in Section 2.5. Their bitmaps were stored in the same memory as the rest of the data. As a consequence, large bitmaps could limit the amount of data that could be stored in memory. Since the bitmap size is dependent on the amount of grains mapping memory, the number of grains were reduced by increasing the grain size. This grain size was additionally shown to affect execution time negatively when decreased.

The buddy allocator was implemented with one bitmap mapping to different levels with different block sizes. Since implementations were made naively the buddy allocator was made to always merge freed blocks when possible and only split them when necessary. As a result, the buddy allocator would always have as big blocks as possible free.

The naive implementations were thought to be fair but proved to be a catastrophe for the buddy allocator. By merging free blocks as soon as possible, the allocator was forced to do splits that could have been avoided by re-using unmerged free blocks. This resulted in a very slow execution time compared to other allocators. The buddy allocator 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 and as a consequence would reuse unmerged free blocks when possible.

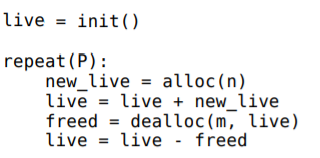
Apart from the core implementation and implementation of a new strategy the allocators were implemented to store heap information. This information was needed for later implementation of strategies. The information constituted of data about the memory bank’s largest free block (LFB) as well as its total remaining free space.

## Synthetic Profiling

The different allocators were combined with different access profiles enabling a pre-analysis of the allocators’ behaviors. The goal of the pre-analysis was to discover the allocators’ strengths and weaknesses under different allocation patterns. Since 3 synthetic profiles and 3 allocators were used, a total of 9 combinations were analyzed.

To synthesize the access profiles, described in Section 2.7 three different object categories were utilized. These categories were small sized objects, medium sized objects and large sized objects. Regarding the small and medium category, the object sizes were uniformly distributed in two different size ranges. These ranges were created since same sized objects was to synthetical and could benefit some allocators but not others. The object sizes aimed at replicating the object sizes used in practice and were chosen based on the available expertise at INSA.

The small object sizes varied from 12 bytes to 24 bytes representing mainly small local variables used temporally. The medium object sizes varied from 128 to 256 bytes, representing larger allocation such as arrays and data structures. Large objects were of size 4096 bytes. The large objects 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 function used a global array to keep track of allocated objects. To generate the access profiles the function invoked the malloc and free functions repeatedly with varying parameters. The malloc function was called with either the large size or a stochastically chosen size from the small or medium size ranges. The free function deallocated objects with addresses acquired randomly from the array of allocated objects.

To simulate a plateau the first 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 5.A. This allocation was simply done by calling the malloc function with the large block size as the parameter and storing the returned pointer. After this steep increase, 10000 objects were allocated and deallocated with sizes in the small or medium category. This would result in the creation of a nearly flat memory usage as seen in the middle of Figure 5.A. After the allocation and deallocation of these 10000 blocks the large sized block would be freed to generate the decreasing end of the plateau.

The ramp, which can be seen in Figure 5.B, was created without a large block. Instead it started to allocate small and medium sized blocks immediately. To create the almost linear increase of the ramp, allocations were done more frequently than deallocations. The reason why the ramp would not be completely linear was that the freed blocks were randomized and thus the freed sizes as well.

The peaks, which can be observed in Figure 5.C, were designed to mimic the intense memory usage commonly seen in programs [10]. They were created on top of a plateau substituting the flat part with peaks. The pattern was created by allocating a large sized object followed by 10 peaks and a deallocation of the large object. The peaks were created by allocating 10000 medium or small sized blocks and then deallocating these repeatedly. In essence, all 10000 object would always be allocated before being deallocated in a randomized order.

## Simulating Real Program Behavior

As an example of a real-life application 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[[1]](#footnote-1) 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.

The execution of this application 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[[2]](#footnote-2).

## Managing Multiple Banks Using a Memory Manager

To study memory allocation on heterogenous memory a memory manager was designed. The memory manager works on top of one or many memory allocators which in turn work with their proper memory bank as well as data structure. This permits the memory manager to use various allocator types for different memory banks. Since we concluded that the free list allocator was the most suitable this won’t be a necessary feature to investigate. Instead, focus is put on various strategies for choosing which object to put in which memory bank. To present this, this section is structured in two subsections presenting memory architectures and allocation strategies respectively.

### Emulating NVRAM Using Memory Architectures

The various memory architectures contain different setups of memory banks having with varying properties. For example, one of the banks 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 is dependent on the architecture design and can thus, in theory, be considered as arbitrary.

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. This is done by utilizing latencies of the same order of magnitude as NVRAM latencies since the exact latencies are not known. These latencies were implemented as additional CPU cycles.

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. This was chosen to emulate the read and write latency asymmetry typically observed in NVRAM implementation [4].

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. The reason for using similar read and write latencies in the second architecture was that some NVRAMs and symmetrical while others are not.

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. This bank emulates SRAM because of its symmetrical latencies.

Table 1: The sizes and latencies of different memory banks in different memory architectures.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

### Applying Various Strategies

The goal of the memory manager is to minimize the total cycle count and fragmentation by deciding in which memory bank to put what data. To do so, the memory manager used various strategies. These strategies are referred to as Threshold, Occupancy and Fastest First. The strategies were, as the allocators, implemented naively to enable profound understanding of their behaviors.

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 was applied, the memory manager checked if the size requested was less than a specified threshold. If that was the case, the memory manager delegated 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. A slightly different implementation was made for architecture C where this strategy instead used 2 thresholds. The thresholds for the strategy were chosen arbitrarily.

The Occupancy 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 allocators 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 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.

When implementing the allocators, variables were added to keep track of the largest available memory block in the memory banks. By utilizing this information, 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.

# Results

This section contains the results of the experiments described in the Methodologies chapter. The first section of the results show that the free list is generally the most suitable for single-bank allocation for memory banks of our sizes. The second section presents results from the strategies evaluated on multiple memory architectures. In this last section, there were no outstanding strategies as in the first section. All figures found in this chapter are available at a larger scale in appendix A.

## Single-Bank Allocation Using Various Allocators

This section presents the results of the synthetic and real profiling. The first 3 sections study the free list, bitmap and buddy allocators performances during the plateau, ramp and peaks. The last section describes these allocators’ performances when facing a JSON-parsing application. It is concluded that the free list is the most suitable allocator for memory banks of the given sizes.

The collected data from the synthetic and real profiling is presented in Table 2 at the end of this section. As can be seen in this table, 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. Focus is thus instead put on the LMF value instead of if it is likely to increase or decrease.

### One Large Object - A Plateau

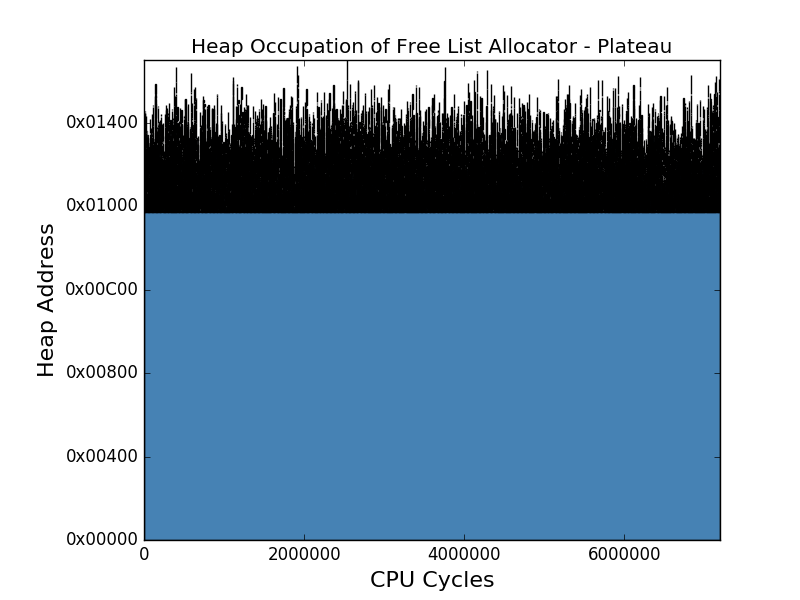
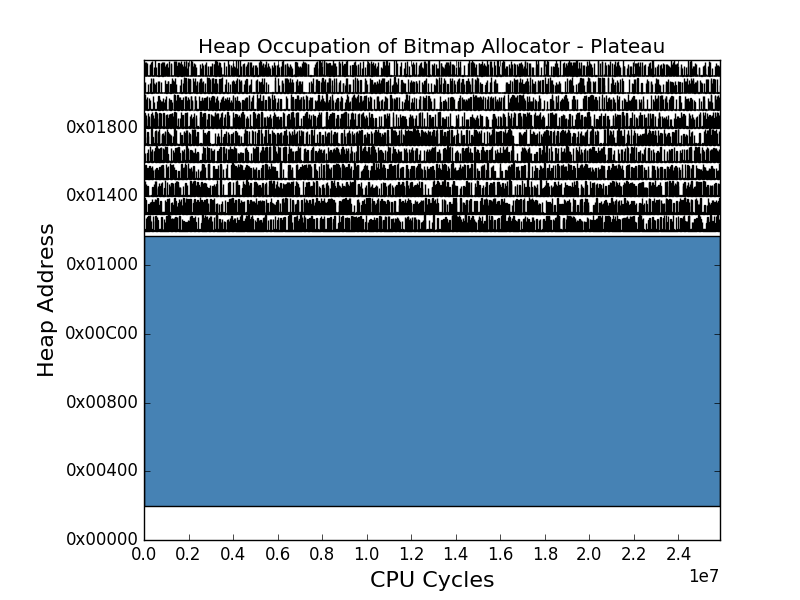
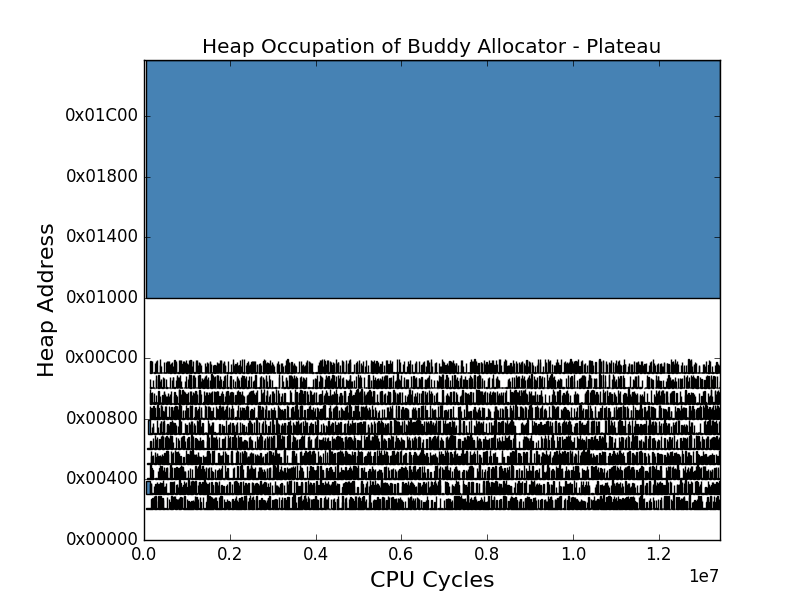
The plateau was handled well by all of the allocators. The free list, bitmap and buddy allocators’ behaviors during the Plateau can be observed in Figure 9. In this figure, every blue box symbolizes an allocation where its height illustrates the size of the allocated object and its width the accumulated time. Interestingly to note in Figure 9 is the white space at the start of the bitmap’s and buddy allocator’s heaps which is due to the storage of their bitmaps.

