Node-oriented dynamic memory management for real-time systems on ccNUMA architecture systems

Seyeon Kim

Doctor of Philosophy

University of York
Department of Computer Science

April 2013

Abstract

Since the 1960s, most operating systems and programming languages have been able to use dynamic memory allocation and deallocation. Although memory allocation has always required explicit interaction with an allocator, deallocation can be either explicit or implicit. Surprisingly, even though memory allocation/deallocation algorithms have been studied extensively over the last five decades, limited attention has been focused on the real-time properties. Most algorithms are general-purpose and do not satisfy the requirements of real-time systems. Furthermore, the few allocators supporting real-time systems do not scale well on multiprocessors. The increasing demand for high-performance computational processing has resulted in the trend of having many cores. ccNUMA architecture systems are part of this trend and provide a systematic scalable design. This thesis contends that current memory allocators for Operating Systems that support cc-NUMA architecture are not appropriate for real-time applications. We further contend that those real-time allocators that have been proposed in the literature are not cc-NUMA aware. The thesis proposes and implements (a prototype of) a new NUMA-aware dynamic memory allocation algorithm for use in soft real-time systems. We study the behaviour of our new allocation algorithm in comparison with related allocators both theoretically and practically.

Contents

Abstract iii

List of figures vi

List of tables viii

Acknowledgements xiii

Declaration xv

1 Introduction 1

1.1 Motivation 2

1.2 Hypothesis 4

1.3 Thesis Objectives 4

1.4 Organisation of the Thesis 6

2 Dynamic Memory Managements 9

2.1 Introduction 9

2.2 Fundamental Issues 16

2.3 Memory Management Algorithms 24

2.4 Summary 44

3 nMART: A ccNUMA-aware Dynamic Storage Allocation Algorithm 45

3.1 Design Principles 46

3.2 An Overview of nMART 51

3.3 Kernel-level Node-based Memory Management 52

3.4 User-Level Memory Management Algorithms and their Implementation 64

3.5 Summary 93

4 Evaluation **95**

4.1 Experimental Environment 95

4.2 Workload Models 97

4.3 Temporal Behaviour Analysis 111

4.4 Summary 160

5 Conclusions And Future Work **163**

5.1 Contributions 165

5.2 The Hypothesis Revisited 166

5.3 Future Work 167

Appendices **169**

A Additional Evaluation of Spatial and Cache Behaviour of memory Allocators **169**

A.1 Spatial Behaviour Analysis 169

A.2 Cache Behaviour Analysis 181

B Additional Evaluation of the Revised Node Distance Tables **189**

B.1 Implementation of Node Distance tables in Linux 189

B.2 Measuring Node Distances 192

B.3 Evaluation On The Real Node Distance 198

C Further Evaluation of the Synthetic Models **205**

D Supporting Implementations **219**

D.1 Supplemental Applications 219

D.2 Examples Of Conventional Allocators 231

Abbreviations **235**

References **237**

List of Figures

2.1 Overview of Sequential Fit algorithms 25

2.2 Logical view of Buddy system 32

2.3 Structure of *DLmalloc* 36

2.4 Structure of Half-fit 37

2.5 Structure of TLSF 39

2.6 Structure of *tcmalloc* 41

2.7 Structure of *Hoard* 43

3.1 The wasted memory of TLSF in small sizes of blocks 48

3.2 The structure of nMART 51

3.3 The relationship between nodes, zones and pages on x64 architecture system 57

3.4 The default zones lists for our experimental hardware 61

3.5 The sorted zone lists for our experimental hardware 62

3.6 The thread-based private heap on the first layer 67

3.7 The node-based free arena management on the second layer 69

3.8 The structure of an nMART control block 71

3.9 The header of an arena list 71

3.10 The structure of a thread control block 72

3.11 The header of an arena 73

3.12 The header of normal blocks 74

3.13 A block of small blocks 74

4.1 A four node-based ccNUMA architecture system 96

4.2 A more complex architecture system 97

B.1 The ratio of performance improvements 202

List of Tables

2.1 Worst-case time complexity of algorithms 44

3.1 SLIT of the experimental machine 55

4.1 Real workload characteristics of test set 1 102
4.2 Real workload characteristics of test set 2 102
4.3 Real workload characteristics of test set 3 103
4.4 Real workload characteristics of test set 4 103
4.5 The number of *malloc()* and *free()* calls by *cfrac* 104
4.6 The number of *malloc()* and *free()* calls by *espresso* 105
4.7 The number of *malloc()* and *free()* calls by *gawk* 105
4.8 The number of *malloc()* and *free()* calls by *p2c* 106
4.9 The MEAN of B_S, B_{IT} and B_{HT} generated 110
4.10 The CDF model of B_S, B_{IT} and B_{HT} generated 111
4.11 The average *malloc()/free()* time of Set 1 of *cfrac* 115
4.12 The average *malloc()/free()* time of Set 2 of *cfrac* 116
4.13 The average *malloc()/free()* time of Set 3 of *cfrac* 116
4.14 The average *malloc()/free()* time of Set 1 of *espresso* 119
4.15 The average *malloc()/free()* time of Set 2 of *espresso* 120
4.16 The average *malloc()/free()* time of Set 3 of *espresso* 120
4.17 The average *malloc()/free()* time of Set 1 of *gawk* 123
4.18 The average *malloc()/free()* time of Set 2 of *gawk* 124
4.19 The average *malloc()/free()* time of Set 3 of *gawk* 124
4.20 The average *malloc()/free()* time of Set 1 of *p2c* 127
4.21 The average *malloc()/free()* time of Set 2 of *p2c* 127

4.22 The average *malloc()/free()* time of Set 3 of *p2c* 128
4.23 The average of total *malloc()/free()* time for *cfrac* test set 1 132
4.24 The average of total *malloc()/free()* time for *cfrac* test set 2 132
4.25 The average of total *malloc()/free()* time for *cfrac* test set 3 133
4.26 The average of total *malloc()/free()* time for *espresso* test set 1 . . . 136
4.27 The average of total *malloc()/free()* time for *espresso* test set 2 . . . 136
4.28 The average of total *malloc()/free()* time for *espresso* test set 3 . . . 137
4.29 The average of total *malloc()/free()* time for *gawk* test set 1 140
4.30 The average of total *malloc()/free()* time for *gawk* test set 2 140
4.31 The average of total *malloc()/free()* time for *gawk* test set 3 141
4.32 The average of total *malloc()/free()* time for *p2c* test set 1 144
4.33 The average of total *malloc()/free()* time for *p2c* test set 2 144
4.34 The average of total *malloc()/free()* time for *p2c* test set 3 145
4.35 The execution time of MEAN-Value model with two threads 147
4.36 The execution time of MEAN-Value model with four threads 148
4.37 The execution time of MEAN-Value model with eight threads 149
4.38 The execution time of MEAN-Value model with sixteen threads . . . 150
4.39 The execution time of MEAN-Value model with thirty-two threads . 151
4.40 The execution time of MEAN-Value model with sixty-four threads . . 152
4.41 The execution time of CDF model with two threads 155
4.42 The execution time of CDF model with four threads 156
4.43 The execution time of CDF model with eight threads 157
4.44 The execution time of CDF model with sixteen threads 158
4.45 The execution time of CDF model with thirty-two threads 159
4.46 The execution time of CDF model with sixty-four threads 160

A.1 The total block sizes requested/provided by *cfrac* 171
A.2 The total block sizes requested/provided by *espresso* 171
A.3 The total block sizes requested/provided by *gawk* 172
A.4 The total block sizes requested/provided by *p2c* 174
A.5 The size of virtual memory provided for test set 1 176
A.6 The size of virtual memory provided for test set 2 178

A.7 The size of virtual memory provided for test set 3 179
A.8 The size of virtual memory provided for test set 4 181
A.9 The states of event counters for *cache-scratch* in test set 4 183
A.10 The states of event counters for *cache-thrash* in test set 4 184
A.11 The states of event counters for *larson* in test set 4 186
A.12 The states of event counters for *shbench* in test set 4 187

B.1 Measured Node Distances 194
B.2 The SLIT of more complex architecture system 194
B.3 The measured results for more complex system 197
B.4 The new node distance of more complex architecture system 198
B.5 The measured actual time taken on a four nodes-based ccNUMA system 199
B.6 Allocation timing statistics based on node 0 200
B.7 Allocation timing statistics based on node 1 200
B.8 Allocation timing statistics based on node 2 201
B.9 Allocation timing statistics based on node 3 202

C.1 The mean of all test sets of applications 206
C.2 The cumulative percentage and frequency of *cfrac* test set 1 207
C.3 The cumulative percentage and frequency of of *cfrac* test set 2 208
C.4 The cumulative percentage and frequency of *cfrac* test set 3 209
C.5 The cumulative percentage and frequency of *espresso* test set 1 210
C.6 The cumulative percentage and frequency of *espresso* test set 2 211
C.7 The cumulative percentage and frequency of *espresso* test set 3 212
C.8 The cumulative percentage and frequency of *gawk* test set 1 213
C.9 The cumulative percentage and frequency of *gawk* test set 2 214
C.10 The cumulative percentage and frequency of *gawk* test set 3 215
C.11 The cumulative percentage and frequency of *p2c* test set 1 216
C.12 The cumulative percentage and frequency of *p2c* test set 2 217
C.13 The cumulative percentage and frequency of *p2c* test set 3 218

Acknowledgements

Firstly and very respectfully, I would like to appreciate my supervisor, Professor Andy Wellings. This thesis would not have been possible without his great mind, guidance and support. His endless support, understanding, and patience shaped my coarse-grained idea into this final thesis.

I would also like to thank my assessor, Professor Alan Burns, for his great support and invaluable advice regarding my work. An encouraging, cooperative and truly interested advisor is something that every Ph.D. student wants, whereof I am one of the privileged. Especially, I would like to thank all my friends, Abdul Haseeb Malik, Usman Khan, and Shiyao Lin for giving me strength, self-belief and all the enjoyable moments. My sincere appreciation goes to my best friends, Sungki Kang, Youngbo Kim, and Dinesh Manadhar; without their love and support, it would not have been possible to complete this course.

My family has been supportive of me throughout my Ph.D. I would like to thank my wife, Jumi Kim, and my little angels, Noori and Arie Kim, for their patience, sacrifices and support. I also thank my younger brother with his wife, Jaeyeon Kim and Youngeun Song, for encouraging me throughout my work. Last but not least, my sincere appreciation goes to my father for guiding and supporting me in every possible way.

Declaration

I declare that the research described in this thesis is original work, which I undertook at the University of York during 2008 - 2013. Except where referenced, all of the work contained within this thesis represents the original contribution of the author.

Chapter 1

Introduction

Recent years have seen an explosion in the development and use of modern computing technologies from small embedded devices like smart phones and tablets to large industrial machineries such as auto-mobiles and aircraft. For example, a huge number of people use the Internet to perform searches and for the provision of services. This is possible because innumerable machines run these searches and services behind the Internet. The key force providing the diversity of services is high-performance computing technologies such as parallel and distributed computing. Single processor systems are not able to meet the required demand.

Many high performance services are supported by computers that have ccNUMA architectures. ccNUMA architecture systems are multiprocessor systems that have distributed shared memory. They are more scalable and flexible compared to other multiprocessor architecture such as SMP (symmetric multiprocessor systems). ccNUMA systems provide a single address space, and are globally cache-coherent in hardware. In order to execute on ccNUMA systems, an application that executes on SMP systems does not require any changes. This is an important consideration when existing applications are to be migrated to the new architectures. However, the ccNUMA memory hierarchy does affect the application performance. In such systems, there are considerable benefits to be had by allocating related threads and data close to each other.

Real-time systems have also been increasing in size and complexity and their processing demands can no longer be met by single processor systems, and they are

likely soon to outpace the computation power of SMPs. Furthermore, it is extremely likely that a real-time application can be sharing the system's resources with other real-time applications concurrently. This exacerbates the problem of meeting the computational demands of the applications. Hence, in the near future, real-time systems will require processors that have ccNUMA architectures as these offer more extensible computing platforms. However, it is difficult to do the global timing analysis of large systems. Accordingly, architectural complexity and tight timing constraints make the development of real-time systems on multiprocessor ccNUMA architectures extremely difficult [Wellings et al., 2010].

In general, a real-time system can be defined in many ways; a real-time system often refers to one that has the ability to perform many computations extremely fast. The powerful computation ability can minimize average response times, but it does not guarantee predictability, as is required in the real-time domain. The faster computation is a necessary condition, but it is not sufficient in the domain.

Burns and Wellings [Burns and Wellings, 2001] give the following definition of a real-time system: "The correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced." Real-time systems are often classified as *hard* or *soft*. Hard real-time systems are those that must provide absolute guarantees that tasks will meet their deadlines. In soft real-time systems, deadlines are important but there is no strict guarantee requirement. Tasks which complete late, still can provide value to the system. This thesis is concerned with the implementation of *soft* real-time systems on ccNUMA architectures.

1.1 Motivation

Memory is one of the significant concerns when developing real-time applications as its management costs are expensive. Consequently, worst-case latencies and memory utilization are the primary concerns of real-time developers; a cost-effective real-time system must also exhibit good average-case performance [Nilsen and Gao, 1995]. As a result of the unpredictable allocation and de-allocation of memory blocks, many

real-time systems use static memory allocation; this refers to the process of allocating memory at compile-time and during program initialization before the application enters into it main real-time phase of execution. They do not use dynamic memory allocation and all physical memory is available as one contiguous block that can be used as and when required. As there is no automatic memory management, a problem arises concerning unnecessary memory space consumption. Memory that is required for the storing of temporary objects cannot easily be reused. Programmers have to implement and manage their own memory pools to reduce unnecessary memory space consumption and reuse space. As has been noted earlier, real-time systems have been increasing in size and complexity with the explosion of multiprocessor and multi-core architecture systems. This requires, among other things, more flexible use of the available resources including memory. The burden of static memory and memory pool management becomes unacceptable. From the developer's view, in the future, the development of real-time applications will increasingly use dynamic memory management[1] to achieve the expected flexibility and performance.

Dynamic storage allocation (DSA) has been one of the most important fundamental part in the general-purpose software domain due to being more efficient and flexible than static memory allocation. With its importance and popularity, the research area of DSA algorithms has been studied for over fifty years. In the past, much scholarly work has been done on the topic of good average response time of DSA algorithms, with respect to how fast and efficiently they allocate or de-allocate memory blocks, and how to reduce memory fragmentation. Although large numbers of faster and more efficient DSA algorithms exist in general-purpose domains, such as *DLmalloc* [Lea, 1996], *TCmalloc* [Sanjay Ghemawat, 2010] and *Hoard* [Berger et al., 2000], and are used widely, surprisingly, the worst-case execution time (WCET) of DSA algorithms, which can be high, has has not been studied in detail. There have been only a few general studies on the dynamic storage allocator in real-time system's domain. Due to the lack of studies, most application developers of real-time systems generally avoid using dynamic storage allocation algorithms. This is

[1]Dynamic memory management is also called dynamic storage allocation, memory management, heap memory, heap space, or just heap for historical reasons [Hasan et al., 2010].

because they are concerned that the worst-case execution time of DSA routines is not bounded or is bounded with an excessive bound [Puaut, 2002]. As well as the WCET of the DSA algorithm, space efficiency should be considered, as the lifetime of a real-time application is usually longer than a general-purpose application's one. During a long lifetime, dynamic memory allocation can leave holes in memory, which cannot be reused due to their small size, and these holes lead to slow unacceptable response time or to miss deadlines. This is known as memory fragmentation.

In addition to the above concerns, ccNUMA architectures introduce another problem for DSAs. To maximize performance, memory allocated to the application must be local to the hardware node that is performing the memory access.

1.2 Hypothesis

The current existing dynamic storage allocation algorithms for real-time systems do not have any appropriate functionality to support multi-processors, multiple threads, and ccNUMA architecture systems. This thesis is concerned with how a dynamic storage allocation algorithm, supporting ccNUMA architecture systems, can be bounded with a small bound to satisfy the timing constraints of soft real-time systems. The hypothesis of this thesis is defined by the following statement:

> *The ability of a dynamic storage allocation algorithm can be enhanced to meet the requirements of soft real-time systems with a small bounded execution-time on ccNUMA architecture systems.*

In order to avoid ambiguity, the term *wasted memory* in this thesis refers to those parts of memory which are free but cannot be allocated to the application. It is also called *memory fragmentation*, which can be divided into two parts: *internal* and *external*.

1.3 Thesis Objectives

This thesis is mainly concerned with dynamic storage allocation algorithms on a ccNUMA architecture system and its efficient implementation on Linux. The major

aim of the research is to enhance memory allocation algorithms to support ccNUMA architecture systems, enabling programmers to maximize exploitation of the system's characteristics without significant effort, and to offers better efficiency with bounded execution times for the allocation/de-allocation of memory.

In order to prove the hypothesis, this thesis provides six objectives. These are:

1. A detailed investigation into the limitation of why the existing dynamic storage allocation algorithms to allocate/de-allocate memory on ccNUMA architecture systems introduce unexpected large space and time overheads.

2. An approach for automatically re-sorting the node order, which minimizes accessing the farthest node on the system and automatically maximize accessing the closer node for memory allocation requests. As one of the main parts of the thesis hypothesis, this approach gives better performance. Also it releases real-time application developers from the responsibility of memory management.

3. Temporal and spatial guarantees to real-time applications to ensure they meet their timing and space requirements in order to eliminate the non-determinism caused by the memory distribution in ccNUMA architecture systems.

4. To support transparency of the underlying architectures to applications. Memory requests are satisfied under this algorithm independently of the underlying architectures.

5. A synthetic model to analyze dynamic storage algorithms on ccNUMA architecture systems to check for improvements in the performance and predictability.

6. An overview of the implementation of: (a) the physical memory management on a modified Linux kernel, and (b) the proposed dynamic allocation algorithm.

Meeting these objectives forms the main contributions of this thesis. Achieving these goals will facilitate the development of efficient and configurable applications

under an enhanced Linux kernel on ccNUMA architecture systems. In summary, the thesis aims to define a more efficient dynamic storage allocation algorithm and enhance the current Linux kernel to enable applications to be more deterministic and portable in their use of memory allocation and deallocation.

1.4 Organisation of the Thesis

The remaining chapters of this thesis are organized with five chapters in accordance with the motivations and objectives of the research. Briefly, the descriptions of the remaining chapters are given below:

Chapter 2. Dynamic Memory Managements: This chapter explores the dynamic storage algorithm models provided by general-purpose and real-time systems, including their policies and mechanisms. Furthermore, conventional memory management algorithms are investigated, with particular emphasis on managing memory blocks. This chapter also highlights hybrid memory management algorithms, which perform better on multiprocessor architecture systems compared to conventional DSA algorithms.

Chapter 3. nMART: A ccNUMA-aware Dynamic Storage Allocation Algorithm: This chapter concentrates on providing and implementing a more efficient and predictable memory management algorithm that is especially designed for use in real-time systems. This is to achieve the primary intention of an algorithm which is bounded with small bounds. It is designed to allocate/de-allocate memory blocks with the "closest node-orientated allocation" policy. Also, the chapter discusses the current methods of modern operating systems for the management of the physical memory in ccNUMA architecture systems, particularly Linux, and how our model performed better in terms of the physical memory management.

Chapter 4. Evaluation: This chapter suggests an evaluation method, particularly in terms of remote memory access, for the performance of DSA algorithms on the target architecture systems. Also, a set of experiments is performed to

evaluate the performance of DSA algorithms. This chapter also compares the results with other schemes such as *First-Fit*, *Best-Fit*, *Half-Fit*, *Hoard*, *tcmalloc*, and *TLSF*. Note that spatial and cache behaviour analysis of algorithms are discussed in Appendix A.

Chapter 5. Conclusions And Future Work: The final conclusions from the research results are given in this chapter. Additionally, some directions for further research are also presented.

Chapter 2

Dynamic Memory Managements

2.1 Introduction

Dynamic memory management is one of the most important techniques in modern software engineering to manage objects created at runtime using high-level languages like *C*, *C++* and *Java*. It manages free or in-use memory blocks which have shorter lifetimes than their owner tasks or processes. In general, it is extremely difficult to satisfy the timing constraint of real-time applications with dynamic memory management. This is because it is necessary to predict the worst-case execution time of dynamic memory management offline. In addition, finding the optimal place to allocate a block of memory is NP-hard when some blocks are already allocated [Robson, 1980]; also, fragmentation can occur where it is not possible to satisfy a request, even if the total size of available memory exceeds the requested memory size.

When using dynamic memory management on multiprocessor environments, new problems, such as false sharing, unbounded growing heaps, and synchronization between threads are introduced. This is because the requirements of applications running on multiprocessors differ from those of applications running in uniprocessor environments.

In this chapter, we will discuss the issue described above, in particular, we will review memory management details related to this thesis. In the first section, the objectives of memory management and some of the terminology of memory man-

agements are discussed. Section 2.2 introduces some fundamental issues of memory management. In Section 2.3, a diversity of conventional memory management algorithms and hybrid algorithms are discussed. Lastly, a summary will be drawn.

2.1.1 Objectives of Memory Management

The research in dynamic memory management for real-time systems is one of the still unconquered areas primarily because real-time applications impose different requirements on memory allocators from general-purpose applications. For instance, one of the most important main requirements in real-time systems is that schedulability analysis should be performed to determine whether application response times can be bounded to satisfy the run-time timing constraints. The analysis should consider the impact of multiprocessor environments such as the high levels of concurrency, lock contention, heap contention, cache misses, and traffic on the bus. Considering all these issues with our target ccNUMA architecture systems, the requirements of real-time applications related to dynamic memory management can be summarised as follows:

Minimize memory fragmentation: The lifetime of real-time applications is generally longer than those of general-purpose applications, and can be as long as a day, a month or even years. In a long lifetime, the application may free memory blocks of any size arbitrarily. This can lead to the creation of holes in the memory, which cannot be reused because they are too small. Consequently, minimizing memory fragmentation needs to be considered as an important key requirement.

Minimize false sharing: False sharing introduces much of the unnecessary traffic on the bus in order to maintain cache coherency. It occurs when multiple processors attempt to read/write different data objects within the same cache line or page. Even if the processors do not actually share data, there are overheads due to coherency operations manipulating cache lines [Bolosky and Scott, 1993] [Jeremiassen and Eggers, 1995]. Consequently, false sharing can be a critical cause in degrading applications' performance on shared memory

multiprocessor environments.

Maximize node-oriented data locality: As ccNUMA architecture systems are one of the distributed shared memory systems, their processors are able to access remote memory but with higher latencies than with local memory accesses. Remote memory accesses, therefore, lead to degrading the system performance. Consequently, node-based data locality, which encourages accessing its local memory, is considered as an important key requirement for improving the performance of the system. Also, allocating memory blocks, which are usually used together and near each other, lead to minimizing page and cache misses at runtime.

Minimize memory access to the farther nodes: Application developers need to consider remote memory access latencies and fully understand the specification of the underlying hardware. However, nobody can ensure that all developers fully understand the characteristics. For this reason, DSAs need to provide transparency to developers so that latencies can be reduced thus minimizing the memory accesses to the farthest away nodes.

Bounded execution time: As has been discussed, applications on real-time systems should satisfy their timing constraints. In order to satisfy their deadlines, the dynamic memory management operations should be bounded with a small bound at run time.

Minimize lock contentions: Applications running in ccNUMA and multi-threaded environments must be concerned with high-level concurrency and scalability issues. In particular, lock contentions lead to limiting systems' scalability and add complexity, thereby, degrading system performance. As a result, minimizing lock contentions in the memory management software is also needed.

Minimize consumed space: Dynamic memory management attempts to consume space conservatively. It needs to use as little memory as possible to keep track of the maintenance information needed during of the memory management of the system.

2.1.2 Terminology

Despite over thirty years of research in dynamic memory management, a precise definition and quantification of the terms had proven to be elusive before Wilson's paper [Wilson et al., 1995b]. In this paper, Wilson provides some of the terminology which is now frequently used in the memory allocation area. For clarity, this terminology will be used in this thesis. The main terms are defined below.

Strategy: A strategy is the basic approach used to design a memory allocator. It takes into account patterns in program behaviour, and determines a range of acceptable policies for placing dynamically allocated memory blocks. The objectives of an allocator may be considered as being equivalent in meaning to the allocator's strategy, for example, "minimizing lock contentions for each allocation" or "maximizing data locality to encourage accessing local nodes". These strategies are achieved by policies.

Policy: A policy is an implementable decision procedure for placing memory blocks dynamically. It determines exactly where an allocating block will be extracted from the memory or where a freed block will be inserted into the memory. For instance, a given policy says: "always attempt to find the smallest block that is large enough to satisfy the request". These chosen policies are implemented by a collection of mechanisms. Policies can be separated into the following: *Exact-Fit*, *Best-Fit*, *Good-Fit*, *First-Fit*, *Next-Fit* and *Worst-Fit*. Some of the most important policies will be discussed in Section 2.3.

Mechanism: A mechanism is a collection of algorithms and data structures that implement a policy. It may be simply equivalent in meaning to an algorithm. An example of a mechanism is to "use a doubly linked list, and search for the position of the free block list from where the last search was satisfied; freed blocks are inserted at the front of the list". Typically, the mechanism can be divided into the following: *Sequential Fit*, *Segregated Fit*, *Buddy Systems*, *Indexed Fit* and *Bitmapped Fit*. Some of the most important mechanisms will be discussed in Section 2.3.

The above set of definitions is important for understanding and designing a dynamic memory management system in detail. For example, given a strategy, different policies may lead to different secondary effects. If some policies introduce good locality with high fragmentation, an application developer may need to choose another policy under the same strategy, which produces low fragmentation. A policy can be implemented by a diversity of mechanisms. If a given policy performs well, but its implementation is not efficient, developers can implement the policy by choosing a different mechanism.

Theoretically, keeping fragmentation under control is one of the major functionalities of dynamic memory management achieved by the placement policy. The placement policy is the choice of where to put a requested memory block in free memory. It is achieved by two techniques: splitting and coalescing.

Splitting: This splits large blocks into smaller blocks, and uses large divided blocks to satisfy a given request. Typically, the remainding blocks are tracked as smaller free blocks, and used to satisfy future requests.

Coalescing: Coalescing occurs when applications free up used memory blocks. In general, when applications free blocks of memory, the memory manager checks to see whether the neighbouring blocks are free or not, merging them into a single larger block if they are freed. This is more desirable because a larger block is more likely to be useful than two smaller blocks.

Coalescing can be separated into two different categories. Firstly, *immediate coalescing* attempts to merge freed blocks immediately whenever a block is freed. This will typically be expensive because freed blocks will be coalesced together by repeatedly and frequently combining adjacent free blocks. In contrast, *deferred coalescing* simply marks a freed block as “unused” or “freed” without merging. This is because many applications repeatedly create short-lived objects of the same size. Such allocators keep blocks of a given size on a simple free or unused list, reusing them without coalescing and splitting so that if an application requests the same-size memory block soon after one is released, the request can be satisfied by simple

operations in a constant time. This may optimize if some sizes are very commonly allocated and de-allocated.

However, [Johnstone and Wilson, 1998] provided an analysis of deferred coalescing, reporting that memory fragmentation problems come into effect for the most common applications, and that deferred coalescing leads to unbounded execution time. For this reason, allocators in real-time systems have used immediate coalescing.

2.1.3 Analysis Methodology

There have been many analysis methodologies used to evaluate dynamic memory management. In many methodologies, two different approaches are generally used. To evaluate fragmentation and worst-case execution time, scenarios are constructed using synthetic workloads; while to compare average execution time, real workloads are used.

- **Synthetic trace analysis**: In the past, this has been one of the most widely used approaches. It consists of a few traces with artificial workloads of allocations and de-allocations. Of course the initial condition is needed as well; the methodology can offer highly precise simulations of what allocators will do because allocators usually provide responses in the order of given requests. The specific workloads can change the size of requested memory blocks distribution and the lifetime distribution of memory blocks to evaluate the affect on fragmentation.

  For example, a simple function (e.g. sizes increased by a power of two) can be used to change the size, or select the size and lifetime according to the values from the function, or to use statistics of the size and lifetime collected from real applications.

- **Real trace analysis**: Another approach is to trace memory operations from real applications, rather than randomly generated requests of the size and the lifetime. This uses a number of memory-oriented applications which consume a large amount of memory and time processing memory operations, most of

which were described by [Grunwald et al., 1993], such as *espresso*, *gs*, *gawk* and *make*. The real trace evaluates the memory allocator performance in both space and time using these real applications.

In [Wilson et al., 1995a], Wilson found that the synthetic trace discards almost all major information relevant to estimating real fragmentation. Furthermore, in [Zorn and Grunwald, 1994], Zorn concluded that synthetic trace analyses are not sufficient to reflect an allocator's performance accurately. In addition, the paper shows that the likelihood between the fragmentation of the real trace and the fragmentation of the synthetic trace is only 0.5, thus meaning that most of the fragmentation corresponding with the original trace cannot be reflected by the synthetic trace. They concluded that both size and lifetime of synthetic traces are insufficient to fully predict allocator performance for real workloads. After these papers, most research used a combination of both real and synthetic trace analysis.

There is a correlation between the amount of memory fragmentation and the behaviours of real applications. [Wilson et al., 1995b] defined three patterns of memory usage over time, which have been observed in a variety of applications, as follows:

- **Ramp**: A variety of applications build up specific-purpose data steadily over time, such as stacking event logs. This pattern is called the *ramp* pattern.

- **Peak**: the *Peak* pattern is similar to the ramp pattern but it is over a short period of time. Some of the applications use lots of memory intensely in a short time to build up large data structures. In general, after using data structures, most of the data will be freed.

- **Plateau**: Some applications gather data structures rapidly and use them for long periods, even the whole duration of the application. This situation is called the *plateau* pattern.

In the paper, they concluded that the fragmentation at the *peak* is more important than the average fragmentation. This is because the most important periods are those when the most memory is used. Scattered holes in the memory may not be a problem in the earlier phase if the holes are filled in the later phase but most

applications never reach a truly steady state as applications usually show memory usage patterns with ramps and/or peaks patterns.

2.2 Fundamental Issues

In general, designing a memory allocator is a trade-off between time efficiency and space efficiency. Without making a compromise between them, it is rarely possible to design a memory allocator that is extremely fast with minimum fragmentation for most applications. For instance, Kingsley's memory allocator [Kingsley, 1982] is an example of simple segregated storage algorithms, which is used in 4.2 BSD Unix distribution. The memory allocator rounds memory block request sizes up to powers of two minus a constant. The principle of allocation and deallocation is very simple inasmuch as popping off from and pushing onto an array of segregated lists in size classes. The performance of its implementation is very fast because its algorithm is so simple, e.g. no attempt is made to coalesce memory blocks. Contrary to the time efficiency, it wastes a significant amount of space, potentially an average of 50% of the memory can be wasted due to internal fragmentation. Therefore, the balancing between the time efficiency and the space efficiency is one of the most important aspects of designing a memory allocator.

A number of memory allocators employ either a single heap or several private heaps for uniprocessor environments; however, most of the modern memory allocators have started to consider more complex environments for the emerging multi-core multiprocessor architecture systems. A more complex architecture system brings new and different problems, such as heap contentions, false sharing, unbounded growing heap problems as well as the traditional fragmentation problem. The subsections below will discuss these fundamental problems.

2.2.1 Fragmentation

One of the significant problems of a memory allocator is memory fragmentation. In [Randell, 1969], Randell classified fragmentation as *External* and *Internal*, both of which are caused by splitting and coalescing free blocks.

External fragmentation arises when a requested memory block cannot be satisfied, even if the total amount of free memory is larger than the size of the request. During allocation and deallocation processing, this fragmentation is generally caused when a small number of free blocks are created called '*holes*'. The small number of free blocks are not adjacent so cannot be merged, and are too small to satisfy any request.

Unlike external fragmentation, internal fragmentation arises when an allocator returns a larger free block to satisfy the request, rather than the actual requested size with the remainder being simply wasted. This is the reason why this situation is called internal fragmentation. Formally, the remainder is just inside an allocated block. Arguably, internal fragmentation is only caused by poor implementation of the allocator policy [Johnstone and Wilson, 1998] [Masmano et al., 2008a]. However, in some allocators, internal fragmentation is often accepted for increased performance or simplicity. For instance, many of the segregated fit allocators allocate larger free blocks to avoid creating memory blocks that are non-aligned or too small a size. In binary buddy systems (discussed in Section 2.3.3) and Half-fit cases (discussed in Section 2.3.5.2), the sizes of allocated blocks are always rounded to powers of two by the policy because those allocators cannot divide blocks into different sizes from those preset by the policy; the algorithm pre-defines a set of discrete sizes of the data structure. Unlike external fragmentation, internal fragmentation is unique to each implementation of an algorithm and it must be studied case by case. This is why it is hard to find a general study of internal fragmentation in the literature.

Numerous publications address numerous experimental approaches for controlling fragmentations. Usually, the results depend on three variables: M is the maximum amount of heap memory that the allocator can use, n is the maximum of the block size that the application can request, and C is a constant. In [Robson, 1971], Robson showed that the amount of memory needed by any strategy is bounded below by a function, $M \log_2 n{\cdot}C$, which rises logarithmically with the size of blocks used. Robson addressed upper and lower bounds on the worst-case fragmentation of the optimal allocation algorithm. The paper showed that the upper

bound of a worst-case optimal strategy would be between $0.5M \log_2 n$ and about $0.84M \log_2 n$. Another of Robson's papers [Robson, 1977] showed that the upper bounds of address-ordered first-fit policy are about $M \log_2 n$, whereas the best-fit policy needs a store of at least $(M - 4n + 11)(n - 2)$, and the pessimistic asymptotic bound is around $M \cdot n$. In [Knuth, 1997](first edition 1973), Knuth proved that the upper bound of fragmentation in a binary buddy systems could be calculated as $2 \cdot M \cdot \log_2(n)$. Confessore [Confessore et al., 2001] addressed a periodic allocation problem in which allocation and deallocation time of each item are periodically repeated and is equivalent to the interval colouring problem on circular arc graphs. They provided a 2-approximation algorithm, and also showed that the solution value is equivalent to the length of the longest weighed path of the oriented graph.

The dynamic memory allocation problem, storing all objects in the minimum-size memory block, is known to be NP-hard in the strong common sense [Garey and Johnson, 1979]. In [Gergov, 1996], Gergov achieved an approximation result for this problem with a performance ratio of 5. In another paper [Gergov, 1999], Gergov had achieved a 3-approximation algorithm for memory allocation. Luby [Luby et al., 1994] introduced a new parameter called k, which denotes the maximum number of occupied memory blocks, for analyzing algorithms, and improved on Robson's previous research [Robson, 1977]. This proved that the first-fit policy needs a store of at least $\mathcal{O}(M \min\{\log n, \log k\})$ words.

Given the above theoretical analysis, the situation seems rather pessimistic. For instance, assuming that an allocator uses a single heap of M = 1M bytes with a first-fit algorithm, with the maximum block size being n = 4K bytes, the allocator needs at least $2^{20} \cdot (1 + \log_2(2^{12}))$ bytes; a total of 13M bytes is needed to ensure that the allocator never fails because of external fragmentation. In the case of the best-fit policy, it is even larger. With the same condition above, it needs at least $2^{20} \cdot 2^{12}$ = 4G bytes to ensure the best-fit policy always satisfies all requests. In contrast, experimental results are much more encouraging.

Hirschberg [Hirschberg, 1973] compared a binary buddy system with a Fibonacci buddy system. The paper showed that the fragmentation of Fibonacci buddy can increase memory usage by about 25% in contrast to binary buddy's 38%. In [Shen

and Peterson, 1974], Shen showed that a weight-buddy system using FIFO-order with a uniform size distribution wastes more memory than a binary buddy system - around 7% - due to fragmentation. With an exponential distribution, the weight buddy system using FIFO-order gives an improvement of around 7% over binary buddy. In contrast to FIFO-order, memory usage had been worse - around 3% - with LIFO-order. Shore [Shore, 1975] compared best-fit, address-ordered first-fit, worst-fit, and combined best-fit and first-fit as a hybrid policy. The results showed that best-fit and first-fit policies roughly outperformed the others in fragmentation, and the maximum difference between them was less than 3%.

Bohra [Bohra and Gabber, 2001] found that the behaviour of a long-running application with a memory allocator is fairly different from the typical patterns for which memory allocators are optimized. In their experiment, the best optimized algorithm caused 30.5% fragmentation with a long-running application called Hummingbird, and another called GNU Emacs caused 2.69%, but the worst case had predicted 101.5% fragmentation. Real applications are designed to solve actual problems, which affect their pattern of memory usage, so that applications do not behave randomly by the chosen methods used to solve the original problems. Unfortunately, application behaviours have a wide variety of implications for fragmentation so that in order to understand fragmentation, it is necessary to discuss application behaviours as seen in [Bohra and Gabber, 2001]. For instance, the size distributions of requested memory blocks determine memory fragmentation. The lifetimes distribution of memory blocks determines which memory blocks are occupied or free over time.

In [Johnstone and Wilson, 1998], Johnstone investigated the fragmentation produced by a group of policies including *first-fit*, *best-fit*, *next-fit*, *address-ordered first-fit*, *address-ordered best-fit*, *address-ordered next-fit*, *DLmalloc*, etc. with a set of real traces. They concluded that the fragmentation problem is produced by a poor allocator implementation, and well-known polices did not suffered from almost any genuine fragmentation.

Barret [Barrett and Zorn, 1993] introduced an interesting approach to avoiding fragmentation by predicting the lifetimes of short-lived objects when they are allo-

cated. They showed that their scheme would predict that a large fraction (18% to 99%) of all allocated bytes are short-lived.

Consequently, it can be concluded that some dynamic storage allocation algorithms have pessimistic fragmentation; however, in many studies, many allocator algorithms show low fragmentation in memory usage with well-designed policies.

The Measure of Fragmentation

To compare allocators, a metric is needed. In general, the time cost and the space cost are the most commonly used measurements; in particular, the time cost denotes speed, and the space cost indicates fragmentation.

Memory fragmentations can be defined in many different ways. For instance, assume that there are 10 free blocks of size 4K bytes and 50 free blocks of size 1K bytes in memory at some point in time, and an application will request 5 free blocks of size 4K and 40 free blocks of size 1K in the near future. In this case, was cannot say there is high fragmentation because the requests can be satisfied. With the same condition above, if the allocation will request 10 free blocks of size 8K, there will be a problem and we can say that this is due to high fragmentation. Johnstone [Johnstone and Wilson, 1998] suggested four metrics to describe the amount of fragmentation, which are now widely used. They used both metric 3 and 4 in the paper.

Metric 1: The measured fragmentation is the amount of memory used by the memory manager, which is normally called the heap, over the amount of memory requested by the application, averaged at all points through time. This metric of fragmentation measure is simple, but a problem with this approach is that it hides the spikes in memory utilization, with these spikes being where fragmentation can become a problem.

Metric 2: The fragmentation is the amount of memory used by the memory manager over the maximum amount of memory required by the application at the point of maximum memory utilization. The problem of this metric is that the point of maximum memory utilization cannot normally be considered the most important point of the application at runtime.

Metric 3: The fragmentation is equal to the maximum amount of memory used by the memory manager over the amount of memory required by the application at the point of maximum memory usage by the memory manager. The drawback corresponding with this measure of fragmentation is that it will lead to high fragmentation, even if the applications uses slightly more memory than the size of needed memory.

Metric 4: The fragmentation is the maximum amount of memory used by the memory manager over the maximum amount of memory used by the application. The disadvantage of this measure is that it can report low fragmentation when the point of maximum memory usage is a point where a small amount of memory is used by the application.

All the metrics described above are available to measure the fragmentation caused by application behaviours; however, a problem with these metrics is that they do not distinguish between unused memory and memory used by the memory allocator for its own data for management, such as keeping free blocks. In our experiment, we will consider the actual amount of internal and external fragmentation, with space consumed by data structures being maintained. Therefore, we will use the following equation to calculate the amount of fragmentation (f):

$$f = \frac{h - a}{h} \tag{2.1}$$

In this equation, h denotes the actual amount of memory provided by the allocator, and a points out the amount of allocated memory requested by the application.

2.2.2 False Sharing

Writing multi-threaded applications is a challenge for many well-known reasons, such as debugging, avoiding race conditions, a variety of contentions, and deadlocks. With the emergence of multiprocessor architecture systems, memory allocators need be more concerned with the potential pitfalls resulted from parallel executions of threads requesting memory. In the remaining sub-sections, we will discuss some issues arising in multiprocessor environments.

Most multi-threaded applications share system resources between threads. In this case, contention arises when threads try to read or write a shared resource. However, the contention can sometimes happen when multiple threads access different objects. This is because multiple objects happen to be close enough in memory, wherein they reside on the same cache line.

For instance, a thread updates object *obj1*, whereas another thread updates another object *obj2*. Assume that both objects reside on the same cache line and both threads are running on different processors, the cache-coherency protocol will make the entire cache line an invalidated state when one of them is modified. The cache line will "Ping-Pong" between the processor caches. Therefore, it leads to degrading the application performance [Hyde and Fleisch, 1996].

This situation is called *false sharing*. Recently, with the popularity of using multi-core architecture systems, the trend towards increasing cache line sizes makes false sharing increasingly common [Liu and Berger, 2011]. It is rarely possible to eliminate false sharing automatically. One representative approach is that either application developers deal with this problem by adjusting the data structure layouts, i.e. alignment and padding, or compilers schedule a parallel loop [Jeremiassen and Eggers, 1995]. These approaches can reduce the correlation between false sharing and data objects; however, these strategies cannot avoid it completely because of the effects on array-based data structures. Nevertheless, a well-designed strategy can reduce and eliminate the possibility of false sharing in practice [Berger et al., 2000].