Figure 9: The free list, bitmap and buddy allocators’ behaviours during the plateau.

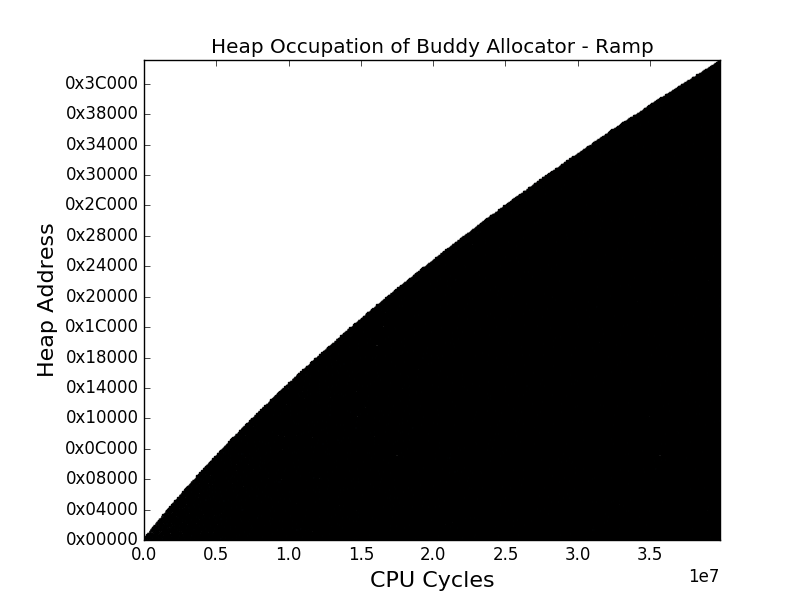
The free list handled the plateau in 2260 thousand malloc cycles and 1640 thousand free cycles with a largest memory footprint (LMF) of only 5900 bytes. This LMF was the lowest observed LMF of the plateau-allocator combinations. Thus, the free list is showing high reliability when faced with the plateau compared to the alternatives.

The bitmap, despite being slower than the free list, did well in comparison to other of its performances when stressed by other profiles. The bitmap allocator spent a total of 20 000 thousand cycles handling memory allocation requests and a total of 2610 thousand cycles handling free requests. In total it executed 25 million cycles. This was significantly worse than the total cycles of the free list and buddy allocator. In conclusion, the bitmap allocator was the slowest candidate when faced with a plateau.

The buddy allocator did, as the other allocators, have its best performance during the plateau. It spent 13 million total cycles having an LMF of 7200 bytes. The buddy allocator did worse than the free list allocator in terms of total cycle count and memory usage. But on the other hand, had a better total cycle count than the bitmap. Thus, in terms of execution speed during plateaus, the buddy allocator is faster than the bitmap allocator but slower than the free list allocator.

The buddy allocator could, at a first glance, seem to act in a strange fashion when observed in Figure 9.c. Compared to the other two allocators it chose to allocate a large block at an offset of 0x00100. This was due to the first required right sized block already being split into smaller blocks and thus not available for the large object. The large object was instead forced to populate the second block matching its size. The remainders of the first block, that was split, were then utilized for smaller requests.

### Increasing the Memory – A Ramp

The ramp was in general tougher for the three allocators than the plateau. This is due to its increasing memory usage. The behaviors of the free list, bitmap and buddy allocator during the ramp can be seen in Figure 10.

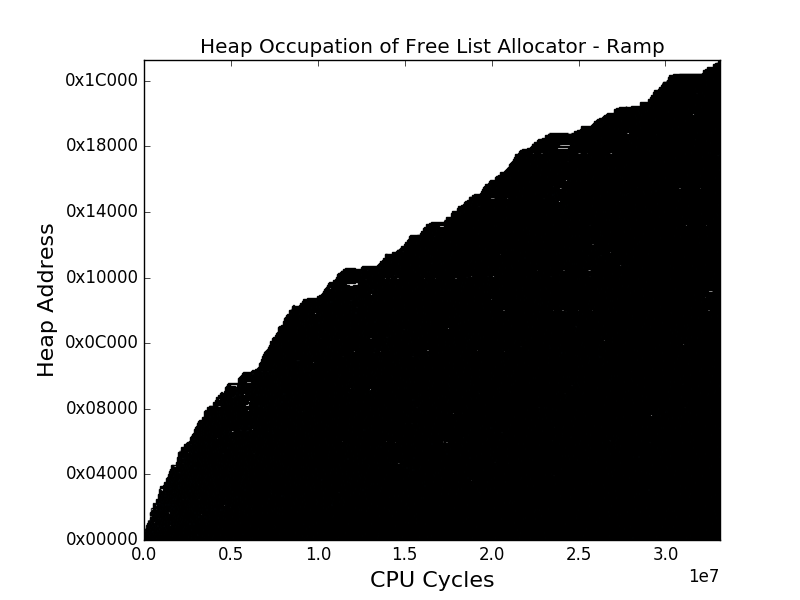
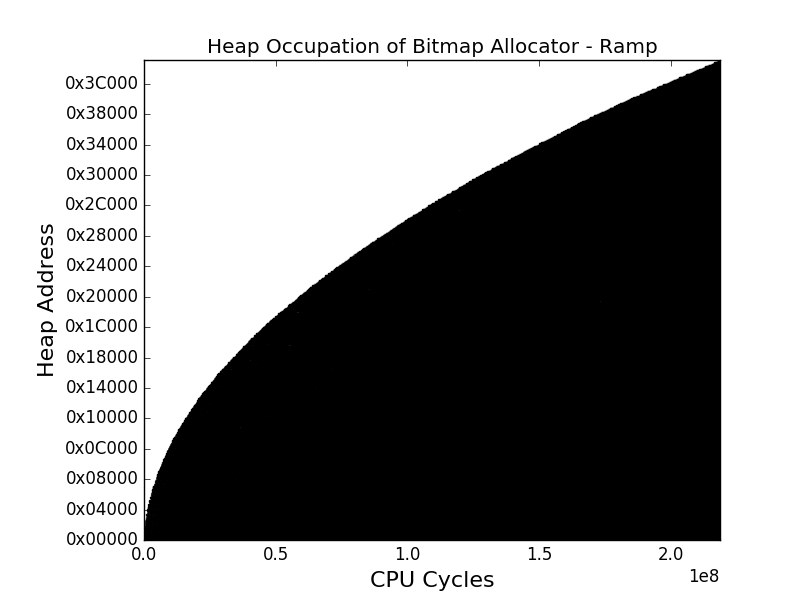


Figure 10: The free list, bitmap and buddy allocators’ behaviours during the ramp.

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 allocator’s performances. On the other hand, the free list only spent 17600 thousand cycles allocating blocks and had an LMF of 119 thousand bytes. The last two being significantly better than the bitmap’s and buddy’s values. To conclude, the free list is affected significantly negative by ramp-like allocation patterns.

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 and buddy allocators’ number of allocation cycles, were of catastrophic nature. With that said, it can be concluded that applications with ramp-like characteristics should not be combined with bitmaps.

The buddy allocator did, as the bitmap, not suffer much in terms of freeing cycles but rather in terms of allocation cycles and fragmentation. The buddy allocator’s allocation cycles were almost twice as many as those seen in the free list and it had an LMF of similar size to the bitmap’s. On the other hand, the amount of cycles spent freeing were significantly lower than those of the free list. In conclusion, the buddy allocator can compete with the free list in terms of execution cycles but not in terms of fragmentation and thus memory usage.

Interesting to note is the smoothness of the bitmap’s and the buddy allocator’s curves in Figure 10.B and Figure 10.C. This is due to the grain size of 256 bytes often being larger than required. If the grain size would have been smaller the ramps would have been coarser as with the free list’s ramp. The extra space in each block contributes to the total internal fragmentation. This internal fragmentation does, in turn, contributes to the total fragmentation as well as the memory footprint since these three metrics are dependent of each other.

### Studying Intense Memory Usage - Peaks

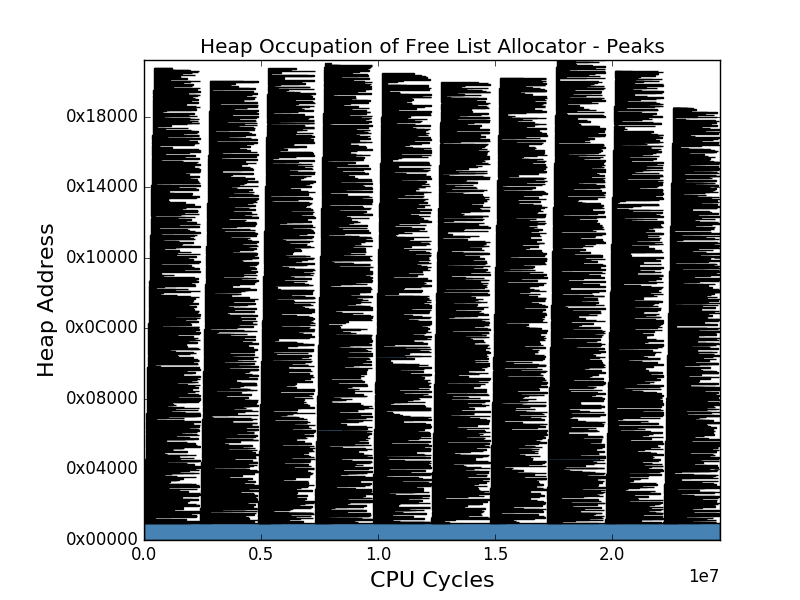
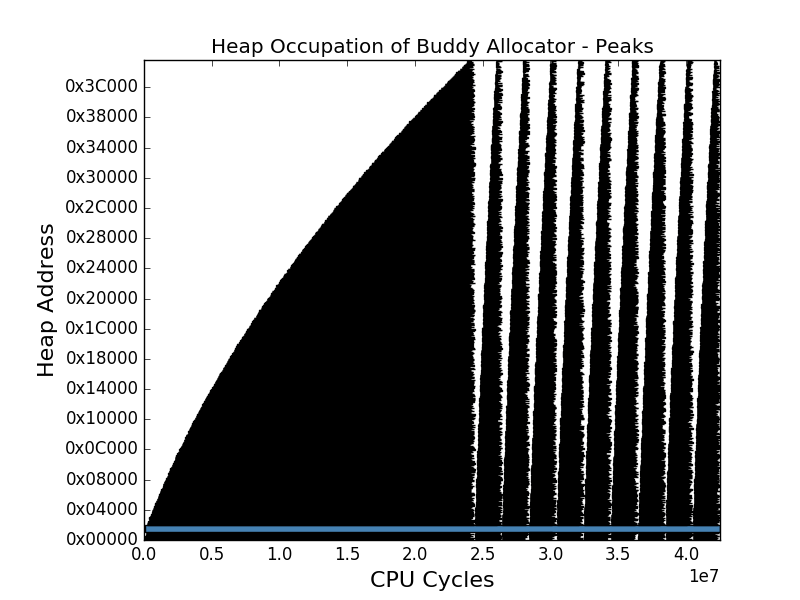
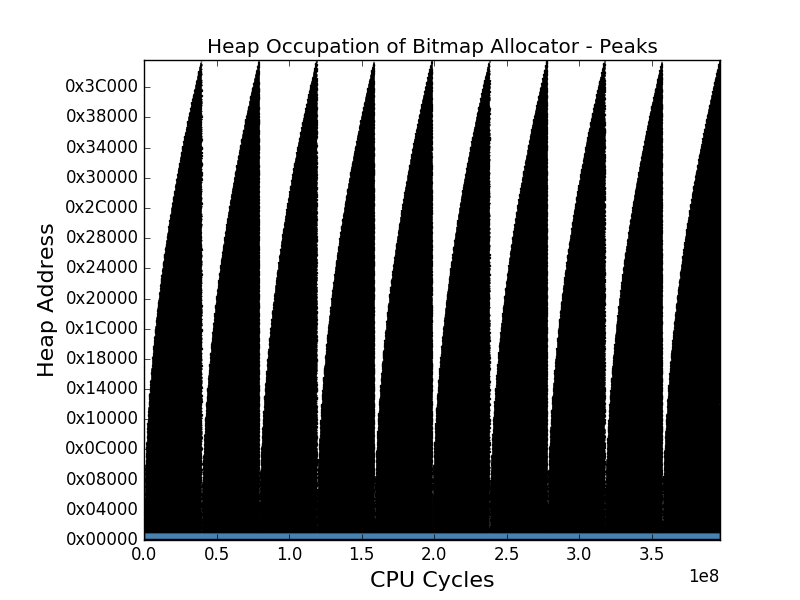
The peaks were the toughest of the access profiles for the allocators. The typical intense memory usage associated with peaks meant numerous allocation and freeing request. The behaviors of the free list, bitmap and buddy allocator during the peaks can be observed in Figure 11.

Figure 11: The free list, bitmap and buddy allocators’ behaviours during the peaks.

When facing the peaks, the free list kept a satisfactory low amount of 2250 thousand allocation cycles, about 16 times faster than the buddy allocator and significantly faster than the bitmap allocator. It also kept acquiring a lower total cycle count than its competitors. On the other hand, the free list allocator struggled with freeing, reaching 19100 thousand freeing cycles. During the peaks, the free list had an LMF of 111 thousand, which was the lowest LMF of our three candidate allocators’ LMFs. In conclusions, the free list seemed to be the most suitable allocator when dealing with peaks.

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 allocator’s LMF but more than the double of the free list’s LMF. The total amount of cycles spent allocating data was 397 million cycles, almost a factor of 10 times the buddy allocator total cycles. The bitmap is thus not the optimal allocator for peak behavior in the context of our memory properties.

When stressed by the peaks, the buddy allocator struggled more than during earlier access profiles. The buddy allocator spent 36300 and 2940 thousand cycles allocating respectively freeing. Its LMF kept a size similar to that of the bitmap and about twice as large as the free list’s LMF. In other words, it only 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 allocator’s difference in time when handling various peaks. The first peak took markedly more time than the following peaks which took 2.0 million cycles comparable to the free list allocator’s peaks of 2.5 million cycles. The reason behind the slowness of the first peak was the splitting of the largest block. This block had to be recursively split until the acquired block size matched the requested object size. This variation in peak duration significates that the buddy allocator might be faster than the free list allocator during long runs with peak like behavior. The speed up in peak handling was a consequence of the deferred coalescing described in Section 3.4.

### Facing the Real World – A JSON Parser

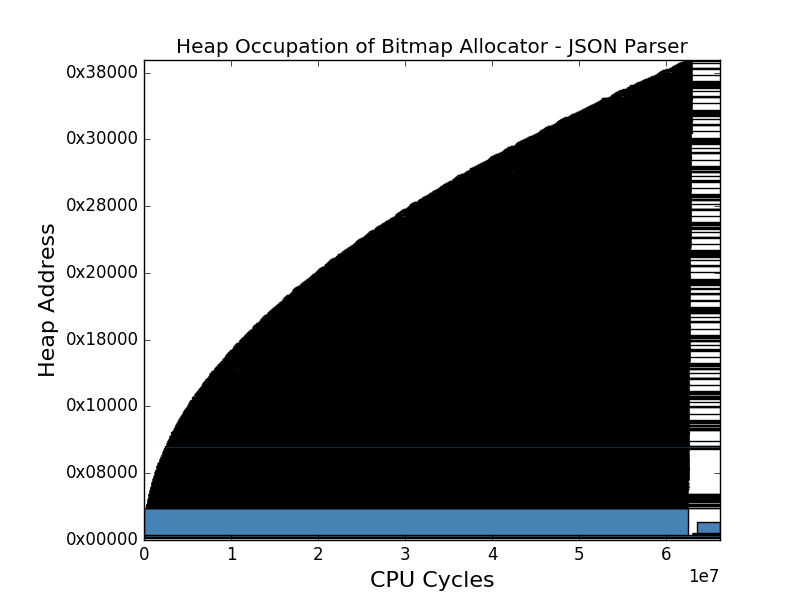
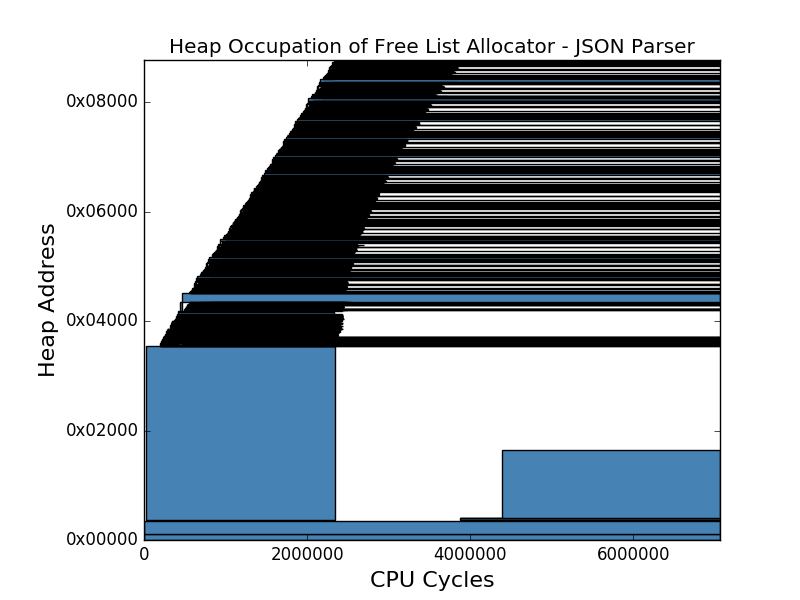
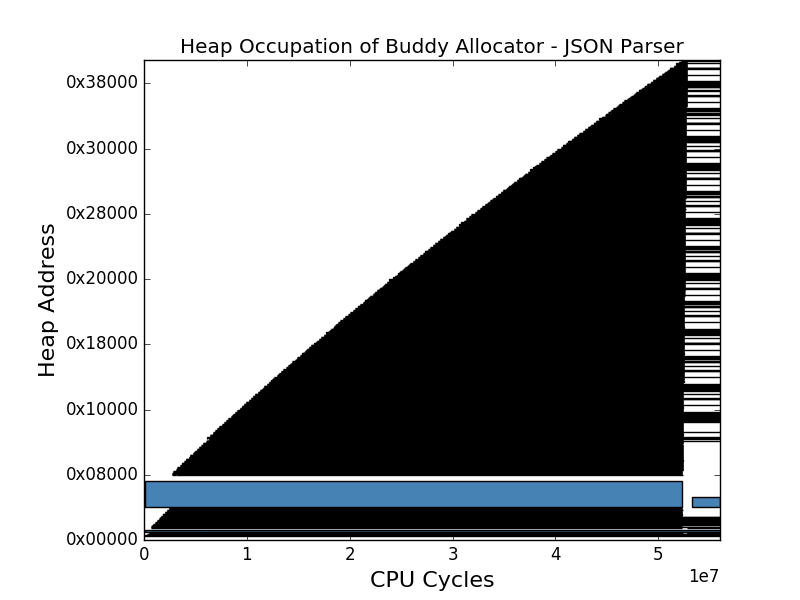
The application, which aimed at replicating a real-life application, yielded interesting results. The allocators’ responses to the applications requests can be observed in Figure 12. By observing these graphs, one can see that the application has characteristics similar to that of the synthetic ramp.

Figure 12: The free list, bitmap and buddy allocators’ behaviours during the JSON parsing application.

The free list handled the JSON Parser’s allocation requests with ease. The allocator only needed a total of 396 thousand cycles in the malloc routine. The free list also exhibited a favorably low LMF barely reaching 36 thousand bytes. On the other hand, the free list spent 1500 thousand cycles freeing, being almost three times as much as the buddy system did. This was expected since the JSON Parser exhibited ramp characteristics and thus should show similarities between its malloc-free cycles ratio and that observed during the synthetic ramp. Conclusively, the free list is fast in terms of malloc cycles but may be slow in terms of freeing cycles.

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 at handling the requests of the JSON Parser compared to the free list and buddy allocator. On top of this, it had a large LMF of 236 thousand bytes compared to the free list’s LMF of only 36 thousand bytes. In conclusion, the bitmap is inappropriate for our memory sizes since it is too slow and to memory consuming.

The buddy allocator spent 50 million cycles allocating blocks but only 600 thousand cycles freeing blocks. This low amount of freeing cycles is a result of the buddy allocator having a freeing time complexity of O(1). This complexity appears when the buddy allocator does not need to merge or split since it then only needs to mark a block as free in the bitmap. Since the buddy allocator implements a deferred coalescing strategy, it will only merge and split in the malloc routine while keeping the freeing quick. Contrary to this good result, the buddy allocator had a considerably larger LMF than the free list. This significates that the buddy allocator is more likely to throw a heap overflow and thus, is less reliable.

Figure 13: The execution time of the free list, bitmap and buddy system when combined with the plateau, ramp, peaks and JSON Parser.

Figure 14: The memory usage of the free list, bitmap and buddy system when combined with the plateau, ramp, peaks and JSON Parser.

### Discussion

By looking at Figure 13 we can clearly see that the free list allocator is faster than the bitmap and buddy allocator. From a performance perspective the free list allocator was thus clearly better in all cases except during the peaks. During the peaks, the buddy allocator could be a potential 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 allocator instead of the free list. However, this would require the object sizes to be allocated in a manner reusing free blocks with minimal splitting and merging since those could delay the allocation too much.

From a perspective regarding effectiveness of memory usage the free list allocator was an obvious winner as well. This can be seen in Figure 14 where the free list’s largest memory footprint was always smaller than that of the bitmap and the buddy allocator. In other words, the experiments imply that the free list allocator’s critical points will be of less danger than those of the bitmap and buddy allocator. In conclusion, usage of free list allocators, rather than other candidates, will contribute to a more reliable system since the free list version proved to be the best at avoiding heap overflows and generally was faster.

When adjusting the minimal grain size of the bitmap and buddy allocator we discovered a trade-off. This trade-off stated that the cycles spent in malloc and free increased when the minimal grain size 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 allocator reasonable.

A big reason for the buddy and bitmap allocator’s large memory footprints were the relationships between their grain sizes and the requested object sizes. The object 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 grain size of the buddy and the bitmap allocator were set to 256 bytes while the objects of medium and small sizes never would exceed that value. The grain size was not decreased since a trade-off between the grain size and cycle counts had been discovered.

Since these 2 allocators only allocated in multiples of 256 bytes this meant that a small request would result in 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 not be utilized. 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 allocators is that the buddy allocator would use deferred coalescing while the free list allocator wouldn’t. This optimization was done since we discovered that the buddy allocator was completely unusable, in terms of execution time, when splitting and merging aggressively.

By changing the buddy allocator’s strategy, it was given a chance to compete with the free list and bitmap allocator. Since the free list allocator didn’t benefit by this optimization, this further strengthens our decision to choose the free list as the most favourable allocator. 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.