2.2.3 Single and Multiple Heaps

Despite the increasing popularity of concurrent application on both multi-core and multiprocessor architecture systems, there have been few studies on concurrent memory managers. The most representative paper on the dynamic storage allocation [Wilson et al., 1995b] was surprisingly limited to investigating non-concurrent memory managers. Typically, in the uniprocessor environments, two approaches can be used by a memory manager to deal with multiple threaded applications: a *serial single heap* and a *concurrent single heap*.

However, those heaps naturally suffer contention when multiple threads access the same heap. The problem with the single heap is that the greater number of threads accessing the heap, the more heap contention is likely to arises. As a result, a variety of concurrent memory managers started to use multiple heaps. There are many mechanisms to assign threads to heaps, such as the allocators mapping threads onto heaps by assigning one heap to every thread, by using an unused heap from a group of heaps, by assigning heaps in a round-robin fashion, or by using a mapping function to assign threads to a group of heaps [Berger et al., 2000]. Berger has classified these heap usage approaches as follows:

- **A serial single heap**: The allocator which uses a serial single heap is normally fast and can likely keep low fragmentation in practice. However, the heap is protected by a global lock, and naturally it introduces the serialization of memory accesses and significant lock contention.

- **A concurrent single heap**: This heap is helpful to reduce the serialization and the heap contention. It is normally implemented using a concurrent data structure, like a B-tree or a free list with locks. However, its cost of memory access is relatively high, since it usually employs many fine-grained locks or atomic operations on each free block. Furthermore, false sharing still remains likely.

- **Pure private heaps**: A pure private heap indicates that a separate heap is allocated to each thread and these are completely isolated from other threads so that each thread cannot access any other private heap for any memory operation except its own heap. As a result, an allocator with multiple heaps can reduce most of the lock contentions on the private heap, and expect to be scalable. Unfortunately, it is likely to cause a private heap to grow without bounds. For instance, assume that threads are in a producer-consumer relationship, if a producer thread *T1* allocates memory *M1* and a consumer thread *T2* releases *M1*, the memory *M1* will be added into *T2*'s heap.

- **Private heaps with ownership**: Unlike the allocators using pure private heaps, the allocators with ownership return free blocks to the heap where the

target block came from. However, in a producer-consumer model applications, which exhibit round-robin behaviour, the allocators can eliminate allocator-induced false sharing but it still induces unbounded memory consumption.

- **Private heaps with thresholds**: The allocator employs a hierarchy of heaps, and some heaps can be shared by multiple threads, except the private heaps. The shared heap can exhibit some heap contentions, but rarely is there contention on the private heaps. Therefore, the allocators can be efficient and scalable. When the number of private heaps exceeds the threshold, a portion of free memory will be moved to the shared heap, with fully empty heaps being returned to the underlying OS in bounded heap increments. However, the memory management cost, particularly the time cost, can be high because it needs multiple memory operations.

Since using multiple heaps, many memory managers have suffered from unbounded heap increments, wherein memory consumption cannot be unbounded by a policy even if the required memory is fixed. Berger [Berger et al., 2000] called this phenomenon *blowup*. Some follow-on studies have used the terminology, so we will use it as well even though it is not common. The *blowup* phenomenon results from two types of memory consumption patterns. The first type is from the use of pure private heaps in the allocators used in the C++ Standard Template Library [SGI, 2004]. The second is based on the private heaps with ownership used in Ptmalloc [Gloger, 2001] and LKmalloc [Larson and Krishnan, 1998], where the pattern of memory consumption linearly increases with the number of processors. Some allocators, Hoard [Berger et al., 2000] and Vee and Hsu [Vee and Hsu, 1999], exhibit bounded memory consumption as they support the private heaps with thresholds policy.

2.3 Memory Management Algorithms

It is appropriate to review the most well-known policies of memory allocation, even if they are derived from the 1960s, because most modern memory allocators are

variants of these allocation algorithms. Also, the original allocation algorithms are simple and easy to use in small devices. There are two approaches to analyze the WCET of memory management algorithms: static WCET analysis, or worst-case complexity analysis. However, it is impossible to use the static WCET analysis without knowledge of the history of the allocation/deallocation requests in the lifetime of application using the allocation algorithms [Puaut, 2002]. Consequently, worst-case complexity analysis will be used to obtain the worst-case allocation/de-allocation time in this thesis.

In this section, we will discuss sequential fit, segregated fit, buddy systems, indexed fit, and modern hybrid memory allocation algorithms in this order.

2.3.1 Sequential Fit

Typically, sequential fit algorithms can be classified into four types: *Best-Fit*, *First-Fit*, *Next-Fit* and *Worst-Fit*. Figure 2.1 illustrates the differences between sequential fit algorithms. In particular, it contains five free blocks of different sizes as well as six used blocks. The block sizes are given in the header of each block, which contains pointers comprising doubly links.

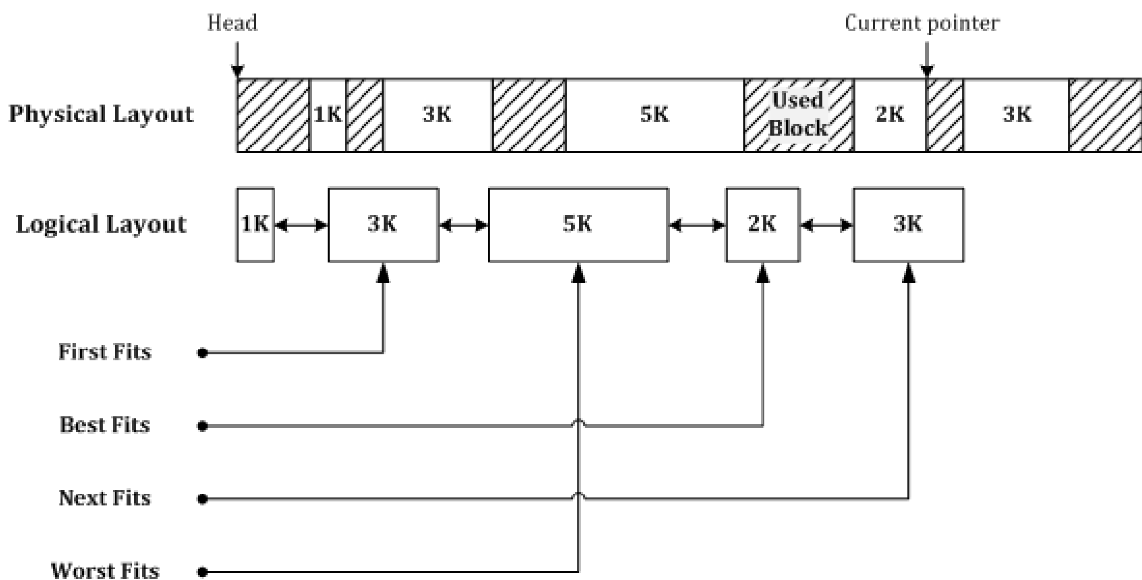

Figure 2.1: Overview of Sequential Fit algorithms

Assume that the application requests 2K bytes of free memory, Best-fit will

return the 2K bytes free block, and *First-fit*, *Next-fit* and *Worst-fit* will return the 3K bytes, the 3K bytes, and the 5K bytes in that order, as seen in the figure.

- **Best-fit**

The *Best-fit* allocation algorithm is one of the most well-known and simplest algorithms. As indicated by its name, the algorithms always attempts to find the smallest free block that is large enough to satisfy the application's request. The policy tends to minimize the wasted space to ensure fragmentation that is as small as possible; however, the algorithm performs an exhaustive search of the data structure in order to find the best-fitting block similar to the requested size. In the worst case, *Best-fit* can be achieved in a time complexity of $\mathcal{O}(n)$ for finding the first block using a doubly linked list array.

The algorithm is typically implemented using either a doubly linked list or circularly linked list, but its data structure may not only be a single linear list of all free blocks of memory, but also more complex, such as arrays of multiple size classes like the segregated fit mechanism or a self-balancing binary tree to implement the same policy with a better time response.

Typically, the *Best-fit* policy can be implemented by the address-order, FIFO and LIFO mechanism. Although those algorithms are based on exhaustive searching algorithms, they shows fairly good memory usage in practice with either real or synthetic workloads; however, they tend to accumulate small fragmentation [Knuth, 1997], but this does not seem to be a significant problem. This is because it does not appear to happen in practice with either real or synthetic workloads [Bays, 1977] [Wilson et al., 1995b].

The *Best-fit* algorithms can be summarised as follows: firstly, the allocators search for a free block that is large enough to satisfy the request iteratively from the head of the free block list until it encounters a suitable block. If the allocators find a block, searching will be terminated and they will return the block. If the allocators find multiple suitable blocks, the choice of the block depends on the implementation of the algorithm. If the size of the found block is larger than the requested size, the block will be split and the remaining one will be inserted into the list. Otherwise the

allocators have failed and will return with a failure. In terms of the deallocation, free blocks are merged with adjacent blocks if they are freed already. They are extracted from the list and merged with the newly freed block to create a larger-size block.

- **First-fit**

First-fit allocation algorithms [Brent, 1989] [Knuth, 1997] are also one of the most common sequential fit mechanisms. They attempt to find the first free block that is large enough to satisfy the memory allocation request. The algorithms can be implemented with a address-ordered, LIFO or FIFO mechanisms. In the worst case, *First-fit* allocation can be achieved with a time complexity of $\mathcal{O}(n)$ for finding the first block with a doubly linked list array.

The *First-fit* algorithms can be summarised as follows: firstly, the algorithms search for a free block from the head of free lists iteratively until finding the target. If there is no acceptable block, they will return with a failure. If the found block is larger than the requested size, it will be split and the remaining block will be inserted into the list. Corresponding with the deallocation case, the free block will be merged with the adjacent blocks if they are free. If they are free, they will first be extracted from the list, and merged with the free block to create a larger-size block. The new larger block will be inserted into the free list.

In general, a problem with *First-fit* is that frequent splitting occurs near the head of the free list which results in the accumulation of lots of small blocks near the head of the list. Those small blocks increase searching time because searching needs to pass through them each time, and it causes high fragmentation. Consequently, the conventional *First-fit* is not suitable for an application that allocates and deallocates many different sizes of blocks frequently. However, as with *Best-fit*, it can be more applicable if implemented using more sophisticated data structures. In practice, *First-fit* algorithms using either address-ordered or FIFO exhibit lower fragmentation than *First-fit* based on LIFO [Wilson et al., 1995a].

- **Next-fit**

Next-fit allocation algorithms are one of the variants on *First-fit* algorithm that employ a roving pointer for allocation [Knuth, 1997]. The algorithms keep track of

the pointer, which records the position of where the last search was satisfied, and they will be used as the beginning point for the next search. The algorithms can also be implemented with: address-ordered, FIFO and LIFO mechanisms, like the other convenient mechanisms.

Theoretically, *Next-fit* reduces the average search time under a single linear list; however, it tends to get worse locality because it always searches each free block before examining the same block again due to the roving pointer.

Furthermore, the roving pointer cycles through the free block lists regularly, and is likely to accumulate objects in memory with different sizes and lifetimes from different phases of the application's execution. Consequently, it has been shown to cause more fragmentation than other sequential fit allocation algorithms. In particular, *Next-fit* with LIFO mechanism allocators has significantly worse fragmentation than address-ordered *Next-fit* algorithms [Wilson et al., 1995a]. However, *Next-fit* is the best solution to minimizing the mean response time in a shared memory symmetric multiprocessor for hard real-time system applications [Banús et al., 2002].

In terms of allocation and deallocation procedures, *Next-fit* allocation algorithm originals are almost equivalent to *First-fit* algorithms, except for the starting point of search.

In summary, sequential fit algorithms are implemented using a single linear list, which consists of doubly-linked or circularly-linked lists with very different policies in practice, such as *address-ordered*, *FIFO* or *LIFO* policies. *First-fit* and *Best-fit* are based on either an address-ordered or *FIFO* policy and they seems to work well. However, a problem with sequential fit algorithms is that they are not scalable because as the number of free blocks grows the search cost linearly increases.

Sequential fit algorithms can be combined with other mechanisms, such as *optimal-fit*, *half-fit* or *worst-fit*. In particular, *worst-fit* mechanisms search for the largest free block that is large enough to satisfy the request of memory allocation because it attempts to make the free block as large as possible in order not to accumulate small fragmentations; however, in practice, this algorithm seems to work badly.

Overall, the algorithms allocate memory blocks in $\mathcal{O}(n)$ in the worst-case, where n is the size of heap. The algorithms are not predictable and not acceptable for

real-time systems even though the sequential fit mechanisms are implemented using various types of data structures rather than linear lists in order to improve scalability.

2.3.2 Segregated Free Lists

Most of the modern memory allocation algorithms, such as *DLmalloc*, *tcmalloc*, *TLSF*, *Hoard*, employ segregated free list mechanisms which use an array of free block lists. This is because their use reduces search time, although it does cause small internal fragmentations. The algorithm generally uses sizes that are to the power of two apart so that each size class holds free blocks of a particular size.

Typically, the algorithms rounds the requested size up to the closer size class or nearest size class if that is empty. After that, the memory algorithms search for a free block of the requested size that is large enough to satisfy the request in a certain size class or for any slightly smaller size that is still larger than any smaller size class.

In terms of the deallocation, the allocators insert a free block into the free list for the given size when the used block is freed.

Wilson [Wilson et al., 1995b] indicates that algorithms that employ segregated free lists can be divided into two categories: *simple segregated storage* and *segregated fit*.

- **Simple Segregated Storage**

Simple segregated storage is one of the simplest allocators that uses an array of free lists. No splitting and no coalescing of free blocks are required. These characteristics distinguish simple segregated storage mechanisms from buddy systems. An advantage of this approach is that no headers are required. This can decrease memory consumption as the headers introduce overheads – the headers usually increase memory consumption by 10% to 20% [Zorn and Grunwald, 1992] – which is particularly important when the average requested size is very small.

As there is no required splitting or coalescing of blocks and maintaining of headers, the algorithms are usually fairly fast; especially when the blocks of a given size

are requested in a short time repeatedly. As a simple policy, it achieves a time complexity of $\mathcal{O}(1)$.

However, a problem with the algorithm is that it induces large external fragmentation, which is proportional to the maximum amount of memory used by the allocator times the maximum of the block size requested by the application. It also suffer from internal fragmentation.

- **Segregated Fit**

Segregated fit algorithms use arrays of free lists. Each array holds free blocks within a certain size class. The allocator is faster than a single free list for most cases, as it searches the free list for the appropriate size class when the application requests memory. After choosing a certain array within a size class, it searches for a free block in the array by a sequential fit search. If there is no free block, the algorithms attempt to search for a larger block in the closest array repeatedly until finding a larger block, with the larger block then being split and the remainder being inserted into a particular array. Such algorithms can be categorised as follows: *Exact Lists*, *Strict Size Classes with Rounding* and *Size Classes with Range Lists*.

Exact Lists In order to support this algorithm, the allocator needs to have a huge number of free lists to have each possible block size. In practice the allocators only use the exact lists within small size classes to reduce a very large number of free lists.

Strict Size Classes with Rounding This mechanism rounds the request size up to the closer sizes in the size class sets, even though it wastes some space as internal fragmentation. One advantage of this algorithm is that it can maintain all blocks on a size list that are of the same size exactly.

Size Classes with Range Lists This is one of the most widely used mechanisms, which allows free lists to hold blocks of slightly different sizes.

In summary, segregated free lists are used with other mechanisms such as *First-fit* or *Best-fit* to search for a certain free list. If the allocator finds a particular size

list, it searches for a freed block in the list based on those mechanisms. Regarding the optimization, the algorithms employs boundary tags [Knuth, 1997] [Standish, 1980] to support faster splitting and coalescing of free blocks.

Consequently, the algorithms can be achieved a time complexity of $\mathcal{O}(1)$ in the worst-case. However, these algorithms are not suitable for real-time system due to large external fragmentation.

2.3.3 Buddy Systems

Buddy systems are particular variants of segregated lists using size classes with rounding. In the system, the whole heap area is theoretically divided into two areas, with these areas being further divided into two smaller areas, and so on in a simple buddy case. The standard algorithm of (binary) buddy system can be achieved in a time complexity of $\mathcal{O}(\log_2 n)$ in the worst case, where n is the maximum size of the heap.

One advantage of this algorithm is that it contains all blocks on a size list that are of exactly the same size. For instance, if a size of a free list is 4K bytes, all blocks contained in the list are a size of 4K bytes. The only difference between this algorithm and other segregated lists algorithms is the support of limited splitting and coalescing of free blocks using a certain function, e.g. a power of two function, as followa:

$$i = \lfloor \log_2(r) \rfloor \tag{2.2}$$

In this case, r is used to calculate a certain index which represents a particular free list within the size class to be used for either inserting a free block or searching for a free block.

In terms of allocation in a simple buddy, it searches for a free block within a particular array obtained by function 2.2. If there is no free block, the algorithms try to find a larger block in the closest array iteratively until finding a larger block. If a block has been found in some higher array, the block will be extracted from the list and recursively split into the size power of two logarithmically to be smaller, but the size of the block is still large enough to satisfy the request. The remaining

blocks generated in the process of splitting are inserted into the corresponding free lists.

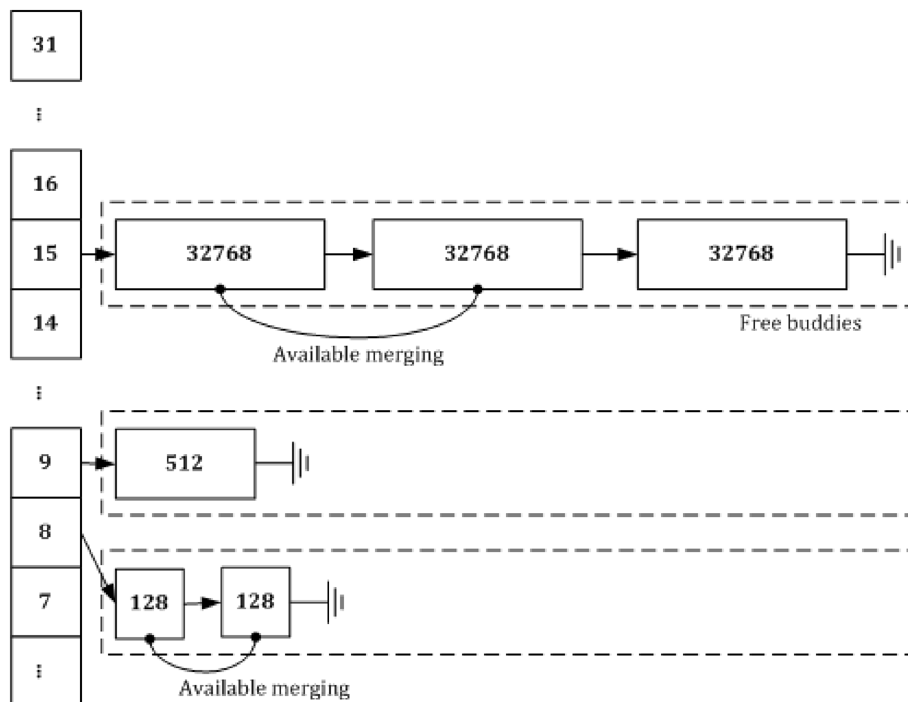

Figure 2.2: Logical view of Buddy system

When the application frees a used block, it uses the function 2.2 to find the certain array within size classes. The array obtained by the function 2.2 is examined to see if it holds adjacent blocks that are already free. If this is the case, the algorithm merges the freed block with the adjacent blocks to create a new block of double the size. After that, this operation iteratively repeats until there is no free block that meets the condition described above. Such a simple algorithm is called a binary buddy system. Wilson [Wilson et al., 1995b] addressed how buddy systems can be typically divided into four categories, as follows: *Binary Buddy*, *Fibonacci Buddy*, *Weighted Buddy* and *Double Buddy*.

- **Binary buddy**: As seen above, binary buddies are one of the simplest variants of buddy systems. In terms of these algorithms, the sizes of all blocks are a power of two, with each size being split into two equal parts and merged into one double size. These characteristics make pointer computations simple. For this reason, it has generally been considered as a real-time allocator. This

algorithm achieves in a time complexity of $\mathcal{O}\left(\log_2 \frac{m}{n}\right)$ in the worst case, where m is the maximum size of the heap, and n is the maximum size of used memory by the application. However, a problem with this algorithms is that internal fragmentation is relatively high around 28% [Knuth, 1997].

- **Fibonacci Buddy**: Knuth [Knuth, 1997] proposed using Fibonacci numbers as buddy block sizes instead of a power of two, resulting in reducing internal fragmentation compared with binary buddies [Hirschberg, 1973]. Due to the sizes of all blocks being the sum of the two previous numbers, a block can only be split where sizes are in the numbers as well. A problem with this algorithm is that the remaining block is likely to be useless if the application allocates many blocks of the same size [Wilson et al., 1995b].