Table 2: Results from synthetic and application profiling.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Application Profile** | **Allocator** | **Malloc**  **cycles** | **Free**  **cycles** | **Application**  **cycles** | **Total**  **cycles** | **LMF** | **Fragmentation at LMF** |
| **Plateau** | **Free List** | 2 260 472 | 1 641 896 | 3 294 064 | 7 196 432 | 5888 | 27.649% |
| **Plateau** | **Bitmap** | 19 964 032 | 2 609 494 | 3 294 489 | 25 868 015 | 8120 | 50.493% |
| **Plateau** | **Buddy** | 7 200 209 | 2 933 358 | 3 298 701 | 13 432 268 | 7152 | 40.436% |
| **Ramp** | **Free List** | 17 359 953 | 12 612 828 | 3 166 199 | 33 138 980 | 119894 | 17.164% |
| **Ramp** | **Bitmap** | 213 226 118 | 2 346 166 | 3 164 519 | 218 736 803 | 258586 | 61.493% |
| **Ramp** | **Buddy** | 34 028 366 | 2 642 408 | 3 165 352 | 39 836 126 | 258582 | 61.492% |
| **Peaks** | **Free List** | 2 257 091 | 19 085 922 | 3 284 664 | 24 627 677 | 111484 | 65.373% |
| **Peaks** | **Bitmap** | 391 214 122 | 2 606 520 | 3 282 325 | 397 102 967 | 260577 | 67.744% |
| **Peaks** | **Buddy** | 36 298 554 | 2 938 646 | 3 280 224 | 42 517 424 | 260573 | 67.743% |
| **JSON Parsing** | **Free List** | 395 744 | 1 469 716 | 5 210 510 | 7 075 970 | 35944 | 78.108% |
| **JSON Parsing** | **Bitmap** | 60 475 720 | 485 662 | 5 210 013 | 66 171 395 | 235524 | 96.659% |
| **JSON Parsing** | **Buddy** | 50 231 938 | 565 385 | 5 210 435 | 56 007 758 | 241160 | 96.737% |

## Multi-Bank Memory Management Applying Various Strategies

In the previous section it was concluded that the free list allocator would be the most suitable allocator for Evaderis architectures. During the evaluation of the memory manager’s potential strategies the degree work was thus chosen to be limited to this allocator type.

After simulating all the strategy-architecture combinations we discovered that they expressed very similar LMF. The fragmentation is thus of greater importance and was thus what was analyzed.

The results show that there was no superior memory manager strategy but rather that some strategies were better than others depending on the circumstances. All the results acquired can be observed in Table 3 at the end of the section. Additionally, all figures are to be found at a larger scale in appendix A.

### A Two Bank Architecture

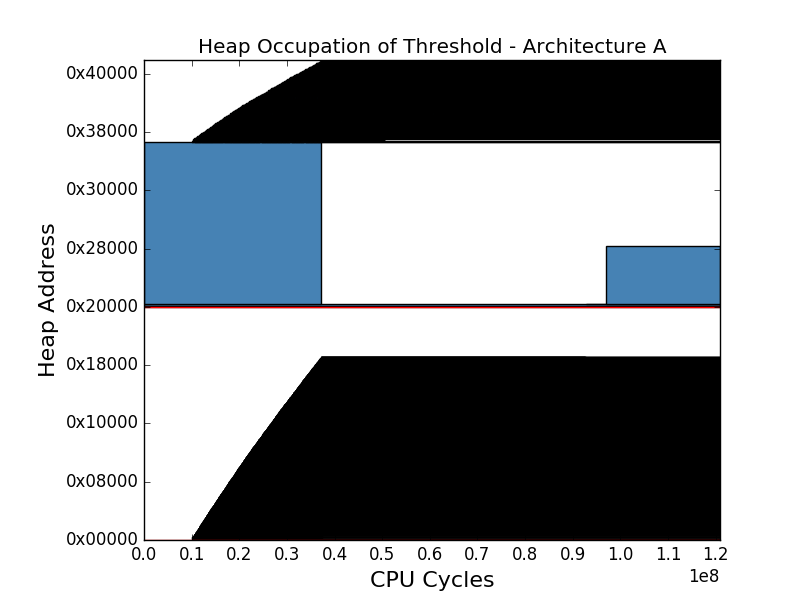
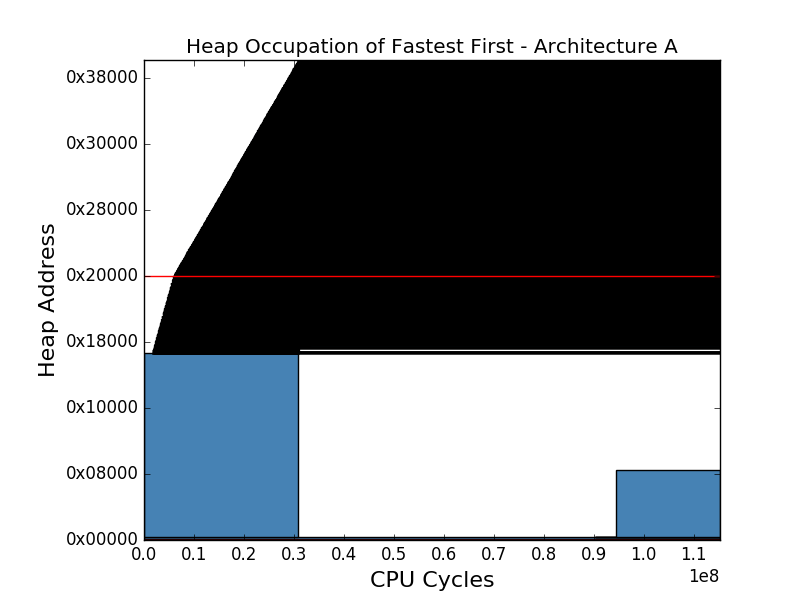
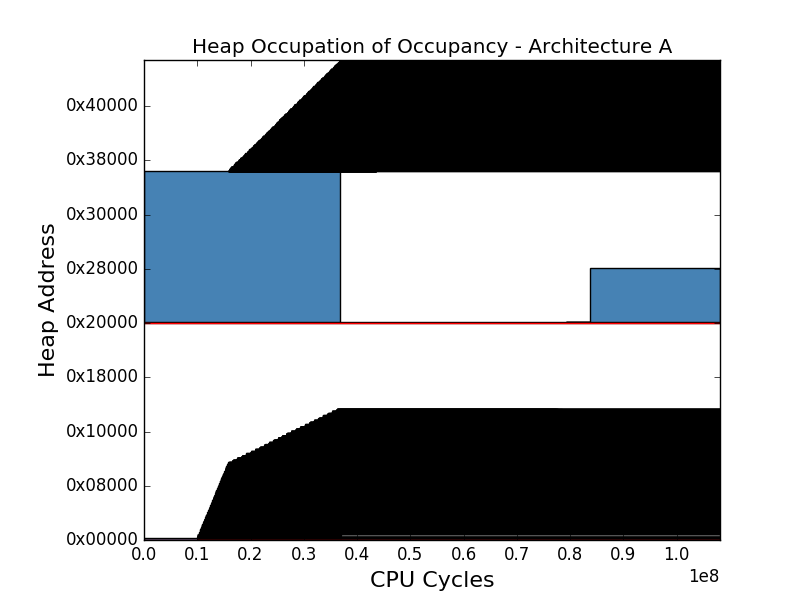
The first architecture had two banks. The first bank being fast and the second one slower but larger. When matching the architecture with the Threshold, Fastest First and Occupancy strategies we acquired different access profiles. These access profiles can be seen in Figure 15. The red line is the border between the first and second memory bank.

Figure 15: The Threshold, Fastest First and Occupancy strategies’ behaviours on memory architecture A.

The Threshold strategy spent 7 million and 56 million cycles in the malloc respective free routine. The large amount of freeing cycles was a sign of a high amount of coalescing taking place. It had a relatively high fragmentation of 82.0%, similar to the 81.8% of Occupancy but significantly different from the 59.8% of Fastest First. Since they have similarly sized LMFs, 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 intensive than memory intensive.

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.

### Modifying the Latencies

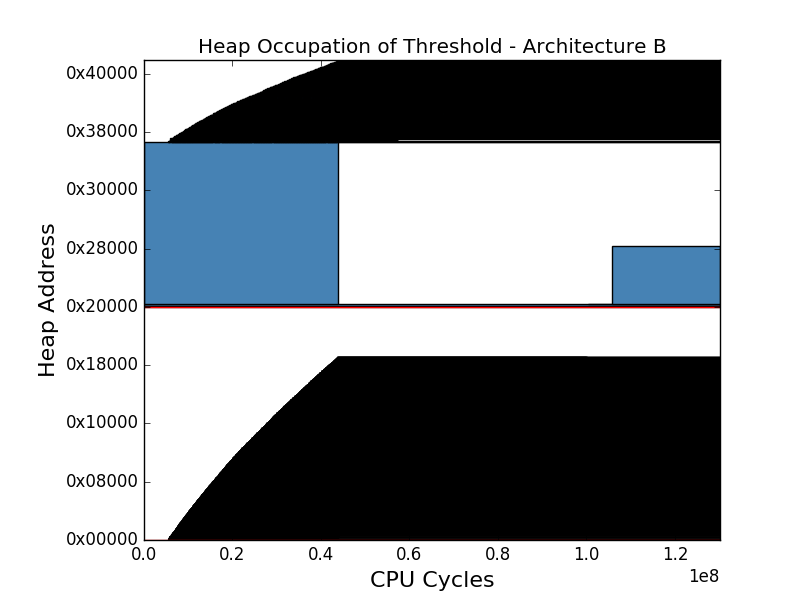
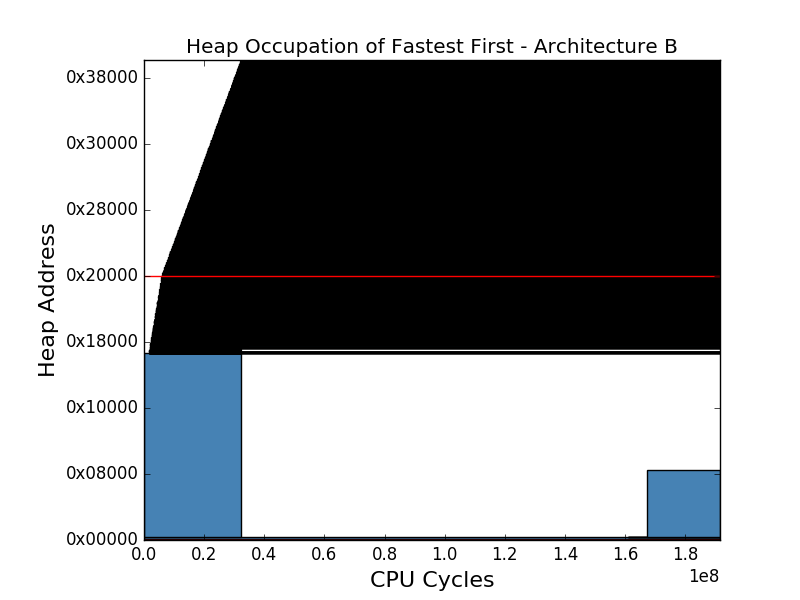
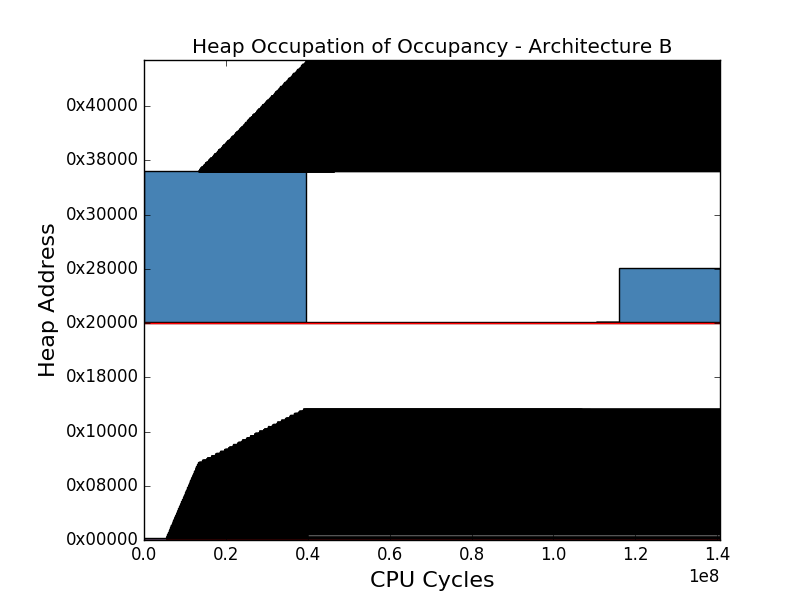
The second architecture was very similar to the first. The only difference being the different read and write latencies. As a consequence of this, the graphs in Figure 15 and Figure 16 are very similar, only differing with a stretch in x-direction.

Figure 16: The Threshold, Fastest First and Occupancy strategies’ behaviours on memory architecture B.

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 190 and 140 million cycles respectively. 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 application 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 resulted in 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.

### Adding an Additional Memory Bank

The last memory architecture differed from the previous two by having 3 memory banks instead of 2. Thus, its graphs seen in Figure 17 contain 2 line delimiters rather than 1. The memory banks are, as in Figure 15 and Figure 16, ordered by speed starting with the fastest one at 0x0 and ending with the slowest at 0x20000.

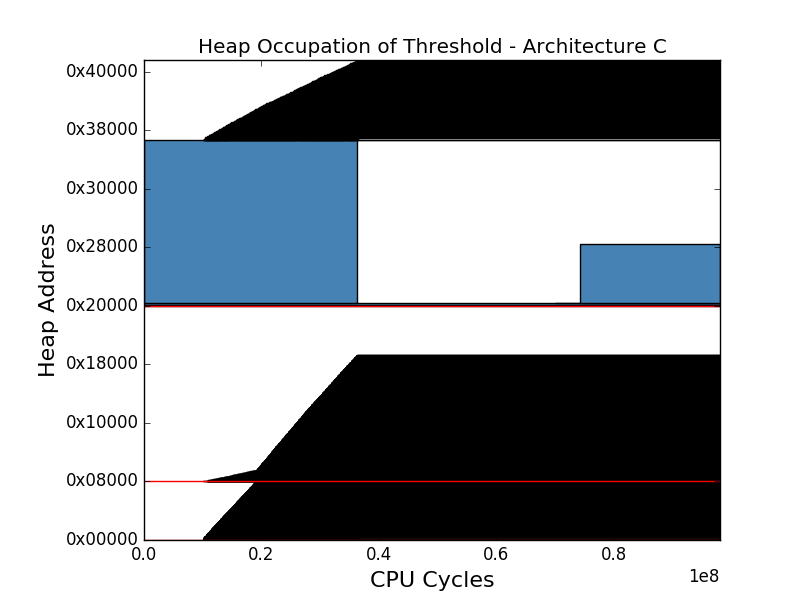
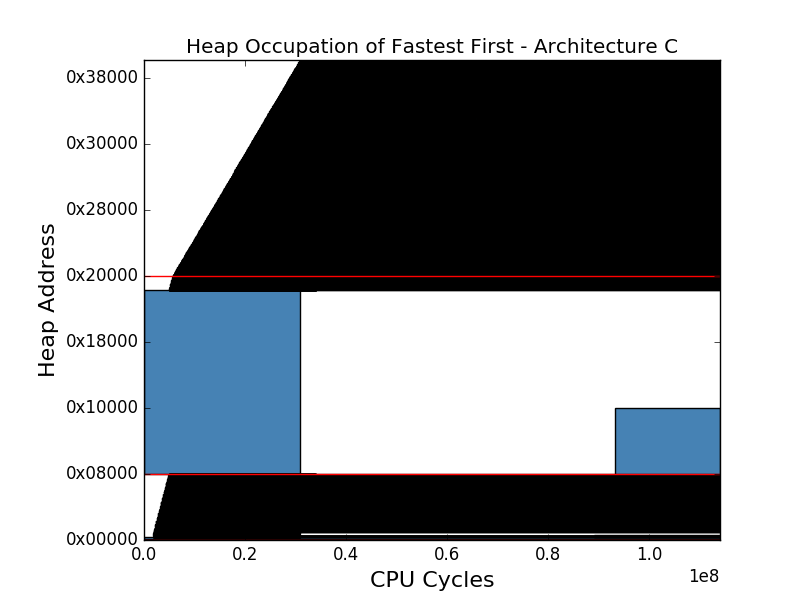
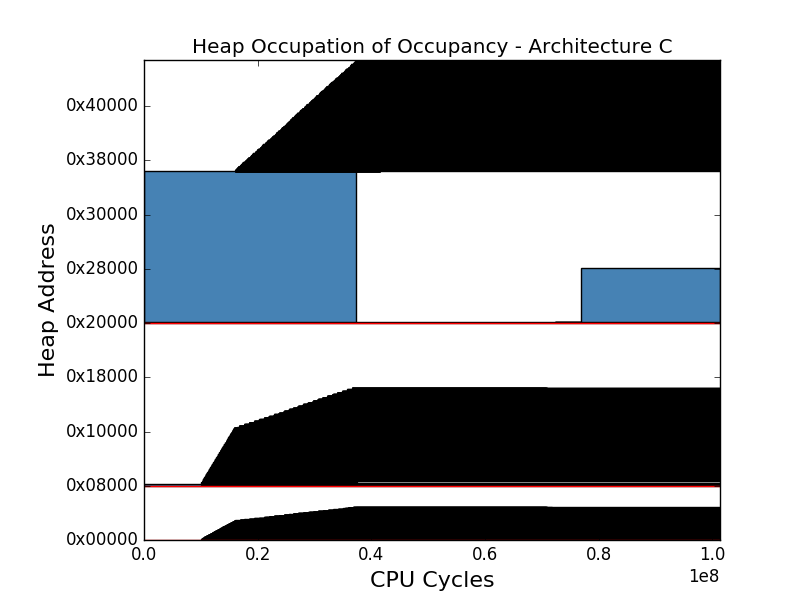


Figure 17: The Threshold, Fastest First and Occupancy strategies’ behaviours on memory architecture C.

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 application cycles and fragmentation. The strategy merely spent 49 million outside the memory routines compared to the two candidate strategies which spent almost 60 million application cycles. The Fastest First strategy has thus been the fastest in terms of application cycles during every architecture. We can thus conclude that the Fastest First strategy could potentially be the most suited strategy for compute intensive 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 application cycles. The strategy realized 59 million application cycles compared to 57 million and 48 million for the Threshold and Fastest First strategies respectively. De facto, the Occupancy strategy had the most application cycles during every architecture, rendering it the least suited for compute intense applications.

### Discussion

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 possibly be to use multiple strategies and pair them with the memory types they would work best with.

Another interesting observation was that the Fastest First strategy perceived the heaps as one continuous large heap. This enabled the allocator to search for a fitting block in all heaps and thus acquiring a better chance of finding a large enough free block before deciding to increase the memory footprint. In comparison, other strategies only had one heap to look through each allocation. An additional consequence of this remark was that the fragmentation at the LMF was low through all runs, which can be observed in Table 3.

It was also noted that Fastest First always had a fewer amount of application cycles than its competitors. Consequently, it might be the favourable strategy when dealing with compute-intensive 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. It can thus be concluded that further studies are needed before deciding which strategy or strategies to apply under which circumstances in future NVRAM-based embedded systems.

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

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 the initial section followed by the final chapter concerning further work.

## Conclusion

This thesis the problem of allocation on heterogeneous memory in the context of Evaderis architectures. The research was limited to three allocator types, three allocation strategies, three memory architectures as well as the memory properties latencies and sizes provided by industrial partners.

The three allocators were analyzed when combined with various access profiles running on a single bank memory architecture. This provided insight into their behaviors and led to the conclusion that the free list allocator was, in general, the most suitable allocator for Evaderis architectures in terms of memory usage and execution time. This was a consequence of the discovery of the tradeoff between grain size and cycle count concerning the bitmap and buddy allocator.

The threshold discovered stated that the execution speed was dependent on the number of grains which in turn was dependent on the grain size. By considering this trade off it was concluded that the buddy and bitmap allocators does not work well when memory is big and grain size small.

An additional conclusion drawn was the buddy system’s potential to be quicker than the free list during peaks. This was due to its shorter peaks after the first peak, resulting from absence of a need to split blocks. This was due to the smallest blocks being large enough for almost all requests.

After concluding the general superiority of the free list allocator, multi bank architectures were designed to represent heterogenous memory. These were constituted of memory banks with various read and write latencies as well as sizes. The three architectures were combined with the three different strategies which then were evaluated.

The evaluation of the strategies suggested that there were no superior strategy. Instead different strategies were optimal depending on the circumstances. The experiments suggested that compute intensive applications should benefit the most from the Fastest First strategy and least from the Occupancy strategy.

The experiments also suggested that the Fastest First in general was slow but had a low memory usage because of its low fragmentation. It also suggested that the occupancy strategy might have lower fragmentation with the addition of more memory banks.

There was two main conclusions of the study. The first being that the free list allocator was the most favorable single-bank allocator. The second that further experiments should be conducted before definitively deciding which multi-bank strategy to use when as well as where.

## Future Work

Continuation of this thesis project can be performed in many ways. Obvious extensions include experimenting with new single-bank allocator techniques, multi-bank strategies and/or memory architectures. One could also feed the application with other JSON objects or even study other application. For instance, one might want to study memory intensive or compute intensive applications.

Concerning strategies, one could try to implement more complex strategies basing memory bank choices on block usage. For instance, if a memory manager noticed that the application often read and wrote to medium sized objects, it could prioritize putting future medium sized objects in the fastest memory bank.

Other possible future work include evaluating different free list implementation. For instance, address ordered free lists could be evaluated against size ordered free lists since both of these have their advantages and disadvantages. One could also experiment with free list policies or implementation optimizations.

For instance, if ten blocks were regularly allocated and then freed consecutively, an intelligent implementation could ensure that they would be allocated consecutively in memory. This would minimize fragmentation when the blocks would be freed since all the freed space could be coalesced back into one block.