- **Weighted Buddy**: These algorithms [Shen and Peterson, 1974] [Page and Hagins, 1986] allow dealing with size classes in two ways. The size of all blocks is 2^n and $3 \cdot 2^n$ for all n so that the size classes contain the powers of two, and there is a size that is three times a power of two in between each pair of consecutive sizes, i.e. 2, 3, 4, 6, 8, 12, 16... An advantage with weighted buddy is that average internal fragmentation is less than the other buddy systems. However, a problem with them is that external fragmentation is larger than other buddy systems [Chowdhury and Srimani, 1987].

- **Double Buddy**: [Page and Hagins, 1986] proposed double buddy systems, which are one of the variants of weighted buddies. Double buddy algorithms reduce the amount of fragmentation compared to weighted buddies. The main difference with weighted buddy is the splitting rule. All blocks can only be split in half exactly, so it results in the size of all blocks being a power of two, as in the binary buddies.

All the above algorithms can be achieved in a time complexity of $\mathcal{O}(\log_2 n)$ in the worst-case. However, these algorithms are not suitable for real-time system due to large internal fragmentation.

2.3.4 Indexed Fit and Bitmapped Fit

In the case of sequential fit, a linear search is required to find a free block. Segregated fit also needs a linear search to traverse an array within a certain size class. An *indexed fit* mechanism is one of the secondary mechanisms used to accelerate search efficiencies of allocator algorithms; it is used in combination with other mechanisms with a variety of data structures. Typically, these mechanisms use more fine-grained indexing data structures to keep track of free blocks within size-based policies. At a high-level abstraction, all of the memory allocator algorithms seem to be one of the variants of *indexed fit* because all of them keep track of which parts of the memory area are in use or which are not.

For instance, *Bitmapped fit* is derived from the *indexed fit* mechanism. This algorithm keeps track of which parts of arrays are in use and which parts are not. In particular, each bit corresponds to a free block, array or other data structure based on their mechanisms. However, a *bitmapped fit* scheme has rarely been used because historically its searching time is normally slow; but now modern processors support bit search instructions and searching now only takes few clocks cycles of the processor.

An advantage of a bitmap structure is that it can be implemented by an extremely small amount memory, for example having one or two words in some cases. If the implementation of an algorithm requires a large amount of memory then the data structures have a greater probability of being interleaved across more than one node in a ccNUMA architecture. Having a bit map per node is also able to improve the locality of searching itself.

Half-fit [Ogasawara, 1995] and TLSF [Masmano et al., 2003] are examples of an *indexed fit* with *bitmapped fit* schemes. They use bitmaps to record which areas are free, and exploit bit search instructions. Another example of an *indexed fit* scheme is *Fast fit* [Stephenson, 1983], which employs a Cartesian tree to sort on both size and address.

2.3.5 Hybrid Policies

As seen above, a single allocation algorithm seems to have several disadvantages, so most modern memory allocation algorithms have combined several algorithms to accelerate searching and insertion speed. Most representative algorithms used by modern allocators are based on segregated fit, buddy systems, and sequential fit algorithms. In the subsection, we will discuss some of the most widely used hybrid allocation algorithms in general-purpose systems and real-time systems.

2.3.5.1 Doug Lea(DLmalloc)

The *DLmalloc* allocator, which was been designed and implemented by Doug Lea [Lea, 1996], is the most popular allocators, and has been incorporated into many memory management algorithms; [Gloger, 2006] [Douglas, 2011] [FSF, 2012a] [FSF, 2012b] are all variants of *DLmalloc*. In the later version of the allocator, the major strategy and policy remain unchanged, despite it having changed the mechanism, e.g. it employed bitmaps to search for available blocks instead of the iterative searching used in the previous version. Currently, it employs a combination of several mechanisms, depending on the requested size. It also implements the *good-fit* policy with the segregated size-classes mechanism. *DLmalloc* is designed to use two types of data structure, depending on the size of the memory blocks. Figure 2.3 (below) shows the *DLmalloc* structure.

For small-size block allocation, the allocator uses a large number of fixed-width arrays called *smallbins*, which hold free blocks with sizes less than 256 bytes. Each bin contains free blocks of all the same size, spaced 8 bytes apart. Assuming that the given requested size of memory blocks is less than 256 bytes, the allocator searches for available blocks in the bins using smallest-first, *best-fit* order.

If the given requested size of a memory block is greater than 256 bytes and smaller than a threshold, which is usually 256K bytes, the allocator attempts to find available blocks in an array called *treebin*, which consists of a *trie* [Fredkin, 1960], a particular data structure. Unlike smallbins, treebins store a range of bin sizes with two bins per power of 2. As seen in figure 2.3, nodes are in the trie data structure, with each node being a smallbin, containing all the blocks of that exact

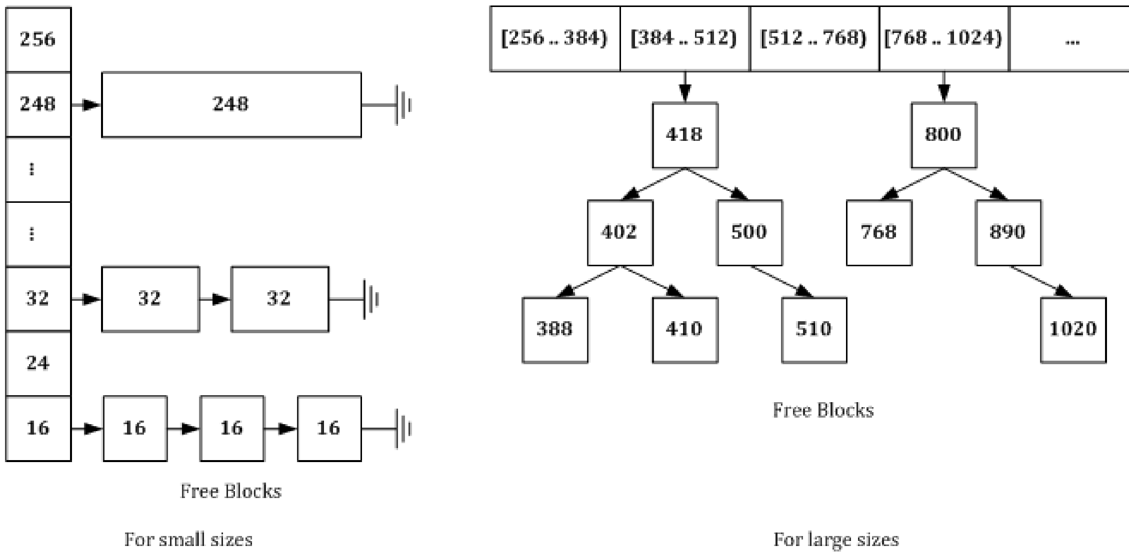

Figure 2.3: Structure of *DLmalloc*

size. For requests above the threshold, the allocator forwards the requests to the underlying OS through the *mmap()* system call.

As discussed, the allocator can achieve a time complexity of $\mathcal{O}(1)$ for searching for a small block with a size less than 256 bytes, but in *treebins*, it can be $\mathcal{O}(m)$, in which m is the depth of the *trie*, in the worst-case.

2.3.5.2 Half-Fit

Half-fit has been proposed by [Ogasawara, 1995], which is known as the first allocator to perform in constant execution time using the *bitmapped fit* mechanism for the management of free blocks, although *bitmapped fit* has not been used for allocators due to being too slow generally [Wilson et al., 1995b]. In *Half-fit*, bitmaps are used to keep track of empty lists, and bitmap search instructions are used to find set bits in the bitmaps. It can achieved a the time complexity of $\mathcal{O}(1)$ on most modern processors.

Half-fit employs a single level of segregated lists in which free blocks of variable size are linked. It takes free blocks of a given size from a free list in which blocks always satisfy the request. Figure 2.4 (below) shows an example of the structure of *Half-fit*. There are three blocks of sizes 156, 250, and 200 bytes in the order where blocks belong to the range $[2^5,2^6)$.

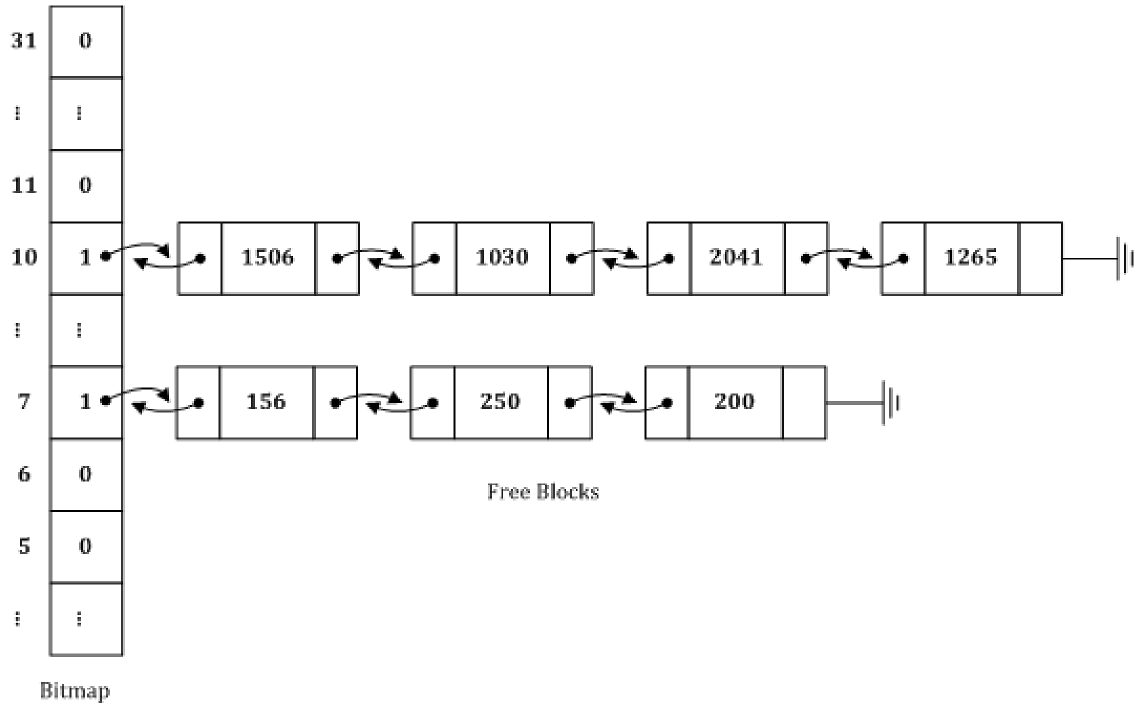

Figure 2.4: Structure of Half-fit

It employs a particular allocation technique in order to avoid searching using bitmaps because of its linear search time. Assume that the given size of a memory request is r, and index i can be calculated by the following equation:

$$i = \begin{cases} 0 & \text{if } r \text{ is } 0 \\ \lfloor \log_2(r-1) \rfloor + 1 & \text{otherwise} \end{cases} \tag{2.3}$$

The index i indicates the free block lists whose sizes range from 2^i to $2^{i+1} - 1$. The given sizes of allocation requests to the free lists i are always between $2^{i-1} + 1$ and 2^i; thus, the given allocation requests can be satisfied by any block on the list. After calculating index i, a free block is taken from the free list indexed by i. If the free list is empty, the next free list whose index is closest to i will be examined.

If the size of an allocated block is larger than the requested allocation size, a free block in the free list of the size class is split into two blocks of sizes r and r' before providing for allocation. After that, the remaining block r' is inserted into the corresponding free list.

For deallocation, freed blocks are immediately coalesced with adjacent blocks if those blocks are also free. After a free block is merged, and the size of the block is

r, the new block will be inserted into the head of the free list indexed by i. For the computation of i. *Half-fit* uses the following equation:

$$i = \lfloor \log_2 r \rfloor \tag{2.4}$$

Coalescing with adjacent free blocks can be achieved with a time complexity of $\mathcal{O}(1)$ because adjacent memory blocks are doubly linked with each other.

Half-fit is acceptable for use in real-time systems considering just its time complexity. In addition, it shows better performance than binary buddy algorithms, which result in data cache misses and TLB entry misses as buddies are distanced from each other when free blocks are large. Unlike the binary buddy algorithm, it always merges adjacent free blocks.

As discussed above, *Half-fit* provides a very good worst-case response time, $\mathcal{O}(1)$, but the algorithm is not ideal for real-time systems as it suffered from a considerable theoretical internal fragmentation by *incomplete memory use* based on the splitting policy [Ogasawara, 1995], as many requests of allocations are performed that are not close to the power of two [Crespo et al., 2006].

2.3.5.3 TLSF

The Two-Level Segregated Fit (TLSF) allocator [Masmano et al., 2003] is an improvement over *Half-fit* [Ogasawara, 1995]. Employing two levels of segregated lists distinguishes *TLSF* from *Half-fit*. *TLSF* is also an allocator designed for real-time system. In *TLSF*, two levels of a segregated array of free lists are used, with each list keeping free blocks within a size class so that it can reduce internal fragmentation theoretically.

The first-level of array (FLI) divides free blocks in classes that are a power of two apart, such as 2, 4, 8, 16, 32, and so on. The subsidiary second-level arrays divides each first-level class linearly by a user configurable variable called SLI. *TLSF* structures are shown in Figure 2.5.

The main objective of *TLSF* is to provide bounded response time in memory allocation and deallocation, whatever the memory size is. In allocation, *TLSF* exploits equations 2.5 [Masmano et al., 2008b], wherein the given size of a block calculates

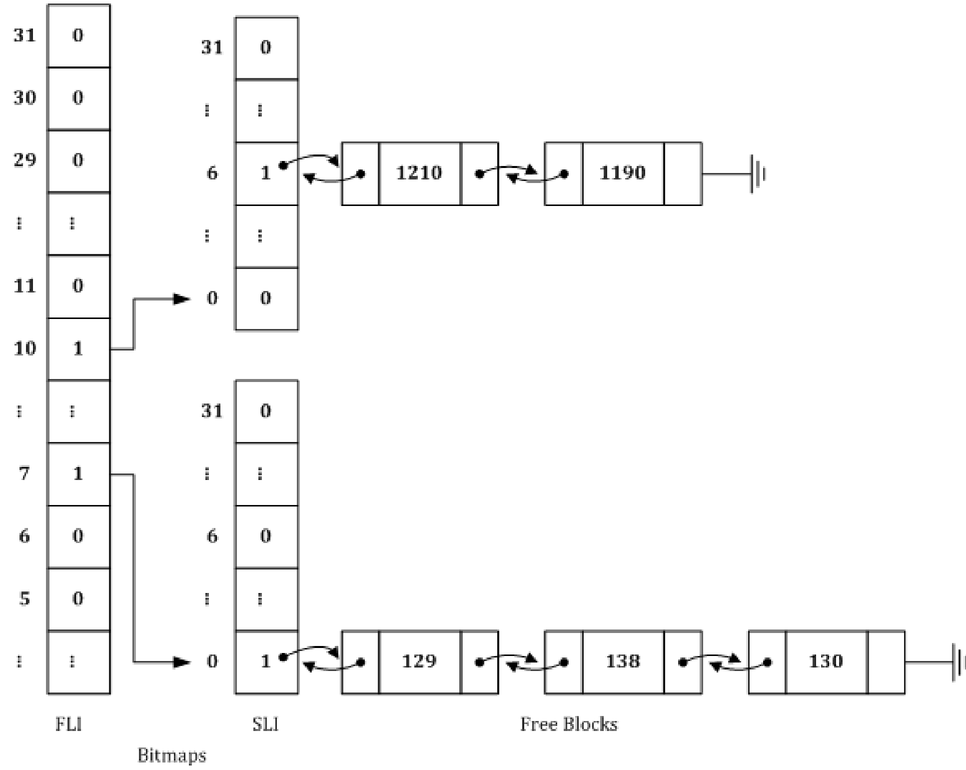

Figure 2.5: Structure of TLSF

the indexes of the two arrays that point to the corresponding segregated list.

$$i(f, s) = \begin{cases} f = \lfloor \log_2 (r + 2^{\lfloor \log_2(r) \rfloor - SLI} - 1) \rfloor \\ s = \lfloor \frac{(r + 2^{\lfloor \log_2(r) \rfloor - SLI} - 1 - 2^i)}{(2^{f - SLI})} \rfloor \end{cases} \tag{2.5}$$

The first-level index (FLI) f can be calculated as the position of the last bit set of the size, in which the bit sets to one. This index indicates memory blocks corresponding to their size. Each FLI position points to a certain size class. For instance, FLI_7 points to size classes ranging from 128 bytes to 255 bytes; FLI_8 indicates size classes ranging from 256 bytes to 511 bytes. The second-level index s can be computed by the above equation. However, those equations can be efficiently implemented using shift and bitmap search instructions, which are available on most modern processors. Each position indicates a memory block within similar sizes.

Assume that the given size of a memory request is r, the index $i(f, s)$, which is used to take the head of the free list holding the closest class list, will be calculated by the above equation 2.5. If the block is found at the free list indexed by i, only

that block will be removed from the head of the free list and returned. If searching the free block at i fails, the next free list whose index is closest to i will be examined. After that, a free block in the closest free list of larger sizes will be split into two blocks of sizes r and r'. The remaining block r' is inserted into the corresponding free list.

Corresponding with the deallocation case, freed blocks are immediately coalesced with adjacent free blocks. If the adjacent blocks are free, the adjacent blocks will be removed from the segregated list and merged with the current block. Finally, the new block will be inserted into the head of the free list indexed by i. For the computation of i, *TLSF* uses the following equations 2.6 [Masmano et al., 2008b]:

$$i(f,s) = \begin{cases} f = \lfloor \log_2 r \rfloor \\ s = \lfloor \dfrac{(r - 2^f)}{(2^{i-SLI})} \rfloor \end{cases} \tag{2.6}$$

As discussed above, *TLSF* uses the request size of the memory block to calculate the appropriate position indexed by the FLI and SLI based on equations 2.5 and 2.6. As the result, operations can be achieved with a time complexity of $\mathcal{O}(1)$.

However, *TLSF* uses a single heap shared by all threads in a process, meaning that it suffers from heap contention on multi-threaded environments and it is not scalable. It originally only supported a fixed size of the memory pool, whose size cannot be grown, in the initial implementation of TLSF.

The latest implementation of TLSF no longer has a constant execution time allocator, despite the fact that its policy can still be achieved in the constant time. This is because it has changed its algorithm. The latest version of *TLSF* [Masmano, 2012] can have multiple memory pools, and an additional memory pool will be created when existing memory pools are full by automatics system calls of *brk()* or *mmap()* in UNIX.

The problem occurs when the last used block at each memory pool needs to be released and the allocator has used multiple memory pools. After freeing the last used block, the memory pool which contains the last used block will be in a completely unused/freed state, with TLSF attempting to merge adjacent memory pools like deallocation of the memory block scheme. Under this policy, the time of

the merging process is able to increase linearly, so if there are a number of freed memory pools (as many as N), it loops N times to merge each other iteratively. Therefore, the policy results in *TLSF* having a time complexity of $\mathcal{O}(n)$ [Masmano, 2012].

2.3.5.4 tcmalloc

tcmalloc [Sanjay Ghemawat, 2010] is one of the highly scalable allocators that combines a global heap and per-thread heaps within a similar discipline that is used in some modern allocators for multiprocessors. For small-size allocations ranging from 8 bytes to 32 kilobytes, *tcmalloc* assigns each thread a thread-local heap used by the owner at allocation time, so memory allocations for small blocks do not require thread synchronization. Sometimes, memory blocks are moved from a global heap into the per-thread heap if this is needed, and periodic garbage collections are used to migrate memory back from a per-thread heap into the global heap.

tcmalloc also employs a global heap that is shared by all threads for allocating large memory blocks from over 32K bytes up to 1M bytes. The thread synchronization, a spin-lock, is used to provide mutual exclusion. If applications request allocation of a large block of memory over 1M bytes, *tcmalloc* passes the request to the underlying OS using OS APIs. Figure 2.5 shows the *tcmalloc* structure.