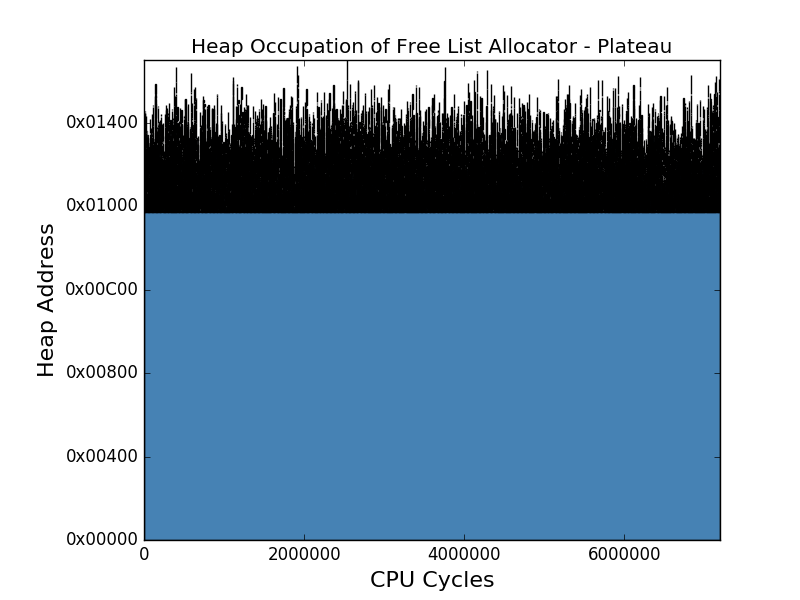
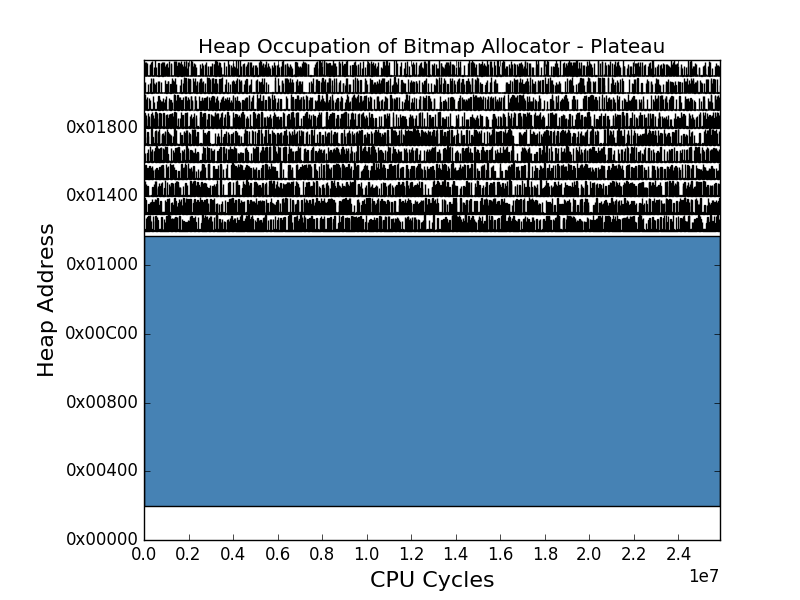
Additional work could be to analyze the different single-bank allocators against memories with asymmetrical latencies. This was not performed in this thesis since the free list performed markedly better than the other allocators and that it was thus believed to excel its competitors on asymmetrical memory banks as well. What could still be of interest, in the context of this report, could be to evaluate various free list implementations on asymmetrical memory banks.

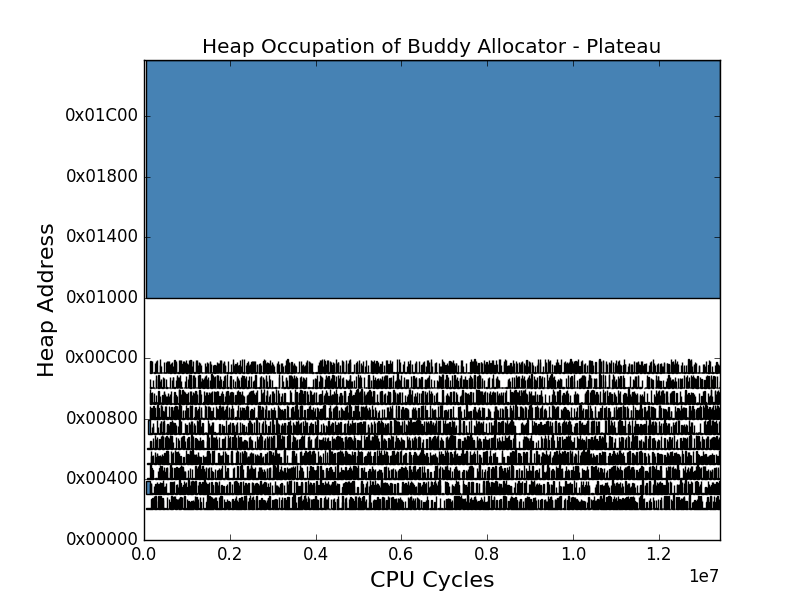
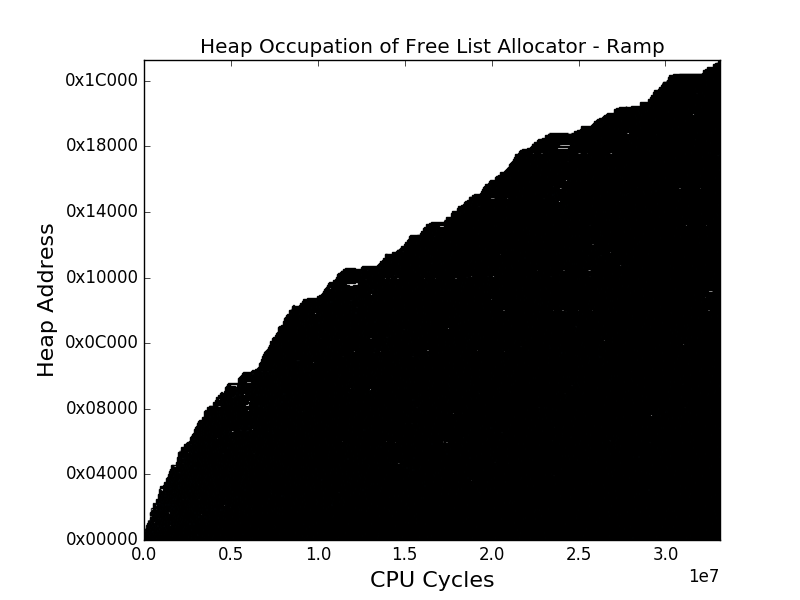
# References

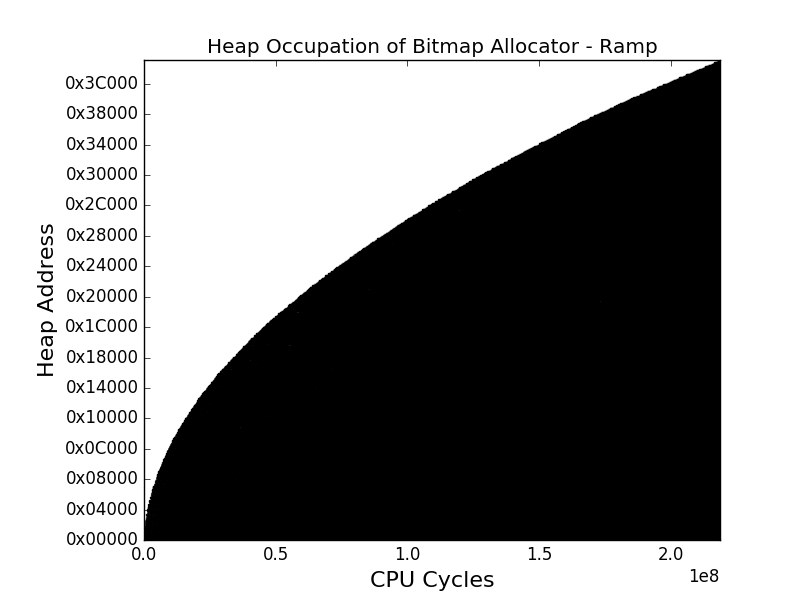
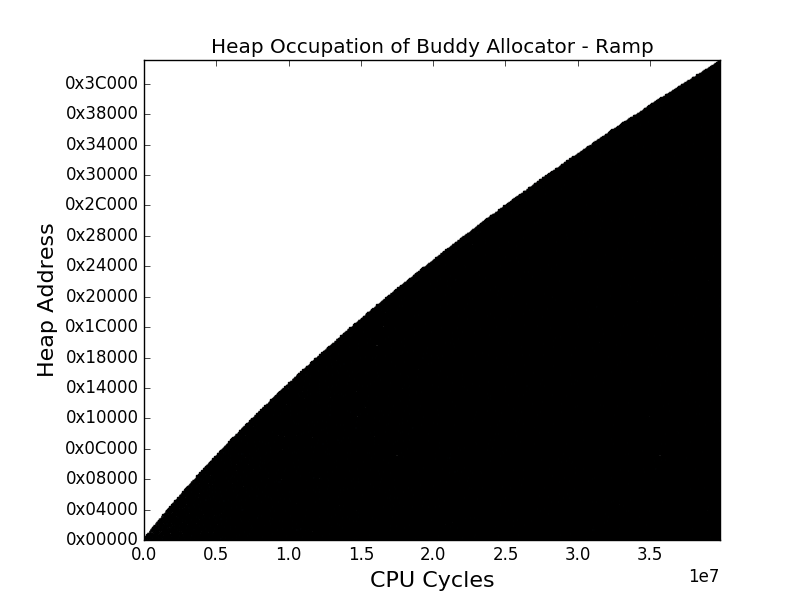
|  |  |
| --- | --- |
| [1] | Spintec, "Overview," Spintec, 2015. [Online]. Available: http://www.spintec.fr/organization/. [Accessed 11 11 2017]. |
| [2] | Evaderis, "About," Evaderis, 2017. [Online]. Available: http://www.evaderis.com/about-evaderis/. [Accessed 11 11 2017]. |
| [3] | C. Layer, L. Becker, 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] | IEEE, "A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems," *IEEE Transactions on Parallel and Distributed Systems,* vol. 27, no. 5, pp. 1537-1550, 2015. |
| [5] | E. Harari, "ELECTRICALLY ERASABLE NON-VOLATILE SEMICONDUCTOR MEMORY". United States of America Patent 770346, 22 02 1977. |
| [6] | Dhirubhai Ambani Institute of Information & Communication Technology, "MRAM," [Online]. Available: http://courses.daiict.ac.in/pluginfile.php/5827/mod\_resource/ content/0/presentation/pres\_N\_MRAM.pdf. [Accessed 26 12 2017]. |
| [7] | H.-S. Wong, S. W. Fong and C. M. Neumann, "Phase-Change Memory—Towards a Storage-Class Memory," *IEEE Transactions on Electron Devices,* vol. 64, no. 11, pp. 4374 - 4385, 2017. |
| [8] | R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, Arpaci-Dusseau Books, March, 2015 (Version 0.90). |
| [9] | 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. |
| [10] | 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. |
| [11] | T. Grötker, "Fundamentals of Systemc," in *System Design with SystemC*, Hingham, Kluwer Academic Publishers, 2002, pp. 11-40. |
| [12] | S. Pelley, P. Chen and T. Wenisch, "Memory Persistency," *SIGARCH Comput. Archit. News,* vol. 42, no. 3, pp. 265-276, 2014. |

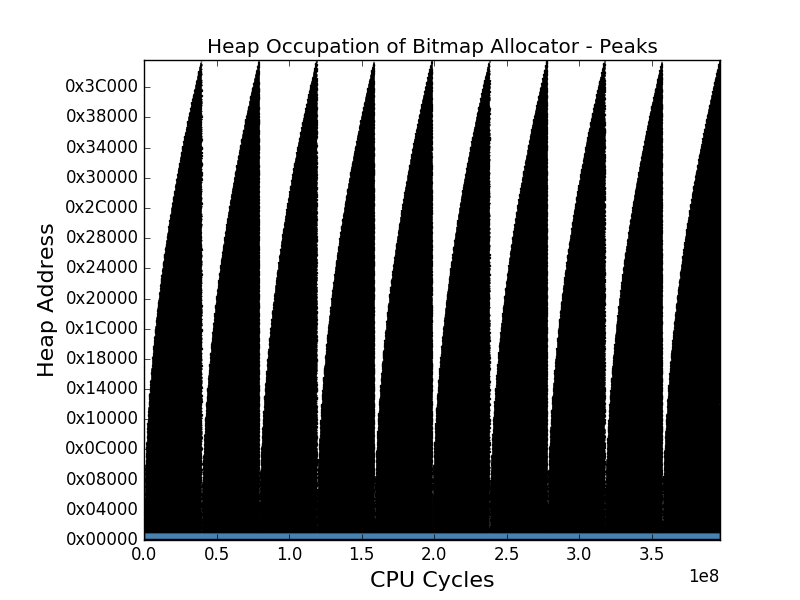
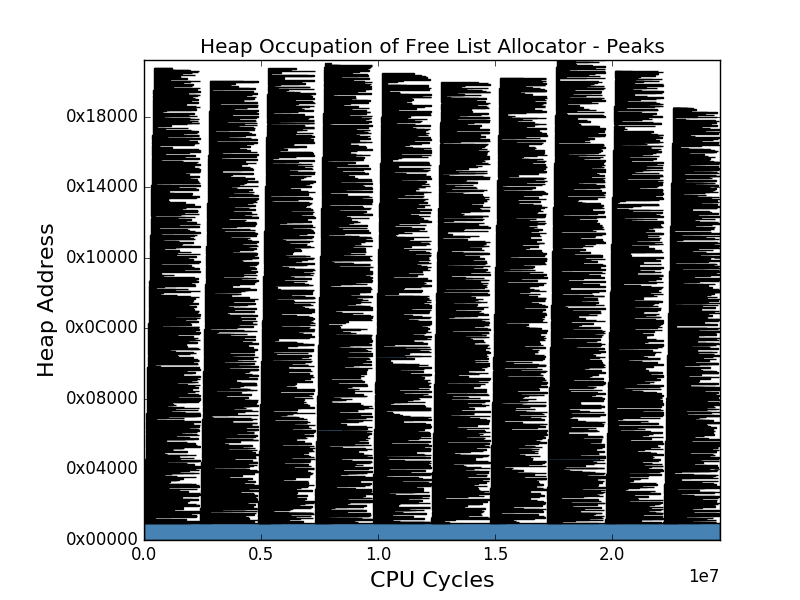
Appendix A

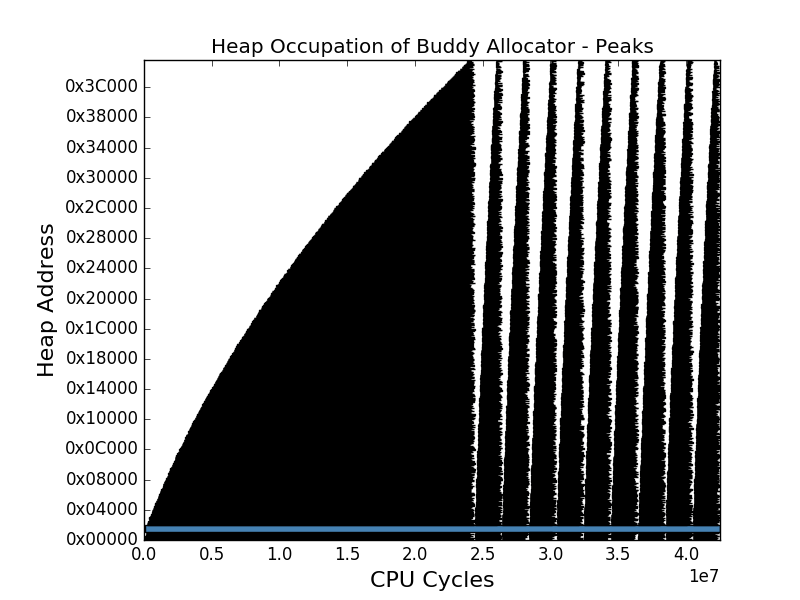
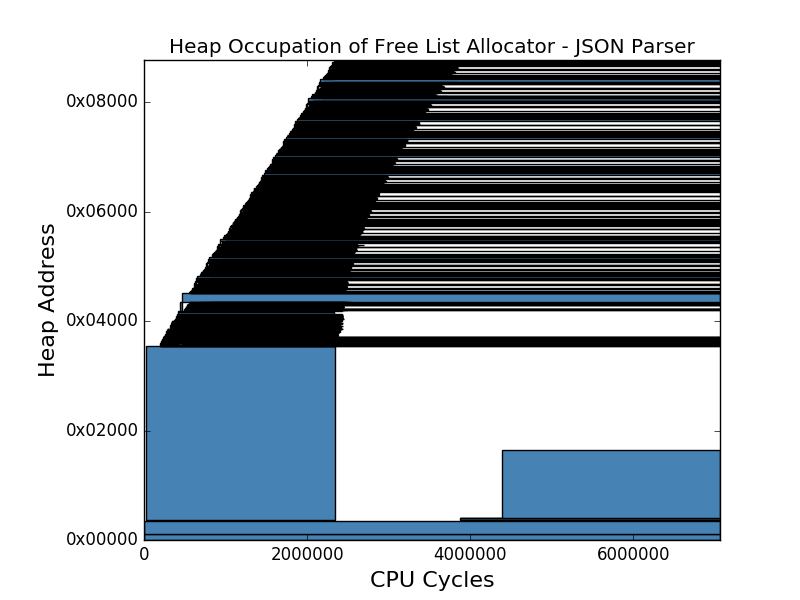
This appendix contains all the figures from the Results chapter at a larger scale. All of the figures are ordered in the order they appear in the document.

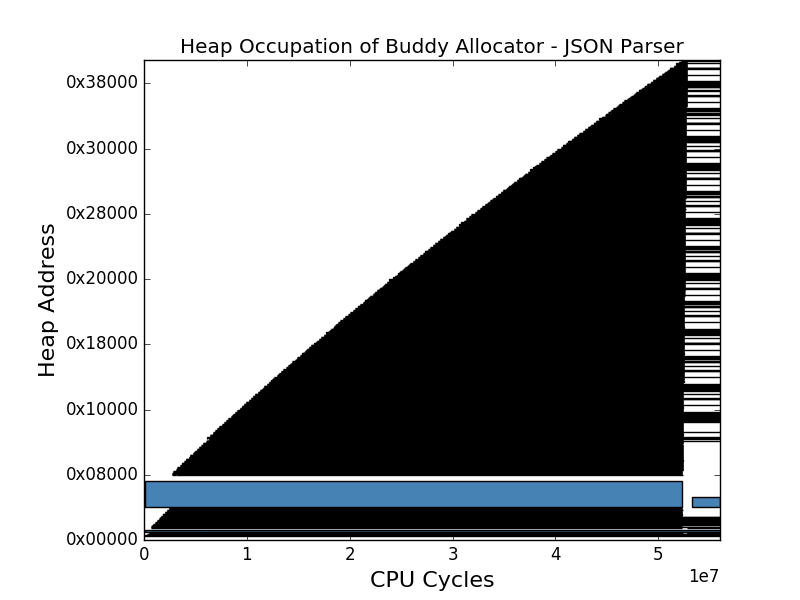
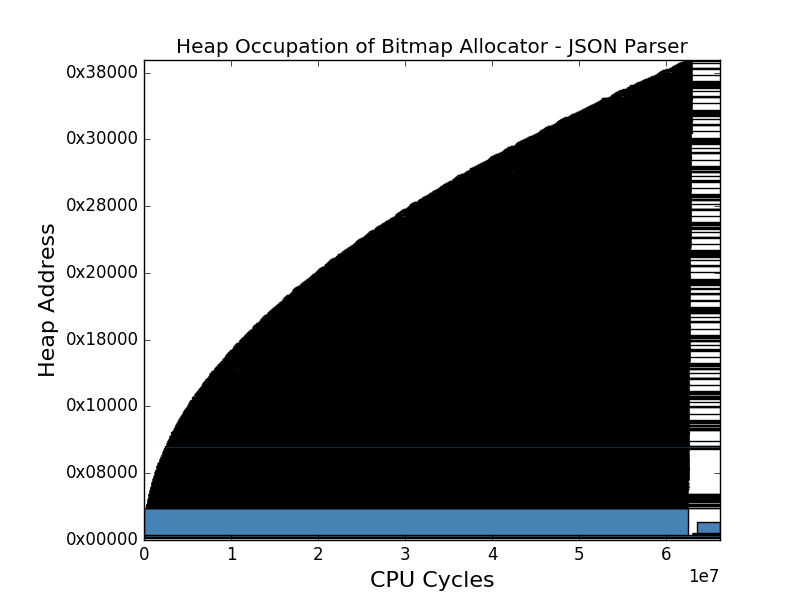


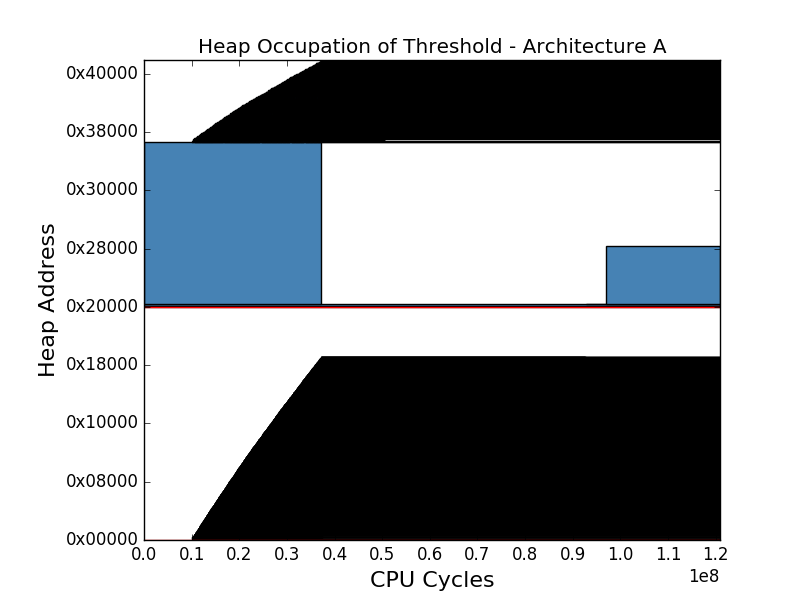
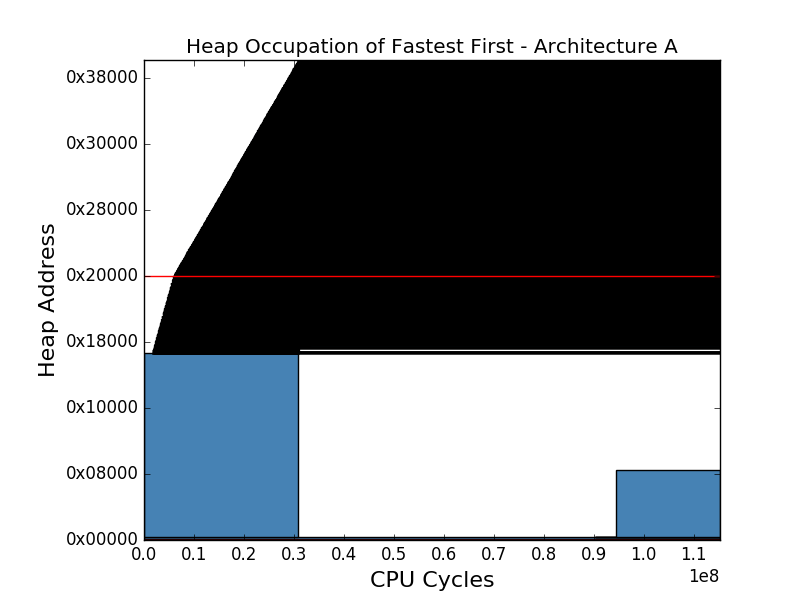