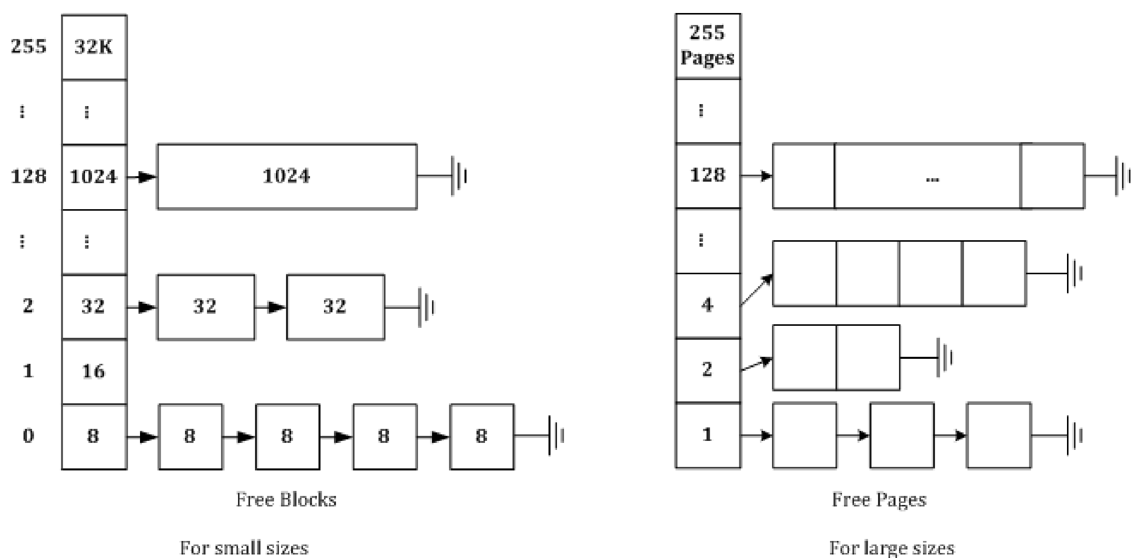

Figure 2.6: Structure of *tcmalloc*

Assume that the given request size of a memory block is smaller than 32K bytes, the allocator just checks to see if it can satisfy the request from the per-thread heap. If this fails, *tcmalloc* checks the global heap using a lock for synchronization. If the free list is empty, the next free list will be examined, and so forth. If the global heap has a free memory block of a sufficiently large size, it will be divided into smaller blocks and one is served, with the remaining free memory block being inserted into one of the free lists in the global heap.

In terms of the deallocation, a thread frees a small block, with the memory block being inserted into the list in the per-thread heap within the corresponding size. If the size of a free list exceeds a particular threshold (2MB by default), the allocator transfers some of the free block back to the global heap.

In terms of the time complexity, each size of a small memory block in the per-thread heap maps to one of 170 properly allocable size-classes, which are divided by 8 bytes, larger sizes by 16, even larger size by 32 bytes, and so on. Thus, all of the request sizes are rounded up to the closest larger size. Each position in the global heap corresponds with the number of pages of the memory block, which is rounded up to an aligned size of 4K bytes. In figure 2.5, the index i from 0 to 254 is a free list that consists of i pages. Note, *tcmalloc* uses the request size to calculate the appropriate index in the per-thread heap or global heap, and this can be achieved with the time complexity of $\mathcal{O}(1)$.

However, *tcmalloc* does not address false sharing, since two small objects assigned to different threads can have close memory addresses [Ferreira et al., 2011], and *tcmalloc* does not release the memory pool despite avoiding blow-up, which is one of the most important issues with parallel allocators [Kaminski, 2009]; thus, it can lead to unbounded memory consumption.

2.3.5.5 Hoard

The *Hoard* allocator [Berger et al., 2000] is designed to be fast and scalable in multi-threaded applications running on multiprocessor environments. It is one of the segregated size-class allocators, like other modern allocators: *TLSF*, *tcmalloc* and *DLmalloc*. *Hoard* employs per-processor heaps and one global heap to avoid

heap contention. Each thread has a private heap to keep memory blocks smaller than 256 bytes, and can also access both its per-processor heap and the global heap. Each per-processor heap can be shared by a group of threads. Figure 2.7 shows the structure of *Hoard*.

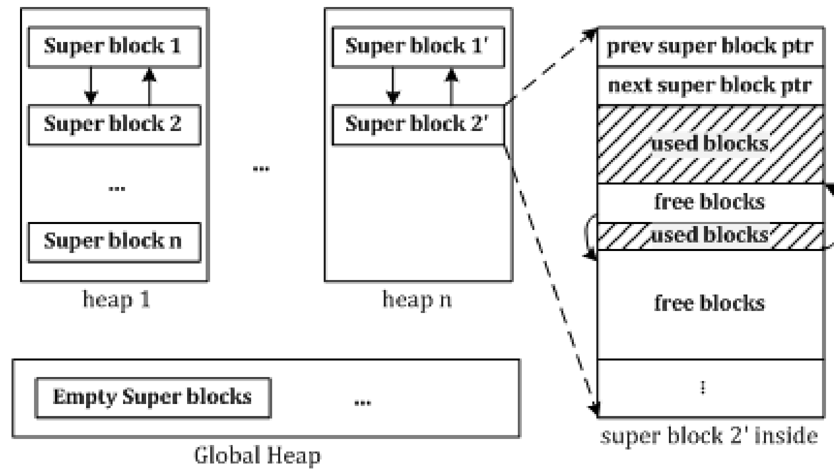

Figure 2.7: Structure of *Hoard*

Assume that the given request size of a memory block is smaller than 256 bytes, the allocator searches available blocks in its private heap per thread. Next, it will search for available blocks in the per-processor heap. In a per-processor heap, the allocator takes memory blocks from the system in chunks called *superblocks*, which is an array of some groups of blocks that contains a free list. If the per-processor heap is fully used, the allocator just uses the global heap, which is shared by all threads. In this case, the thread locks the per-processor heap, and it will be released when the allocator transfers a new superblock from the global heap to the per-processor heap.

Hoard achieved an allocation with time complexity of $\mathcal{O}(n)$, where n is the number of superblocks, because it needs to search sequentially for the superblock of the thread. As discussed, there is the heap contention between threads on the shared per-processor heap, even though *Hoard* employs two times as many per-processor heaps as the number of processors in the system, and on the global heap as well. *Hoard* also exploits a particular mapping function between threads and processors, with thread being reassigned to other processors. It finally leads to more cache misses and TLB misses occurring due to the disorder of node-based data locality.

2.4 Summary

Dynamic storage allocation algorithms are essential for modern application. They allow memory resources to be used more efficiently. Most modern general-purpose operating systems provide the functionality of dynamic memory allocation, but these features are not optimized, so they can introduce significant problems such as fragmentation or expensive cost of searching.

Many memory allocation algorithms have been proposed over the past fifty years. Each algorithm is able to satisfy its defined objectives such as reducing fragmentations or providing a very low response time.

	Allocation	Deallocation	Wasted memory	ccNUMA support
Sequential Fit	$\mathcal{O}(n)$	$\mathcal{O}(1)$	Acceptable	No
Segregated Fit	$\mathcal{O}(1)$	$\mathcal{O}(1)$	Unacceptable	No
Buddy systems	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(k)$	Unacceptable	No
DLmalloc	$\mathcal{O}(m)$	$\mathcal{O}(1)$	acceptable	No
tcmalloc	$\mathcal{O}(1)$	$\mathcal{O}(1)$	Unacceptable	Yes
Hoard	$\mathcal{O}(n)$	$\mathcal{O}(1)$	acceptable	No
Half-fit	$\mathcal{O}(1)$	$\mathcal{O}(1)$	Unacceptable	No
TLSF	$\mathcal{O}(1)$	$\mathcal{O}(n)$	acceptable	No

Table 2.1: Worst-case time complexity of algorithms

Table 2.1 shows the time complexity of the allocation and deallocation in the worst-case of some the the main algorithms. In the Table, n is the size of the heap, m is the depth of *trie*, and k is the depth of the buddies. In terms of real-time systems, *Segregated Fit*, *tcmalloc*, and *Half-fit* are the only acceptable algorithms that offer a constant execution time for both (allocation/deallocation) operations. Also, buddy systems showing $\mathcal{O}(\log_2 n)$ are possibly usable for real-time systems except they exhibit large fragmentation, while others are not acceptable for real-time systems due to unbounded execution time. Also, all algorithms except *tcmalloc* do not support ccNUMA architecture systems. Consequently, existing memory management algorithms mentioned above have not been considered acceptable for the target systems of this thesis.

Chapter 3

nMART: A ccNUMA-aware Dynamic Storage Allocation Algorithm

In this chapter, we introduce a new ccNUMA-aware dynamic storage allocation algorithm called *nMART*, which has been designed to support real-time systems on ccNUMA architecture systems. Its overall objectives are to be predictable, have low fragmentation, to have bounded response time with a node-based memory management policy.

The chapter discusses how the algorithm can achieve these objectives. We start by presenting the design principles on which nMART is based. Then, in Section 3.2, we give a top-level view of our approach. A key observation of this thesis is that current operating systems do not provide an accurate metric to measure distances between nodes in a ccNUMA architecture. This can have a crucial impact on performance. Consequently, in Section 3.3.3, we propose a new model and consider how it can be supported inside an operating system kernel. This is followed, in Section 3.4, by detailed consideration of the user-level nMART management algorithms.

3.1 Design Principles

Our target architecture systems are high-performance real-time systems comprised of multiprocessors and multicores. The requirements of real-time applications are different from general-purpose applications. Real-time applications require deterministic response time, while a low average response time is more important for general-purpose applications. For this reason, make explicit a set of design criteria for our algorithm. These are given below.

A more accurate measure of node distance Remote memory access through an interconnect takes longer than the time of local memory access. For example, the latencies of remote memory access appear to be two times larger than latencies of local memory access in current implementations of ccNUMA architecture systems [Majo and Gross, 2011]. Many researchers have proposed various techniques to improve the application performance running on ccNUMA architecture systems by increasing data locality with their own approaches [McCurdy and Vetter, 2010] [Marathe and Mueller, 2006] [Ogasawara, 2009] [Tikir and Hollingsworth, 2008]. Their papers require discovering the underlying architecture design. Without such discovery, it is impossible to improve the performance of many applications, including memory allocators.

Current operating systems generally measure ccNUMA node distance using a very simple metric, which is insufficient to reflect accurately the resource hierarchy. This degrades the performance of the current memory allocation algorithms. Therefore, we propose a new model to measure real ccNUMA node distances. Under the model, we evaluate all of the node distances between nodes, re-establishing the node order of each node from the closest node to the farthest node. This information can then be used at allocation time to improve the performance. We will discuss these details in Section 3.3.1.

Multiple heaps Using a single global heap results in increasing lock contentions on the heap, as threads sharing the heap must synchronize their access to avoid heap corruption. Usually, a single global heap is protected by a single global lock. The problem with a single global lock is that applications using

the lock have poor scalability and significantly higher access time waiting to acquire the lock.

nMART employs private heaps with an ownership strategy to reduce heap contentions. These private heaps are isolated from non-owner tasks for allocations, however, the algorithm allows threads to release a memory block allocated from other threads' private heaps. In this case, *nMART* employs a deferred release policy. If a thread releases its own blocks, the given block will be released and merged with its adjacent blocks immediately. Multiple coalescing strategies, will be discussed in Section 3.4.4.

Multiple strategies for different sizes of blocks Many modern allocation algorithms use different strategies for allocating blocks of different sizes, in order to reduce fragmentation and improve space efficiency for small blocks. In contrast, *Half-fit* and *TLSF* exploit only a single strategy for all block sizes. The same strategy for all block sizes is easy to use and implement, with their worst-case execution time also being easy to analyze. However, a single size tends to cause significant wasted memory in small-size blocks of less than 512 bytes because they round the requested size up to the closest size in the segregated size classes. This is illustrated in Figure 3.1. If an application requests allocating 68 bytes of a memory block, *TLSF* rounds the requested size up to 80 bytes. Even if the application only uses 68 bytes, the allocator provides 80 bytes so that it has caused internal fragmentation of as much as 12 bytes. The figure shows the percentage of internal fragmentation calculated using equation 2.1.

For the above reason, *nMART* employs three different allocation strategies, depending on the size of the request. In particular, the exact fits strategy is used for a size of request less than 512 bytes, with two level segregated lists being employed for normal sizes of blocks less than a threshold (2M bytes by default), and for request exceeding the threshold, *nMART* passes the request to the underlying operating system via the *mmap()* system call.

Tracking memory block profile Most modern operating systems employ virtual

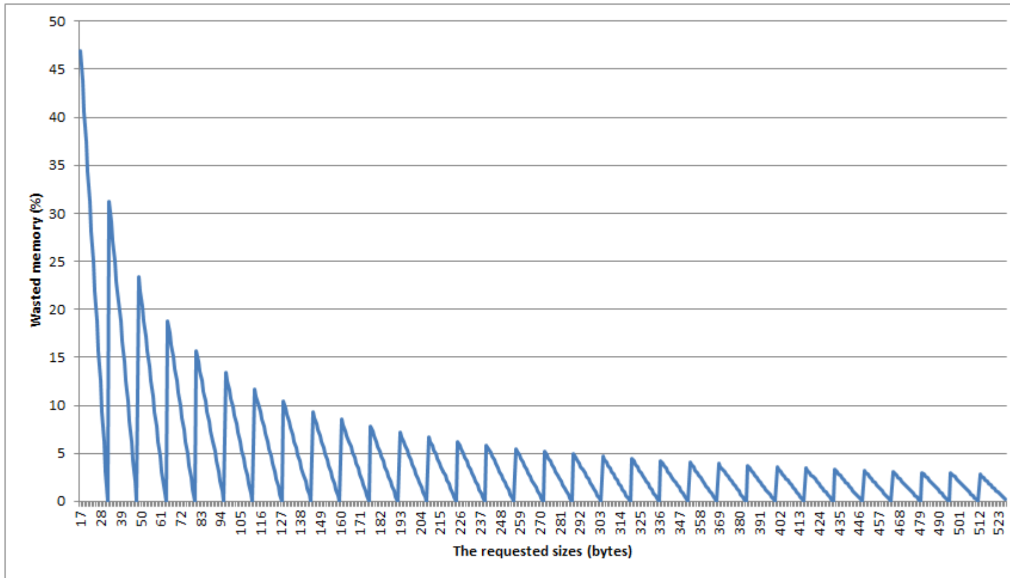

Figure 3.1: The wasted memory of TLSF in small sizes of blocks

memory management which maps virtual to physical memory. The OS provides virtual memory regions when applications request memory, but this virtual memory is not mapped to the physical memory immediately for performance reason. When the application first accesses the memory, the OS maps virtual memory regions to the physical memory regions through the policy called *first touch*. With this policy, the memory allocation is able to allocate the physical memory from the local ccNUMA node.

However, this is still insufficient to ensure minimizing remote memory accesses. For instance, assume that thread *T1* executing on node *N1* has allocated some memory *M1* and later releases *M1*, and then thread *T2* executing on node *N2* requests some memory. In this case, general memory allocators typically reuse *M1* to speed up allocation. Unfortunately, *M1* has already been mapped to the physical memory of *N1*. Now the reused M1 is physically remote memory to *T2*.

To maximize node data locality, *nMART* keeps track of which ccNUMA node each memory block belongs, with this information being stored in the header of the memory block. Based on this information, the allocator is able to allocate memory blocks from the local node as often as possible.

Search strategies As mentioned above, we employ multiple strategies for different sizes of blocks. *nMART* attempts to allocate as small a block as possible, which is still large enough to satisfy the request. In terms of small block allocations, the strategies are based on the *best-fit* policy implemented by *exact-fit* mechanisms. This approach tends to reduce the fragmentation in small sizes of blocks caused by rounding up the request size [Johnstone and Wilson, 1998].

For larger blocks, *nMART* employs the *good-fit* policy implemented by segregated lists, which use an array of free lists. Each array holds unsorted free blocks within a size class. In this case, the *good-fit* mechanism is used to determine a particular free list, the allocator attempts to find a free block of the given size in the determined list, and then extracts the first free block by the first-fit mechanisms (if it is available).

Coalescing strategies *nMART* attempts to coalesce adjacent free blocks to create a larger free block immediately when a target block is freed, even though general-purpose memory allocators tend to use the deferred coalescing techniques, or even not coalesce at all, in order to improve the overall performance. As discussed in Section 2.1.2, deferred coalescing is a useful strategy for systems where applications reuse short-lived memory blocks of the same size repeatedly because it is able to reduce overheads of repeatedly splitting and coalescing the same block. However, it tends to make it difficult to fully analyse the worst-case execution time of the memory allocation algorithms.

Use of the virtual memory Most modern operating systems support virtual memory managements. These techniques are able to use the physical memory more efficiently, and allow applications to execute, which are larger than the size of physical memory. Moreover, these management techniques have achieved high access rates and low cost, thereby supporting a memory hierarchy. However, it is extremely difficult to analyse the worst-case execution time of memory allocation algorithms going through virtual memory management due to the non-deterministic execution time of demand paging, so the use of virtual mem-

ory management in real-time systems is usually avoided. However, the trend towards high-end embedded systems offers virtual memory management due to their significant advantages.

With the trends, some researchers attempt to achieve predictable virtual memory management for real-time systems. [Zhou and Petrov, 2011] have proposed a new page table organisation for real-time systems, which requires less memory, and has a fast deterministic page table traversal based on hardware. [Puaut and Hardy, 2007] have also proposed a predictable paging mechanism achieved at compile-time.

However, these approaches are limited and cannot be applied in ccNUMA architecture systems because they are only appropriate for small embedded systems or a single task. To the best of our knowledge, there is no acceptable real-time virtual memory management for ccNUMA systems.

nMART cannot be directly involved in virtual memory management, as it is a user-level memory allocator. For this reason, we will not discuss virtual memory management techniques further in this thesis. This is an area for future work. In order to minimize the effect of virtual memory management, we disable swapping, thereby, locking memory in core. Furthermore, we have assumed that the target systems have a huge amount of available physical memory, and the virtual memory is also supported by the memory management unit (MMU). Therefore, the allocator is able to call *sbrk()* / *brk()* / *mmap()* system calls to make a request for additional memory when the initial memory pool is exhausted.

In summary, *nMART* is designed to meet the requirements of real-time applications. It employs a segregated list policy with three bitmaps based on a *best-fit* policy for small memory blocks as well as *good-fit* and *first-fit* policies for normal blocks, which can achieve a constant execution time. It can reduce the *wasted memory* and avoids exhaustive search in the lists with these policies. Also, establishing accurate node distances can make it access either the closest node or the closer nodes as often as possible.

3.2 An Overview of nMART

In order to achieve the objectives discussed above, *nMART* is comprised of three levels as shown in figure 3.2. Each layer is designed to accomplish the specific objectives we have discussed in Section 3.1. The figure shows the three layers stacked, with the upper two layers executed in the user-level space, and the bottom one in the kernel space.

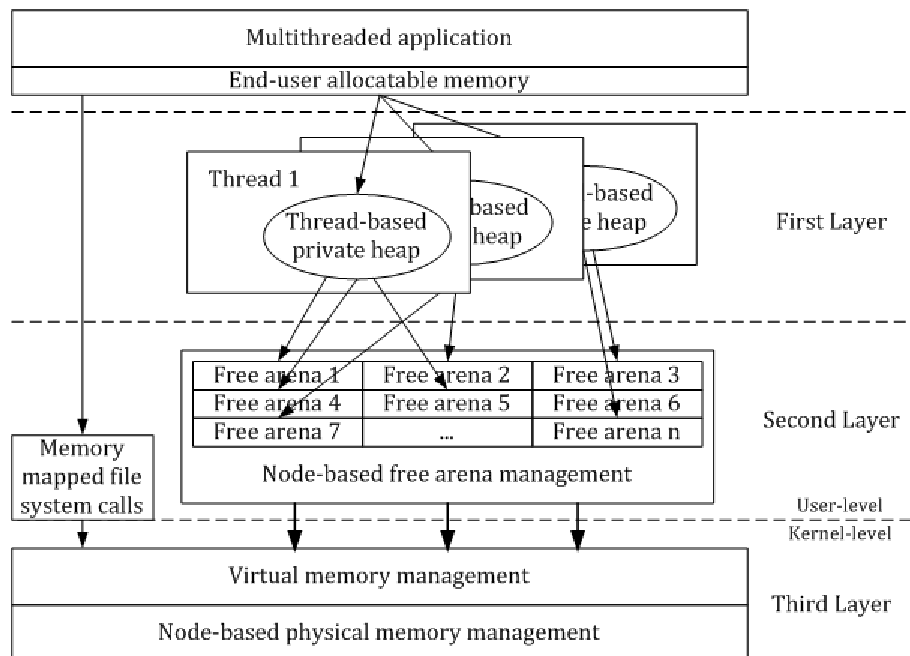

Figure 3.2: The structure of nMART

We have already mentioned that a key observation of this thesis is that current operating systems do not provide an accurate metric to measure distances between nodes in a ccNUMA architecture. Although nMART is primarily a user-level memory management algorithm, it is necessary to make requirements on the underlying operating system. Hence, the bottom layer considers this aspect. We discuss this layer in section 3.3. Following this we focus on the main user-level algorithms of nMART.

3.3 Kernel-level Node-based Memory Management

In this section we discuss the importance of discovering the underlying architecture and its impact on kernel-level memory management.