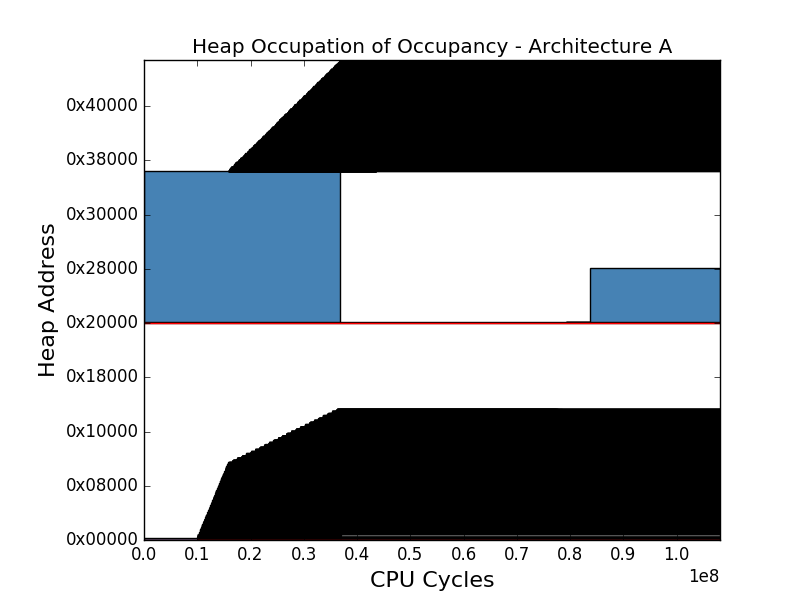
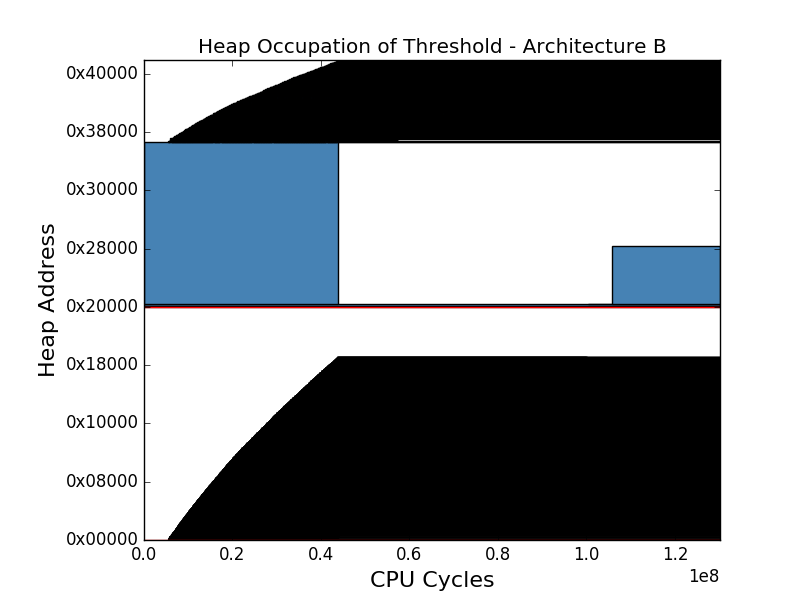


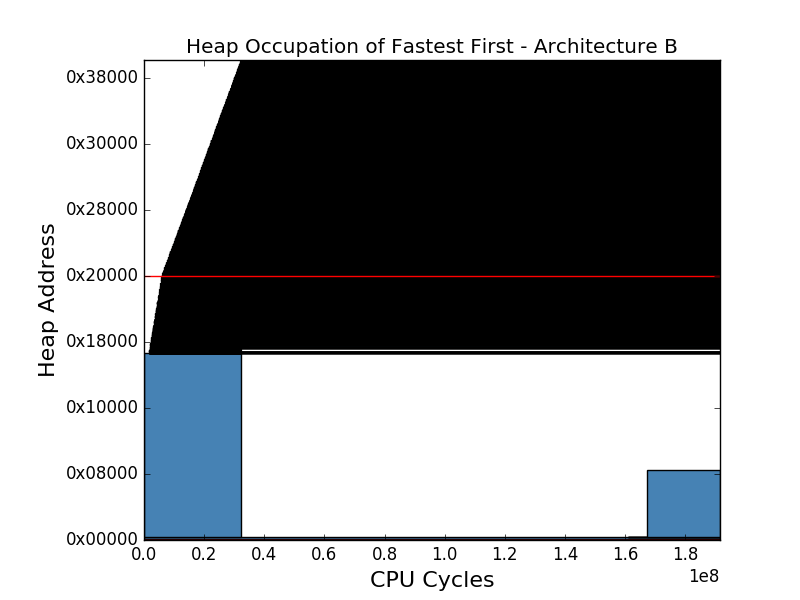
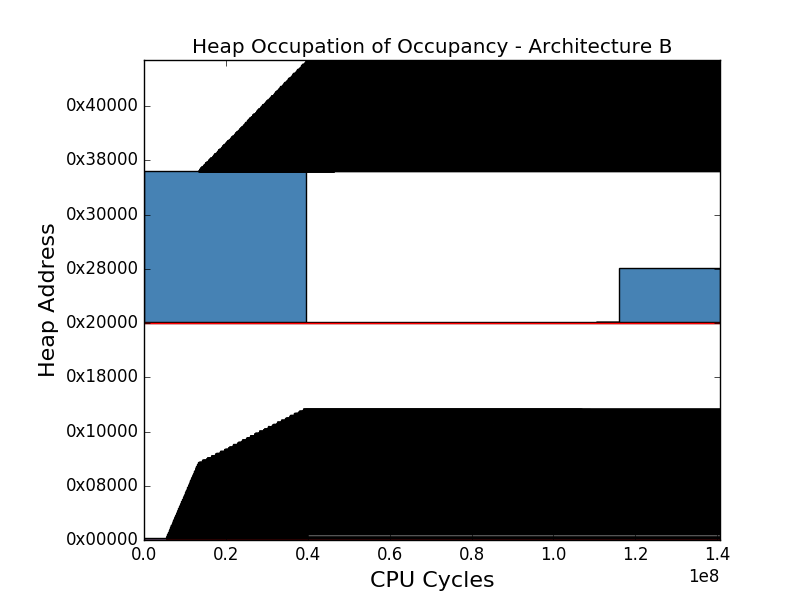


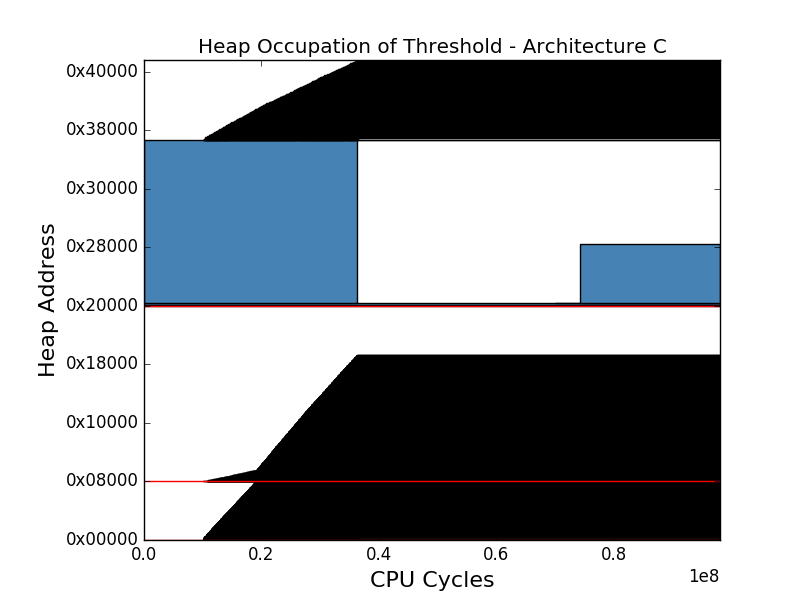
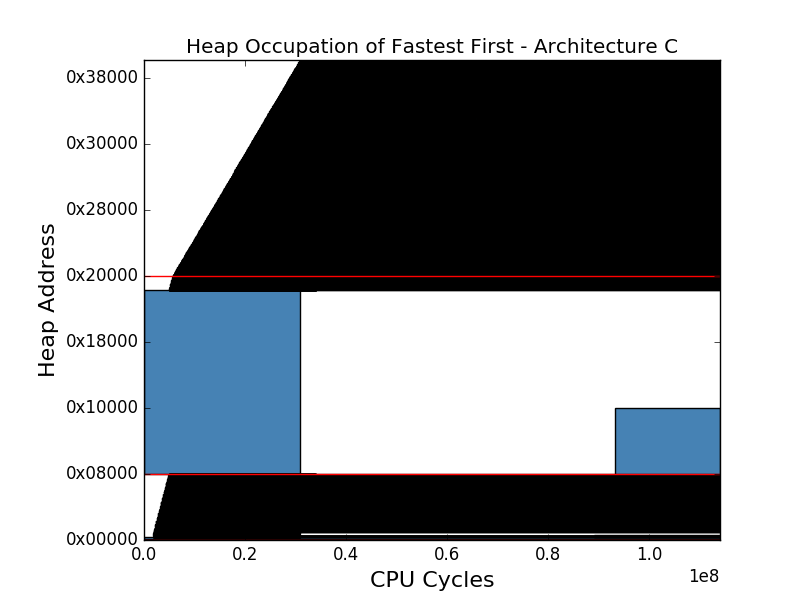


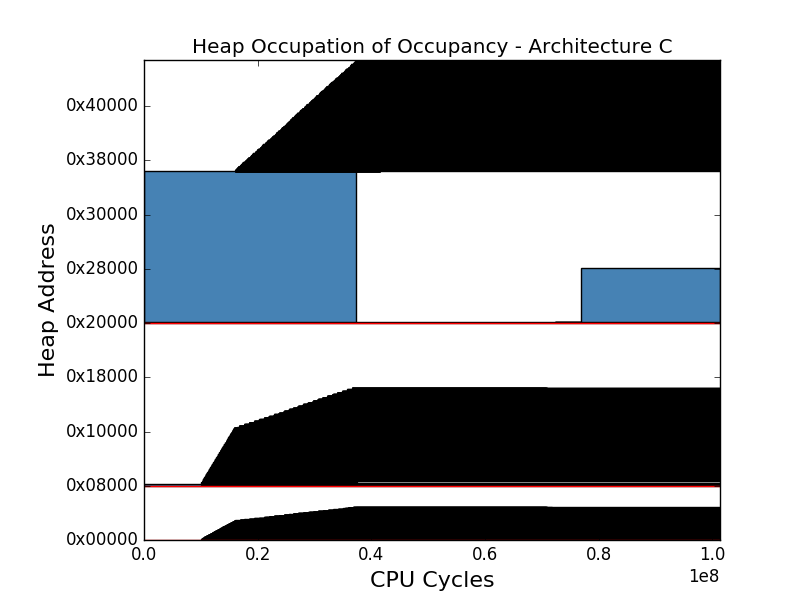












1. Source code can be found at <https://github.com/kgabis/parson> [↑](#footnote-ref-1)
2. See <https://people.kth.se/~thpeter/DynamicMemoryAllocation/walking_dead.json>

   and <https://people.kth.se/~thpeter/DynamicMemoryAllocation/walking_dead_short.json> [↑](#footnote-ref-2)