3.3.1 Discovery of the Underlying System Architecture

ccNUMA architecture systems have been designed to not be limited by a single shared system's resource such as the traditional system bus or centralized shared memory. These systems are comprised of multiple processors and a huge amount of memory. They are able to make application programming easier than other scalable multiprocessors because they support a global shared address space, and are globally cache-coherent. Moreover, the applications for SMP systems are able to be executed on such systems without any changes. However, such advantages do not come without cost. Due to distributed nature of the system, resource accesses are routed through the interconnection. The performance of memory operations depends on the memory location requested and its relationship with the requesting processor. Hence, the interconnect can become a new performance bottleneck; thus, recognizing and discovering the characteristics of the system resources as well as their hierarchy are two of the most important factors to optimize the system performance.

Rearrange system resources

Many well-known modern operating systems have supported ccNUMA architectures. These ccNUMA-aware operating systems have attempted to find an optimal node to allocate memory in order to reduce memory access latency and the bandwidth consumption of the interconnection. Usually, OSs have divided system resources, such as processors, and physical memory into a few logical abstractions based on nodes. This is because a set of processors comprising a ccNUMA system can consist of a node without their own local memory (called a 'hole'), and many hardware vendors also tend to partition the physical address space in order to simplify their hardware designs, and for specific purposes (such as video display buffers where, for instance, the physical address space on x86 architecture systems

from 0x000A'0000 through 0x000F'FFFF is reserved by the hardware). This means that the physical addresses exist on the system, but the associated page frames can never be assigned dynamically by the operating system [Daniel P. Bovet, 2005]. This would cause much memory to be wasted if the physical address space were to be represented as a linear array. This is the reason why operating systems re-arrange their system resources. For example, any processor in a node without its own memory will be re-assigned to another node which has its own memory; thus, it cannot guarantee that it will show equal access time to local memory for all processors in the associated node. This is an important factor that degrades the system performance; however, this thesis contends that most operating systems do not measure this *real* node distance.

Abstraction of system resources

Modern operating systems attempt to hide the complex hardware designs of the underlying architecture, and provide the appropriate abstract information to applications. A problem with the abstraction layers is that they do not provide enough information to develop predictable systems. Furthermore, processor makers have been separated from the mainboard makers in the market, so many mainboard makers cannot specify which type of processor is used in their boards for users' systems, except for a few huge vendors such as IBM, HP and SUN microsystems. Both the design of a processor and the design of a mainboard are the most important factors affecting the configuration of the topology of ccNUMA architectures; but the separation does not allow the appropriate information on how the underlying system has been designed to be determined.

Methods to recognise the system topology

In order to provide some interfaces to discover the underlying systems' architecture, Hewlett-Packard, Intel, Microsoft, Phoenix and Toshiba developed an open industry specification called ACPI (Advanced Configuration and Power Interface) 1.0 in 1996 (currently 5.0, since 2011), which establishes industry-common interfaces. The ACPI specification is to enable robust operating system-directed moth-

erboard device configuration and power management of both devices and entire systems [Hewlett-Packard Corporation et al., 2011].

The ACPI specification defines optional tables, which are a collection of interfaces that allow the platform to recognise associated processors and memory ranges with system localities and proximity domains. On ccNUMA platforms, the OS configure the information of the underlying systems using a *System Resource Affinity Table* (SRAT) during the OS initialization.

The processor's local APIC/SAPIC affinity structure in the SRAT provides the association between a processor and the proximity domain to which the processor belongs. And the memory affinity structure in the SRAT provides topology information such as the association between a range of memory and the proximity domain to which it belongs.

Going through such tables, OSs are able to configure their system to associate a processor and a range of memory to which it belongs using an unique integer value given to each proximity domain. After acquiring this information, the OS parses another table called the *System Locality Distance Information Table* (SLIT). The SLIT provides the relative distance information between all proximity domains. In particular, each value of element *(i, j)* in the table indicates the distance from a certain ccNUMA node *(i)* to every other node *(j)* in the underlying system so that the distance exists as much as $i * N + j$ is identical to equation 3.3, where N is the number of ccNUMA nodes. As each element value is normalized to be relative to 10, every element must have at least 10 or more because the distance values of 0 to 9 have no meaning as well as being reserved. Each diagonal element of the array has a value of 10, which means the relative distances from a node to itself. For many cases the SLIT is automatically set by the distance value of the local node as 10, and the value of remote nodes as 20. Table 3.1 shows the default SLIT information of our experimental machine describing figure 4.1, where the cost is 10 to local nodes or, otherwise, 20 to remote nodes.

This approach simplifies the underlying systems' architecture, and is able to provide the hardware abstraction, particularly, the type of processors, the number of processors and cores, and the number of nodes. Unfortunately, the approach can

degrade the systems' performance since it never specifies the cost of remote memory accesses accurately, even if they require a different number of hops, which indicates access distances on ccNUMA systems. Assuming that a thread accesses memory belonging to the same node as that on which it is running, it is a *0*-hop access or local access. Otherwise, it is represented as n-hops based on the number of access distances.

	node 0	node 1	node 2	node 3
node 0	10	20	20	20
node 1	20	10	20	20
node 2	20	20	10	20
node 3	20	20	20	10

Table 3.1: SLIT of the experimental machine

Our main experimental machine, as seen in figure 4.1, is comprised of four nodes. Node 0 has been connected to nodes 1 and 2, and node 3 has been connected to nodes 1 and 2. For economical reasons, node 0 and node 3 have not been connected to each other directly, which means that the two directions from node 0 to node 3 require going through another node: node 1 or node 2. As well as node 0 and node 3, the relationship between node 1 and node 2 requires going through other nodes. In these cases the values of two directions should be greater than the other directions which do not require going through another node, but table 3.1 just shows the value of 20 without considering the number of hops. This is done because these table are only able to provide limited information to discover the underlying architecture design due to the separation between processor and mainboard makers.

The use of application-level frameworks or file systems supported by the OS are alternative approaches to describing the system resources. The framework called *hwloc* [Broquedis et al., 2010] is able to obtain detailed knowledge of the system topology, including the number of processors, cores, shared caches, sockets, ccNUMA nodes, a range of memory, etc. The filesystem *sysfs* [Mochel, 2005] can also be another candidate to allow users to obtain the system topology. The virtual file system consists of a directory-based hierarchy, and its structure is based on the

kernel data structures. The files in the directories include the information exported by the kernel, so it is easy to access and is accurate. Another application-level library called *libnuma* [Kleen, 2005] allows ccNUMA memory policies to be added into user applications, and the *numactl* command allows users to control a program, where it can run on specific cores and memory nodes. Other current operating systems (e.g. Windows, Solaris and AIX) support similar APIs.

Unfortunately, these approaches are not efficient enough to recognise the real node distance. This is because such libraries and the filesystem rely on the information exported by the kernel. However, the Linux kernel obtains the system topology on boot-up by parsing ACPI tables. *hwloc* is only based on the *file system* on Linux, with the file system depending on the ACPI information. Since the information has been built from the ACPI tables, a modern operating system cannot fully optimize their support. Even if the speed is not the most important factor for the real-time system, the performance of their current implementation can be very slow. This is because these have required reading and parsing text files in the *sysfs* filesystem (*/sys/devices/...*).

3.3.2 Memory management and the ACPI Tables

It is instructive to discuss how the current Linux kernel builds and uses the node distance table in order to understand how Linux exploits the information from ACPI tables and how our approach will be approximated in Linux. In general, the Linux kernel divides physical memory into four memory regions, which are managed independently. These regions are called ZONE_DMA, ZONE_DMA32, ZONE_NORMAL, and ZONE_HIGHMEM. This is done to facilitate efficient physical memory management, as it can treat a "hole" in the address space of the physical memory in the system.

Some old devices should use under 16M bytes of physical memory for data transfer. On x64, in order to maintain these ranges, the kernel defines the physical address space up to 16M bytes as ZONE_DMA. Similarly, ZONE_DMA32 ranges from 16M bytes through 4G bytes in physical address space, and ZONE_NORMAL is for ranges of over 4G bytes. Using this approach, each individual node is able to

consist of four zones in its physical address space, but the ZONE_HIGHMEM [1] zone of x64 architectures is usually always empty, with the available linear address space always being much larger than the amount of physical memory. Figure 3.3 diagrammatically illustrates this memory structure.

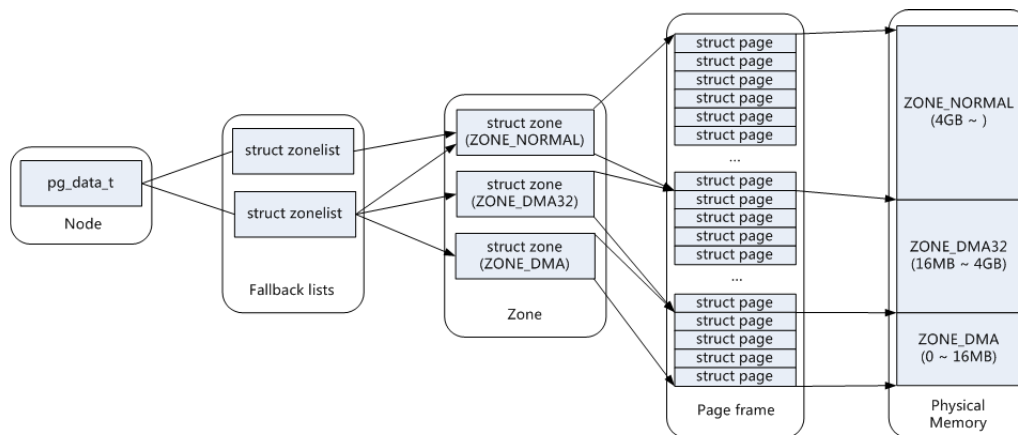

Figure 3.3: The relationship between nodes, zones and pages on x64 architecture system

As illustrated in figure 3.3, the Linux kernel separates the physical memory of a ccNUMA architecture into various address ranges of physical memory according to which each individual node belongs. For this, it uses a data structure called *pgdat_list*. There exists one *pgdat_list* for each node in the system. This data structure stores the information associated with each individual node. Each individual *pgdat_list* points at *node_zonelist* arrays that store entries available for every possible zone type which can be chosen for allocation later. The entry specifies the zones to visit when a given zone cannot satisfy the allocation request because it does not have sufficient free memory.

In this situation, the kernel attempts to allocate memory from the next node in the entry list, which has been established based on the node distance table. Unfortunately, since the table has not considered the real node distance, the process of finding the next node in the entry list can cause a performance degradation of memory allocation operations.

[1] On x86, this zone contains high memory, which is not permanently mapped into the kernel address space. This zone is all memory above the physical 896M bytes.

Furthermore, the Linux kernel attempts to balance presented zones in the system when an application requests allocating memory, meaning that the kernel tries to avoid making a hot-spot by zone-balancing. Therefore, the kernel theoretically can simultaneously take page frames from different memory regions. Although the approach is a good solution to balance use of the zones, it may lead to accessing more remote node memory. For instance, after the running application on node 1 consumes all of the local memory, the next entry node is node 2, from which page frames come. However, if the relationship between node 1 and node 2 shows a worst-case node distance, this significantly degrades the system performance. This is illustrated in Appendix B.2. Also, we discuss how the Linux kernel initializes the zone list and the node distance table at boot time in Appendix B.

3.3.3 A More Accurate Model of Node Distance

As discussed above, when an operating system boots up, it invokes some functions to obtain the information of all node distances. Firstly, it builds a node distance table by collecting information from the ACPI tables in the system BIOS. After that, the OS can choose to optimize its behaviour based on such tables, which means that it can decide to execute some threads on a specific processor in a node, and allocate memory for such threads from the memory inside the node. In practice, many operating systems use demand paging where pages should only be brought into physical memory when they are accessed. In demand paging, a ccNUMA-aware operating systems should consider the node distance, processor affinity, and memory affinity when mapping virtual memory into physical memory. Unfortunately, there is no accurate model to use to evaluate node distances.

We have seen in section 3.3.1 that node distance is a key factor that affects where an operating system places allocated memory, and that the metric used is rather simple. Here we analyze in more detail the factors that affect the cost of remote memory access in order to derive a new metric for the distance between the nodes.

The time taken to access a remote node's memory is affected by the following factors:

The number of hops : The number of hops between the requesting node and the

target node is one of the important aspects affecting the system performance and is one that has been discussed in Section 3.3.1.

The traffic on the interconnection : The traffic on the interconnection is also a crucial factor affecting the performance of the system. It depends on the number of nodes, as well as the number of processors or cores belonging to a node. Moreover, the higher the system's load, the greater the amount of traffic required to maintain the cache coherence between the nodes. For example, in [Hewlett-Packard Corporation, 2012], in an 8-socket system, the snoop protocol consumed 50 to 65% of the interconnection bandwidth based on an internal experiment.

The speed of the interconnection : Many hardware vendors have attempted to reduce the NUMA ratio by increasing the interconnection speed. This is because if the system uses a faster link, it can process much more data. This has the effect of reducing the amount of traffic on the link indirectly.

The congestion on the interconnection : The congestion on the link depends on the system load, the number of nodes, and the link speed. A high system load, many nodes, and a slow link will increase the potential for congestion.

A more accurate model of the current distance between two nodes can be described by graph theory. The notation $P_k(V)$ stands for the set of all k-element subsets of the set V. A node distance D is a pair $D = (V, E)$ where :

- V is a finite set called vertices of D
- E is a subset of $P_2(V)$ called the edges of D

In our example of Figure 4.1, a node distance from node 0 (N0) to node 3 (N3) is represented clearly by $D_{N0,N3}$. The set of V is represented as $V_{N0,N3} = \{N0, N1, N2, N3\}$, which are incident nodes between the start node (N0) and the destination node (N3). The set $E_{N0,N3}$ is also represented, which are paths between the start node and the destination node, as follows:

$$E_{N0,N3} = \Big\{\{(N0, N1), (N1, N3)\}|\{(N0, N2), (N2, N3)\}\Big\}$$

As E is a subset of $P_2(V)$, each element of the set of E can be given a subset recursively. For instance, the first element $\{(N0, N1)\}$ can be given as $E_{N0,N1}$. Given the above factors, an adjacent node distance, e.g. $E_{N0,N1}$, can be given by the following equation:

$$adjacent\ node\ distance\ (D_{i,j}) = \frac{T_{i,j} + C_{i,j}}{S_{i,j}}$$

In this equation a node distance is represented by D, i represents the start node, and j means the destination node. They should be adjacent nodes. T represents the traffic on the interconnection, S means the speed of the interconnection, and C represents the congestion on the interconnection. Therefore, the node distance $D_{N0,N3}$ can be calculated as follows:

$$D_{N0,N3} = \Big\{\{D_{N0,N1} + D_{N1,N3}\}|\{D_{N0,N2} + D_{N2,N3}\}\Big\}$$

This means that the more the speed of the interconnection increases, the more the value of the node distance decreases; and the more the amount of traffic on the link or congestion increases, the more the value of the node distance increases. In this case, the traffic or congestion is able to be changed dynamically based on the system circumstance. Moreover, it is impossible to measure or change these values for an application without any support by some APIs or instructions from the underlying OS and processors. The interconnection speed is the only changeable value that we can modify statically through the BIOS setting.

Currently, most operating systems that consider themselves to be NUMA-aware do not consider all the significant factors mentioned above. Indeed, dynamically monitoring the state of the interconnections would impose an intolerable overhead. However, we will show that the current simplistic measure is also not satisfactory and that a compromise needs to be found.

In order to apply our model to the kernel directly, we can choose one approach among three options as follows. First, is to measures the cost, which is the elapsed time to complete some operations (read and write) instantly for each time, and builds

a node distance table inside the kernel to be accessed each time it is needed. It has a big advantage of instantly and dynamically applying the system circumstance to choose the optimal node to allocate; however, complexity arises, and it suffers from a big overhead.

The second approach is at system boot time, the kernel computes the node distance information and the cost using the same method of the first approach, and builds the table dynamically. It has less of an overhead than the first approach, but it is difficult to implement because building the information arises in the initial step of the kernel boot, so we cannot use the essential functions to compute the distance, such as *kmalloc()*. In addition, if we re-order the *zonelist* after boot, we should use the global lock, but we cannot guarantee the lack of any side effects.

The last approach is before the kernel compiles; it builds the table using information obtained using the same method of the first approach, and applies the table at the kernel compile time statically. A disadvantage of this approach is that cannot apply the system circumstance instantly to system runtime, but it is easy to implement and apply, and is able to provide system performance improvement as well. Eventually, we have chosen a third approach to reduce the system overhead and maximize the impact of our model. In our approach the information will be sorted in an order based on the table after invoking the *build_zonelists()* function.

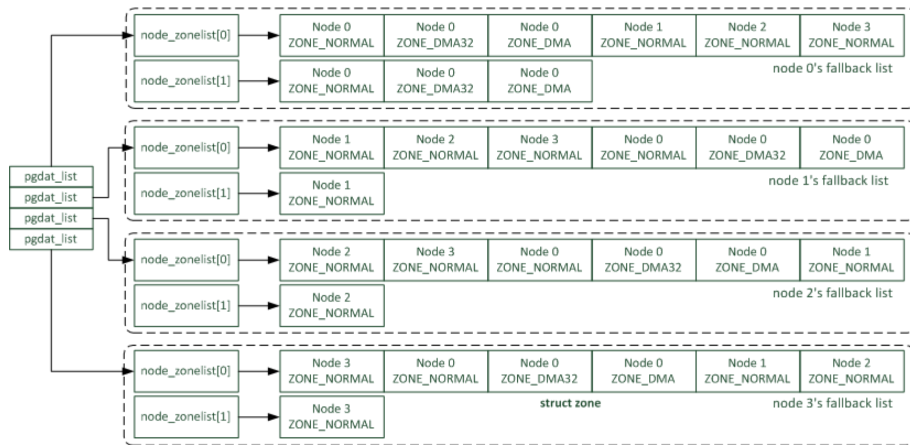

Figure 3.4: The default zones lists for our experimental hardware

As a result, this algorithm, which avoids choosing the worst case node for memory allocation, sorts the memory regions under the model. The default ordering in Figure

3.4 is changed to the new ordering in Figure 3.5.

For instance, if an application running on node 2 requests allocating memory, after it consumes all of the local memory with the original node order, it consumes physical memory from node 3. While under the new model, it consumes node 0's memory instead of node 3's memory by default.

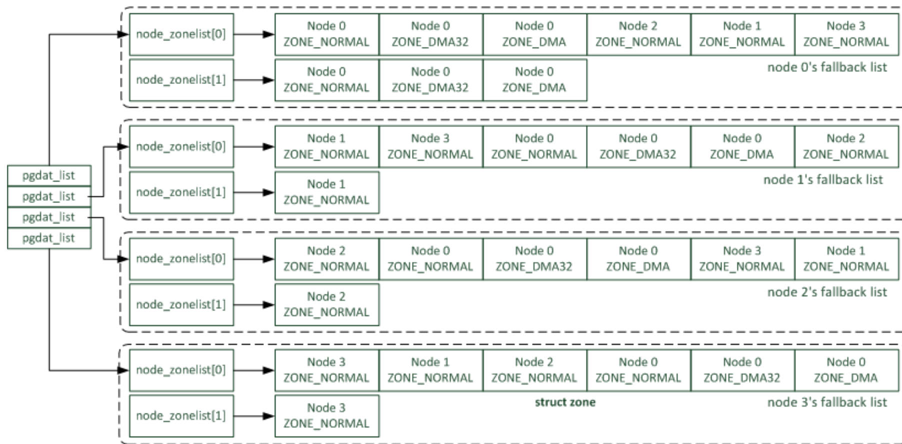

Figure 3.5: The sorted zone lists for our experimental hardware

3.3.4 Required Kernel Modification

Inevitably the modification required to the underlying operating system will be kernel and architecture specific. We summarise how this can be achieved for the Linux kernel and our example architecture and give further information in Appendix D.

Generating the node distance table

We will consider how our model can be approximated in the implementation of Linux. In order to implement the model, we proceed in three steps. Firstly, we have created the new node distance table based on measured node access times, as discussed in Appendix B.2, and then created a header file which stores the node distance. This will be used at compile time of the kernel. Next, we patch the Linux kernel to use the new node distance table, and re-compile the kernel source.

Lastly, we evaluate our modified kernel based on our model using an application in Appendix B.3.

The previous application in Appendix B.2 to measure the real node distance in the system independently generates a sequence of 100 results measured per node, which are formatted with the following: current time, processor ID, and the time spent by the system executing the task.

In a header generating application, it has attempted to read the output files, compute the average execution time, normalize the average execution time to be relative to the average execution time of the local access, and then save the normalized values for all nodes to a header file to be used at compile time of the kernel. Details of the implementation of the application have been provided in Appendix D.1.2

The output header file named *numa_distance_table.h* is formatted by a single two-dimensional array as the following header file represents the real node distance of our main experimental machine described in figure 4.1.

Source 1 numa_distance_table.h

```
1  #ifndef __X86_MM_NUMA_DISTANCE_TABLE_H
2  #define __X86_MM_NUMA_DISTANCE_TABLE_H
3
4  #define __NUMA_DISTANCE_TABLE_ROWS__    4
5  #define __NUMA_DISTANCE_TABLE_COLS__    4
6
7  u16 __numa_distance_table[4][4]={
8          {10,13,12,14},
9          {13,10,14,12},
10         {12,14,10,13},
11         {14,12,13,10}
12 };
13 #endif //__X86_MM_NUMA_DISTANCE_TABLE_H
```

Modifying the kernel source

As is discussed in Appendix B, the *numa_distance* array is the most important factor to represent the real node distance, which means that we are able to achieve the objective of the third layer, wherein the page frames are placed from the closest node or the next closest node, by modifying the array simply.

The *acpi_numa_slit_init()* function attempts to build the table on system bootup, but the actual work is delegated to the *numa_set_distance()* function. We

have modified the kernel to replace the original values from the ACPI tables to the generated values in the output header file. Moreover, at lines 459-462, the *__node_distance()* function originally accesses the *node_distance* array, but we have removed its original codes completely and added line 461 to access the generated table directly instead of the original table. Details of the implementation of the *numa_set_distance()* function have been provided in the below source code.

Source 2 arch/x86/mm/numa.c

```
22   #include "numa_distance_table.h"

37   EXPORT_SYMBOL(numa_distance_cnt);
38   EXPORT_SYMBOL(numa_distance);

437  void __init numa_set_distance(int from, int to, int distance)
438  {
439    ...

455    numa_distance[from * numa_distance_cnt + to] =
456      __numa_distance_table[from * numa_distance_cnt + to];
457  }
458
459  int __node_distance(int from, int to)
460  {
461    return __numa_distance_table[from][to];
462  }
```

At lines 37-38, in order to verify the re-ordering fallback lists in terms of a kernel module, we have exported kernel symbols to be accessed by the module dynamically. At line 455, element values in the *numa_distance* array have been set by the generated values in the *__numa_distance_table* array. In the process the original value of SLIT, the function parameter *distance*, is unused.

After all modification of the kernel, we have re-compiled the kernel. Before compiling, the output header file *numa_distance_table.h* is placed in the directory *arch/x86/mm*.

3.4 User-Level Memory Management Algorithms and their Implementation

nMART combines the segregated fit with bitmap allocation strategies. It uses node locality to accelerate its memory operations. In searching for free blocks in the

list, it employs the *good-fit* and *first-fit* policies to avoid performing an exhaustive search. Also, these policies are able to achieve a constant execution time for memory operations (allocation/deallocation). In this section, we will describe the algorithms and their implementation in detail. Figure 3.2 illustrated the basic structure of the user-level algorithms.

3.4.1 Overview of Levels 1 and 2

nMART employs the *exact-fit* mechanism to improve efficiency of small block allocations and to reduce internal fragmentation. It also uses the segregated-fit mechanisms to implement a *good-fit* and a *first-fit* policy to find the closest segregated size class, thereby avoiding the need to perform an exhaustive search. The implementation uses two types of bitmaps to maintain free blocks. Also, the segregated list with bitmap policies makes the algorithm predictable as they can be implemented in a bounded execution time.

One of the bitmaps is used for the maintenance of small blocks, and is implemented as a two-dimensional array to hold free blocks corresponding with their sizes. For efficient management, the block size is spaced 4 bytes apart to 512 bytes. To indicate whether a corresponding size of a block is free, 64-bits in two bitmaps and a pointer array holding free blocks have been employed, as shown in figure 3.6.

The second type of bitmap consists of a two-dimensional bitmap array pointing to free blocks. The first-level bitmap, indexed by i, indicates free blocks of sizes in ranges from 2^i to $2^{i+1} - 1$, and the second-level bitmap, indexed by j, divides the size range of each first-level in a number of ranges of an equal width linearly. For convenience the number of ranges in the second level is represented as the exponent of two: 2^r (r is 6 by default). The parameter r divides the first-level ranges in a number of ranges linearly. For instance, if r is 6, there are 64 segregated lists within the given size ranges indexed by i; and if r is 1, the algorithm manages free blocks as efficiently as the binary buddy algorithm.

Defining r is important to specify the behaviour of the allocator. This is because there is a trade-off for deciding the minimum block size. If r is larger, it leads to consuming more memory space for the information of bookkeeping such as additional

bits, and pointers. However, a too small a value for size incurs increasing the amount of internal fragmentation significantly.

Consequently, index i refers to the available maximum size of a block: $2^{i+1} - 1$, and the number of ranges (2^r) sets the total number of segregated lists within the given sizes. Moreover, a particular segregated list can be indexed by index *I(i, j)*, and the value of index *I* indicates whether the list *(i, j)* contains any free blocks or not. Thus, all bitmaps do not contain free blocks, but they indicate the potential existence of a certain size of block. All pointers to free blocks are stored in a two-dimensional pointer array called *MATRIX*.

As we set the value of r to 6 by default, each element of the array refers to a list containing free blocks of sizes in a range from $2^i + 2^{(i-6)} \times j$ to $2^i + 2^{(i-6)} \times (j+1) - 1$.

In terms of implementation, the algorithm uses two-dimensional bitmap arrays, which requires a 64-bit variable for the first-level bitmap and 64 x 64 bit variables for the second-level bitmaps, so it requires a total of 66 variables of 64-bit to indicate the free block lists.

First Layer

The first layer is designed to achieve a constant execution time for both allocation and deallocation memory operations. Each segregated lists holds certain size of free memory blocks, and the algorithm can take a freed block using an index calculated by equation 3.2. The layer is implemented using bitmaps and single linked lists for small sizes of blocks, and also it is implemented using a bitmap, arrays of pointers to free blocks and doubly linked lists for ordinary sizes of blocks. Sharing a single global heap among multiple threads tends to increase the probability of lock contentions. In order to reduce this, each thread of the application creates a per-thread heap. The first layer describes the private heap per thread. The private heap is only used by the owner thread at allocation time so that the owner thread is able to access it without any lock contentions. While at de-allocation time, all threads can request releasing memory blocks in use from the other threads' heaps.

When a thread is created by calling the *pthread_create()* function in the POSIX library, an amount of memory space will be assigned as an initial private heap to

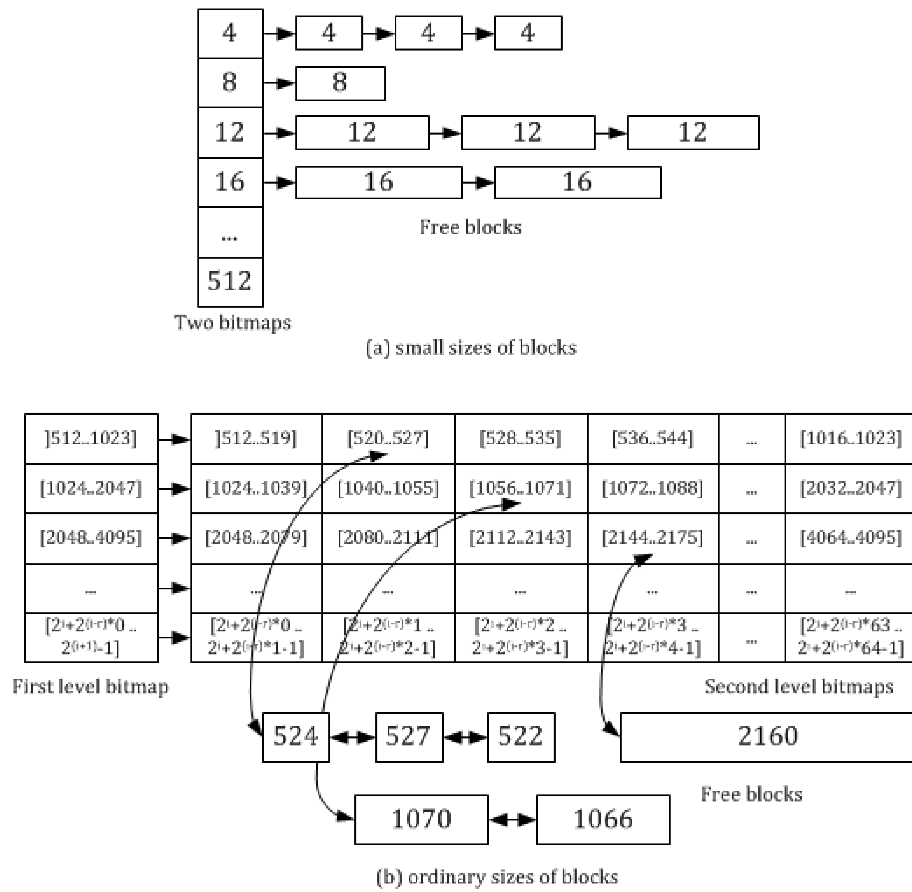

Figure 3.6: The thread-based private heap on the first layer

the creating thread, with a part of that space being reserved for the purpose of book keeping. In order to implement this, we have overloaded the pthread_create() function using a direct mechanism for loading C libraries at runtime called *dl* functions, such as *dlopen()*, *dlsym()*, *dlerror()* and *dlclose()*. In the overloaded function, a thread-specific data key, visible to all threads in the process, is created. This key will be used for mapping per-thread data structures, named *thread control block* (TCB; different from the operating systems' one) and defined as *tcb_t*, which stores the two-level bitmaps with pointers to free block lists, a pointer to an *arena* (a memory pool) and the owner thread ID.

Second Layer

The second layer aims to achieve real-time predictable performance. It allows the allocator to get free *arenas* from the layer instead of making system calls such as *mmap()*. It provides free arenas to the first layer based on node locality. Free arenas can be taken by index using the same approach of the first layer (segregated lists with bitmaps). The layer is implemented using a bitmap, an array of pointers and single linked lists.

In contrast to the first layer, there are multiple separated free arena lists for each ccNUMA node, as illustrated in Figure 3.7. *nMART* employs as many free lists as the number of ccNUMA nodes in the system. At start-up time of the application, this layer is completely empty, and then will be filled gradually. When the private heap in the first layer has insufficient space to satisfy an application request, the thread requests some arenas to be added to its per-thread heap so that the heap can be comprised of multiple *arenas*. If an arena is completely empty (no memory block is in use in the arena), it will be moved into the second layer from the first layer. Extending the private heap can be processed efficiently because moving an arena is achieved by pointer operations.

Each free arena list is associated with an individual ccNUMA node and is able to be shared by a group of threads running on that node.

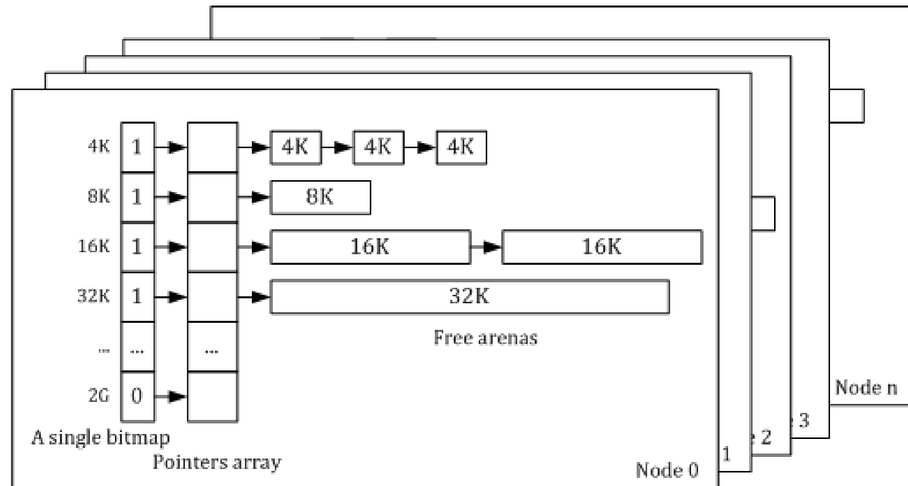

Figure 3.7: The node-based free arena management on the second layer

Third Layer

Going through the first and second layer, *nMART* is able to reduce lock contentions and consider node locality. In order to maximize node locality, the third layer considers the actual node distance (as described in Section 3.3).

3.4.2 Configuration parameters

The behaviour and structure of the *nMART* algorithm depend on some user-configurable parameters, as follows. These parameters can be modified in the source code of *nMART*.

Minimum block size The minimum block size can lead to significant internal fragmentation when applications request many memory blocks that are smaller than the minimum size. Assume that the minimum block size is 32 bytes and an application requests a 16 byte memory block. Then this can result in as much as 16 wasted bytes per memory allocation.

On the other hand, too small a minimum block size will consume more space for the headers. For instance, if the header size is 32 bytes, and the size of block data is 16 bytes, the bookkeeping information is then bigger than real data of a block. For this reason, we have defined the minimum block size as

4 bytes, with it being spaced 4 bytes apart to 512 bytes by the small block management policy.

Maximum block size Most modern processors belonging to ccNUMA architecture systems have 64 bits addressing. For efficiency reason, the size of bitmap used indicating free block existence is one word. Meaning that the size of the bitmap is 64 bits because one word size on the underlying machines is 64 bits. In practice, the C language header file <bits/wordsize.h> in Linux with *glibc* defines one word size (__WORDSIZE) 64.

Moreover, the first level, index i, defines the number of rows in the MATRIX array so that the allocator is theoretically able to provide a free block of 2^{64} bytes. However, applications are very unlikely to request such large blocks in practice. For this reason, we have limited the manageable maximum size of free blocks to $2^{31} - 1$ bytes so that the value of index i can be up to 31.

3.4.3 Data structures

The data structures used by *nMART* are described in this section. Typically, the headers contain the essential information needed to manage the block. The headers can be mainly categorised into three distinctive types, as follows: one is used for the private heap, called *arena*, and others are used for normal memory blocks. Note that all free blocks, satisfying an allocation request, are served in a *LIFO* (last in, first out) policy.

nMART control block

Some of the information regarding the underlying system is essential in order to execute the memory allocator. As seen in figure 3.8, the *nMART* control block contains the following information: the total heap size allocated by *nMART*, the number of processors, the number of ccNUMA nodes, the processor masks, and the pointer of the *arena* lists.

The processor masks store the information of the node IDs to which the processors belong. The *ArenaList* is shown in figure 3.9. This data structure contains

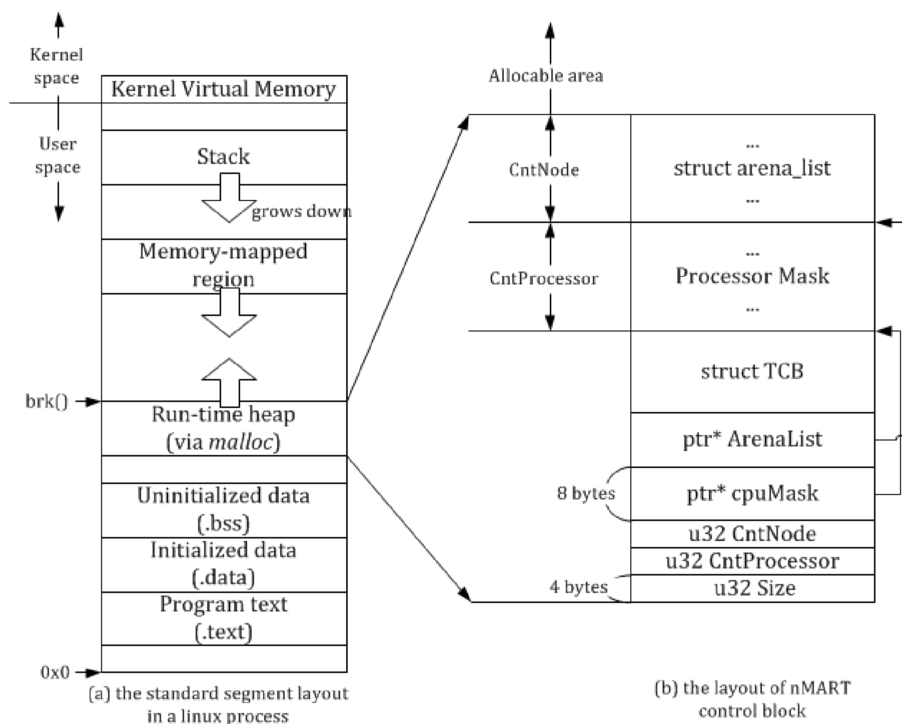

Figure 3.8: The structure of an nMART control block

the head pointer of the arena list and a counter variable. There is an arena list for each ccNUMA node in the underlying system, and the data structure resides in the second layer of the allocator. As we mentioned above, the second-level layer in the allocator manages the number of free arena lists. These free arena lists are separately maintained by each node in order to reduce lock contentions.

Figure 3.9: The header of an arena list

The thread control block

A thread control block (TCB) is assigned to each thread automatically when the thread is created. As seen in figure 3.10, the following information is stored in the TCB: the first-level bitmap, the second-level bitmap array, a two-dimensional pointer array called *MATRIX*, a pointer to the arenas, a pointer to the free blocks list, and thread ID.

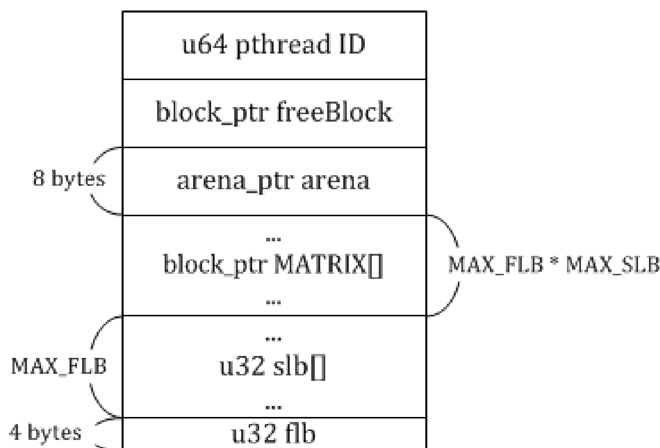

Figure 3.10: The structure of a thread control block

Of particular note is the *freeBlock* pointer, with its role being as follows. Consider an application in which a producer thread requests allocating a memory block and passes its reference to a consumer thread, which requests freeing it. In this case, it should lock the heap area to avoid data corruption. In order to release a memory block efficiently, we have used a pointer field (*freeBlock*).

The pointer of the free block is used for a particular condition, wherein a non-owner thread, like the consumer thread in the above example, frees an unowned memory block. We have called this phenomenon *remote free* or *remote release.* When it happens, the non-owner thread sets the free bit as 1 in the header of the block, adding it to the field. The field points to a single linked lists, consisting of free blocks. All of the free blocks will be released when the owner thread next requests allocating or releasing a memory block. For this reason, it can prevent unbounded heap growing in the procedure-consumer model.

This approach may increase memory consumption because this policy does not release the memory block immediately. However, this does not often happen in practice. In [Larson and Krishnan, 1998], Larson has observed that remote releases (which they called *bleeding*) occurs in the 2% to 3% range of deallocations on large and long-running applications.

The arena header

Each arena has a header that contains the following information: a pointer to the last block in the arena, the ccNUMA node ID, the arena size, and two pointers

to the previous and next arena. The ccNUMA node ID indicates where the arena was created. It is used by the second layer when the arena is released.

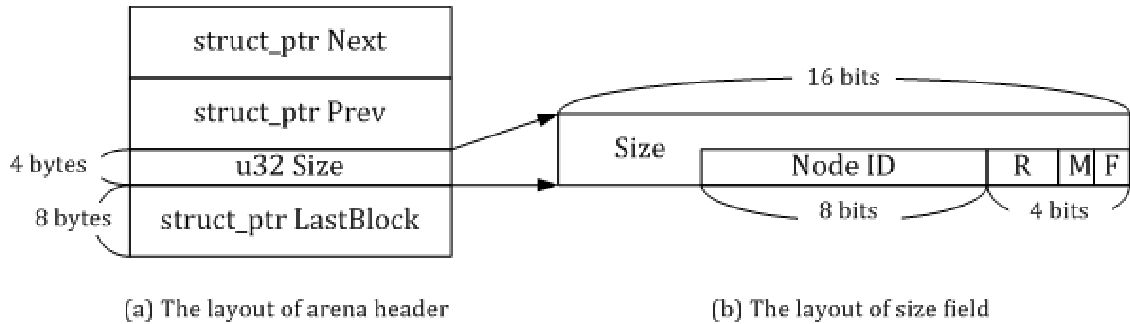

Figure 3.11: The header of an arena

The size of an arena should be a multiple of page sizes of the underlying OS, and the minimum size of the arena depends on the underlying OS. For instance, it is 4K bytes on Linux by default, meaning that we do not need to use all the bits in the size field. For space optimization, we have used some of the spare bits for representing node ID and some states, as seen in figure 3.11.

The block headers

The headers of memory blocks can be separated into two types corresponding with the block size. For efficiency reasons, blocks in use are not linked in any segregated list, and are not managed by the allocator because splitting and coalescing always occur on freeing blocks. As with Knuth's boundary tags technique, the header size of the freed blocks is bigger than the header size of blocks in use.

As shown in figure 3.12, the header of free blocks includes the following information: a pointer to a previous block header physically, a pointer to the owner, the block size, the value of the first and second level, and the boundary tag, which contains pointers to the header of the previous and next block logically. If the block is in use, it does not contain the boundary tag.

The size field for normal blocks is large enough to contain the allocable block sizes because the sizes of manageable blocks range from 512 bytes to $2^{31} - 1$ bytes, with the normal block sizes always being a multiple of four. Consequently, a few unused bits (at least 11 bits) can be used for other purposes, such as indicating some

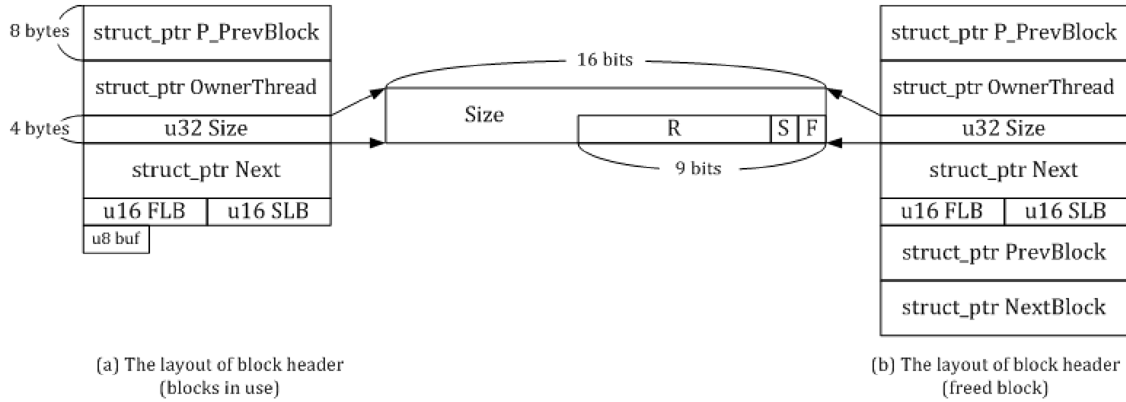

Figure 3.12: The header of normal blocks

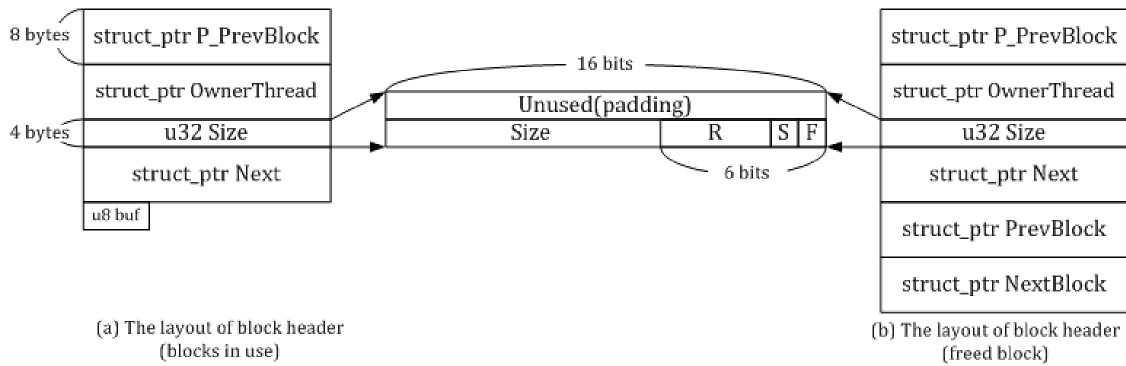

Figure 3.13: A block of small blocks

states; so we have used the three least significant bits in the size fields to indicate whether a block is free or not, whether the block is a normal size or not, and whether the block is managed by the allocator or not.

3.4.4 Function details

The best approach to analysing memory allocation algorithms is to examine the source codes or pseudo codes in detail; so we will discuss some significant function codes of the *nMART* algorithm in this section. We classify them into four types: essential management functions, bitmap functions, arena management functions, and block management functions.

The essential management functions

The essential management functions are the functions used by the allocator when the application requests memory allocation and de-allocation.

- **malloc()**

The *malloc()* function is responsible for allocating memory on the heap. The application accesses the memory block via a pointer that the function returns. Typically, the function takes a parameter with the size of the application requests, and returns a pointer as the result if the allocator can satisfy the request. Otherwise, the allocator returns a null pointer.

Source 3 malloc()

```
1  void* malloc(size_t _rSize)
2  {
3    pthread_once(&InitOnce, InitnMART);
4
5    if ( _tcb == NULL )
6      InitTCB(_tcb);
7
8    AllocateBlock(_rSize);
9
10   return (void*)retBlock;
11 }
```

The *malloc* function is relatively simple. It just invokes two initializing functions when the application requests allocating a memory block for the first time, and calls

the real allocation function with the requested size. The actual work is performed in the *AllocateBlock()* function. This function is the most fundamental function of the allocation.

At the first time of allocation, the data structures used by the allocator need to be initialized with some values corresponding to the underlying system. In *InitnMART()*, the function initializes the *nMART* control block discussed in 3.4.3, setting values to variables such as the number of processors and the number of ccNUMA nodes, and then increases the heap. The *pthread_once()* ensures that an initialization code (defined in *InitnMART()*) is executed at most once.

InitTCB() initializes the header of the TCB used by each thread. Sometimes, the application can be executed as a single process without threads. For this case, the allocator creates an instance of the TCB data structure, setting values to the data structure.

- **free()**

The *free()* is invoked with a pointer to the space to be released when the memory block is no longer needed. After that, the released memory block will be inserted into the corresponding segregated list.

Source 4 free()

```
1  void _free(void *_ptr)
2  {
3    CheckRemoteFreeBlock();
4
5    if ( GET_BLOCK_OWNER(_block) == _tcb )
6      LocalFree(_tcb, _block);
7    else
8      RemoteFree(GET_BLOCK_OWNER(_block), _block);
9  }
```

Like *malloc()*, *free()* does not perform real functionality. The actual work is delegated to *LocalFree()* or *RemoteFree()*. At line 3 the *CheckRemoteFreeBlock()* function checks whether memory blocks exist that have been released by non-owner threads as remote-free operations. If some exist, the function releases all blocks immediately.

The requesting thread then invokes either *LocalFree()* or *RemoteFree()*. If the thread requests releasing a local memory block, it invokes *LocalFree()* or, otherwise, *RemoteFree()*.

- **AllocateBlock()**

AllocateBlock() is responsible for providing a memory block exceeding a threshold (4 bytes). The given size requested is rounded up to the closest size class at line 3. At line 5, it checks whether memory blocks have been released by remote free. If a suitable block exists, the allocator will serve it to satisfy the request. However, at lines 6-10, it searches for a free block in the corresponding segregated list.

Source 5 AllocateBlock()

```
1  void* AllocateBlock(size_t _rSize)
2  {
3    _rSize = ROUND_UP(_rSize);
4
5    block_t* retBlock = CheckRemoteFreeBlock(rSize);
6    if ( !retBlock )
7    {
8      GetIndex4Search(&rSize, &fl, &sl);
9      retBlock = FindFreeBlock(*_tcb, &fl, &sl);
10   }
11
12   if ( !retBlock )
13   {
14     SET_USED_BLOCK(retBlock);
15     ExtractBlockFromList(*_tcb, retBlock, fl, sl);
16     if ( GET_SIZE(retBlock) > SPLIT_THRESHOLD )
17       SplitBlock(retBlock, _rSize);
18
19     return (void*)(GET_BLOCK_BUF(retBlock));
20   }
21   else
22     return NULL;
23 }
```

When an acceptable block is found that is large enough to satisfy the request, it will be extracted from the segregated list and split at lines 12-20 if necessary. Otherwise, the allocator returns NULL as an error, indicating that the system does not have enough memory space.

- **CheckRemoteFreeBlock()**

CheckRemoteFreeBlock() is a special function with the responsibility of releasing unused blocks that are freed by *remote free*. It just checks the existence of unused blocks linked to the *freeBlock* in the TCB. It is invoked when the owner thread attempts to allocate or release a memory block.

Source 6 CheckRemoteFreeBlock()

```
1  void CheckRemoteFreeBlock()
2  {
3    AcquireLock();
4    block_t* block = _tcb->freeBlock;
5
6    while ( NULL != block ) /*pop blocks from a stack*/
7    {
8      block_t* next = GET_NEXT_BLOCK(block);
9      LocalFree(_tcb, block);
10     _tcb->freeBlock = next;
11     GET_PREV_BLOCK(next) = NULL;
12
13     block = next;
14   }
15   ReleaseLock();
16 }
```

In the function, it pops unused blocks from a stack and releases the blocks iteratively. This could make the algorithm unpredictable but it is only executed in an extremely rare condition. It depends on the design of the multi-threaded applications. If the application does not cause a *remote free* operation or the program is just a single-threaded application, the function is never executed. Also, the *remote free* is observed in 2% to 3% range on large and long-running applications [Larson and Krishnan, 1998]. It is for this reason that nMART cannot be used in hard real-time systems.

- **LocalFree()**

LocalFree() is a core function with the responsibility of releasing a memory block that is no longer needed. It is invoked when the owner thread attempts to allocate or release a memory block.

Source 7 LocalFree()

```
void LocalFree(tcb_t* _tcb, block_t* _block)
{
  SetFreeBlockStatus(_block);
  ResetLinkInfo(_block);

  MergeFreeBlock(&_block);
  if ( !FreeArena(_tcb, _block) )
  {
    SetBlockIndex(_block);
    InsertFreeBlock(_block);
  }
}
```

In the function, it sets the free bit to 1 to indicate its free state, and resets all link information (lines 3-4). After that, at line 6, it checks whether adjacent blocks have been already freed. If so, it merges to create a larger block.

If the block is the last one in use of the containing arena, the arena will be completely freed by *FreeArena()*. The arena will be inserted into the LIFO data structure of the TCB if it is free, or it will be passed to the second layer if the total size of the arena linked to the TCB exceeds the threshold. Otherwise, at lines 9-10, the block will be inserted into the corresponding list.

- **RemoteFree()**

The remote freeing of a block is accomplished by *RemoteFree()*. The function simply invokes *AddFreeBlockToRemote()*. The actual work is delegated to the *PushStack()* function. *PushStack()* inserts a memory block into the LIFO stack.

Source 8 RemoteFree()

```
void RemoteFree(tcb_t* _rtcb, block_t* _block)
{
  AddFreeBlockToRemote(_rtcb, _block);
}

void AddFreeBlockToRemote(tcb_t* _rtcb, block_t* _block)
{
  PushStack(_rtcb, _block);
}

void PushStack(tcb_t* _tcb, block_t* _block)
{
  AcquireLock();
```

```
14
15    GET_NEXT_BLOCK(_block) = _tcb->freeBlock;
16    GET_PREV_BLOCK(_block) = NULL;
17
18    __sync_bool_compare_and_swap(
19        &_tcb->freeBlock, GET_NEXT_BLOCK(_block), _block) )
20    ReleaseLock();
21  }
```

PushStack() uses *__sync_bool_compare_and_swap()* (one of the built-in functions for atomic memory access provided by *gcc*) and compares the value of the second parameter with the value of the variable to which the first parameter points. If they are equal, the value of the third parameter will be stored in the address referred by the first parameter. Otherwise, no action is performed. This function performs an atomic compare and swap, with a full memory barrier being created when it is invoked.

The bitmap functions

The allocator uses bitmaps to accelerate the search for a free block within a segregated size class. All bits of the first-level bitmap have an associated second-level bitmap indicating whether the state of a particular element in the *MATRIX* is empty or not.

In practice, the functions can be implemented using many different approaches, such as using bit operators in languages or some instructions supported by the underlying processor. As these functions are just utilities, which do not affect the core algorithm for the allocation/deallocation, we simply derive them from built-in functions provided by *gcc* or the source code of the Linux kernel.

- **ls_bit() and ms_bit()**

ls_bit() searches for the least significant set bit (LSB) in a variable. This function can be implemented using *bsf* instruction supported by modern processors, or *gcc*'s built-in functions - *__builtin_ffs()*, *__builtin_ffsl()* or *__builtin_ffsll()* - are also able to find the least significant set bit. Using the function, it is possible to find a smaller list containing blocks equal to or larger than the requested size.

ms_bit() searches for the most significant set bit (MSB) in a variable. This, therefore, computes the $\lfloor \log_2(r) \rfloor$ function. Like ls_bit(), it can be implemented using machine instructions such as *bsr* or *bsrl*. However, *gcc* does not provide built-in functions for the MSB searching.

- **set_bit(), clear_bit() and isset_bit()**

These functions manipulate a bit in a bitmap. *set_bit()* sets a bit to 1 in the indicated bitmap, *clear_bit()* sets a bit to 0, and *isset_bit()* returns the value of a given bit. *set_bit64()*, *clear_bit64()*, and *isset_bit64()* functions are extended versions of these functions to support the manipulation of 64-bit bitmaps.

- **GetIndex4Insert()**

An inserting function that implements the following equation to compute the index I (i, j) in the *MATRIX*. The index I indicates a segregated list containing free blocks of sizes in a certain range. s denotes the request size of a free block, and m is the constant 3, which is used as a bit mask.

$$
\begin{aligned}
I &= \frac{(s+m) \wedge (\neg m)}{4} \qquad \text{if } 1 \leq \text{s} \leq 512 \\
I(i,j) &= \begin{cases} i = \lfloor \log_2 s \rfloor \\ j = \lfloor \dfrac{(s-2^i)}{2^{i-6}} \rfloor \end{cases} \qquad otherwise
\end{aligned}
\tag{3.1}
$$

GetIndex4Insert() is for normal size blocks, with the function being explicitly invoked when a memory block needs to be inserted into a certain segregated list. A block should be inserted into the list at a position indicated by *I*. s

Source 9 GetIndex4Insert()

```
1  void GetIndex4Insert(size_t _s, int *_i, int* _j) {
2    if ( _s > MAX_SIZE_IN_SMALL_BLOCK ) {
3      *_i = ms_bit(_s);
4      *_j = (_s >> (*_i - MAX_LOG2_SLI)) - MAX_SLI;
5      *_i -= FLI_OFFSET;
6    }
7    else {
8      *_i = ((_s + 3) & (~3))/4;
9    }
10 }
```

As seen in the above source, the function is able to compute equation 3.1 efficiently with basic arithmetic and bits shifting. *MAX_LOG2_SLI* and *MAX_SLI* are

constants depending on r, which is an exponent of the power of two discussed in Section 3.4.2.

- **GetIndex4Search()**

The allocator employs the following equation to find a suitable free memory block to satisfy a request, instead of equation 3.1. Consider an application in which a thread requests 8204 bytes of a memory. The index *I(i, j)* will be *I(13, 0)* by equation 3.1. The list indexed by the I holds free memory blocks whose sizes range from 8177 bytes to 8304 bytes.

$$
\begin{aligned}
I &= \frac{(s+m) \wedge (\neg m)}{4} && \text{if s} \leq 512 \\
I(i,j) &= \begin{cases} i = \lfloor \log_2 (s + 2^{\lfloor \log_2(s) \rfloor - 6} - 1) \rfloor \\ j = \lfloor \frac{(s + 2^{\lfloor \log_2(s) \rfloor - 6} - 1 - 2^i)}{(2^{i-6})} \rfloor \end{cases} && \textit{otherwise}
\end{aligned}
\tag{3.2}
$$

It is possible to ensure that the function always finds a bigger free memory block, even if it tends to cause a small bounded internal fragmentation. The worst-case internal fragmentation occurs when an application requests the free block whose size is one byte larger than an existing segregated list (the algorithm has to find a free block in the next segregated list) and the next list holds the largest size block of its segregated list. The size will be rounded-up to the next list by the equation 3.2. [Masmano et al., 2008a] provided a function, that can be used with algorithms exploiting the two-level segregated list, to calculate the internal fragmentation theoretically. The function[2] gives a fragmentation of $\simeq 3\%$. Interestingly, the evaluation in Appendix A.1 shows very large difference between the observed fragmentation in practice and the theoretical fragmentation.

Source 10 GetIndex4Search()

```
1  void GetIndex4Search(size_t* _s, int *_i, int* _j)
2  {
3    int _t;
4
5    if ( _s > MAX_SIZE_IN_SMALL_BLOCK )
6    {
```

[2] $(2^{i+1}/32) - (2^i + 1) \simeq 2^i/32$, where the second level holds 32 segregated lists.

```
7      _t = (1 << (ms_bit(*_s) - MAX_LOG2_SLI)) - 1;
8      *_s = *size + _t;
9      *_i = ms_bit(*_s);
10     *_j = (*_s >> (*_i - MAX_LOG2_SLI)) - MAX_SLI;
11     *_i -= FLI_OFFSET;
12
13     *_s &= ~_t;
14   }
15   else
16   {
17     *_i = ((_s + 3) & (~3))/4;
18     *_s = *_i * 4;
19   }
20 }
```

The *GetIndex4Search()* function implements equation 3.2, which obtains a starting point to find a free block, and is only invoked when applications request allocating memory. As with *GetIndex4Insert()*, *MAX_LOG2_SLI* and *MAX_SLI* are constants depending on *r*. Using the function, requesting an 8204 byte size of the free block (like in the above example), the index *I(i, j)* will be *I(13, 1)* instead of *I(13, 0)*.

The arena management functions

An arena is a group of memory pools which contains memory blocks that are both released and in use. Each thread is able to have multiple arenas to satisfy its allocation/deallocation requests.

- **InitArena()**

InitArena() initializes the data structure for an arena. It is invoked when the allocator creates a new arena. In particular, it is called to get an additional arena from the second layer, or when a thread or process is created.

Source 11 InitArena()

```
1  void InitArena(arena_t* _NewArena, u32 _rSize, int _node)
2  {
3    u32 RemainSize = _rSize - SIZE_ARENA_HDR;
4    SET_ARENA_SIZE(_NewArena, RemainSize);
5    SET_FREE_ARENA(_NewArena);
6    SET_MOVABLE_ARENA(_NewArena);
7    SET_ARENA_NODEID(_NewArena, _node);
8
9    block_t* NewBlock = (block_t*)GET_FIRST_BLOCK_IN_ARENA(_NewArena);
10   SET_BLOCK_SIZE(RemainSize);
```

```
11    InitBlock(NewBlock, RemainSize);
12    SetBlockIndex(NewBlock);
13    InsertFreeBlock(NewBlock);
14  }
```

The function sets its size, its ccNUMA node ID, and some states which indicate whether it is a free arena or not, and whether it is a moveable arena or not (at lines 3-7). After that, it creates the first free memory block whose size is identical to the size of the arena, and initializes it (at lines 9-12), with the block being inserted into a certain segregated list (at line 13).

- **GetNewArena() and AddNewArena()**

GetNewArena() is invoked when the allocator cannot satisfy the allocation request due to not having enough heap space. In order to increase the available memory of the application, it invokes the *mmap()* or *brk()* system call.

Source 12 GetNewArena()

```
1  static inline void* GetNewArena(size_t* _rSize)
2  {
3    *_rSize = PAGE_CELLING(*_rSize + SIZE_ARENA_HDR + SIZE_BLOCK_OVERHEAD*2);
4    void *newArena;
5    if ((newArena = mmap(0, *_rSize, ...) == MAP_FAILED)
6      ERR("NOT ENOUGH MEMORY\n");
7
8    return newArena;
9  }
```

AddNewArena() is invoked when an arena is created by the system call or when a thread or process is created. It establishes double links between the previous and next arena, storing the information in the TCB. In particular, the new arena will be inserted into the top of the stack in the TCB (at lines 5-11).

Source 13 AddNewArena()

```
1  void AddNewArena(tcb_t* _tcb, arena_t* _newArena)
2  {
3    arena_t* prevArena = _tcb->arena;
4
5    GET_PREV_ARENA(_newArena) = NULL;
6    GET_NEXT_ARENA(_newArena) = prevArena;
7
8    if ( prevArena != NULL )
9      GET_PREV_ARENA(prevArena) = _newArena;
```

```
10
11    (_tcb->arena) = _newArena;
12  }
```

- **AddFreeArena()**

AddFreeArena() is just a wrapper function to insert a free arena into the list. It is used to manage free arena lists in each ccNUMA node. The actual work is delegated to the *PushStack4Arena()* function. As mentioned before, this is more flexible implementation, rather than calling the function of the LIFO stack management directly, because it is easy to remove or modify them when the stack management algorithm is changed. When *AddFreeArena()* is invoked, it just calls *PushStack4Arena()*.

Source 14 AddFreeArena()

```
1   void AddFreeArena(arena_list_t* _ArenaList, arena_t* _Arena)
2   {
3     PushStack4Arena(_ArenaList, _Arena);
4   }
```

- **GetFreeArena()**

When the allocator needs to reuse a free arena stored in a certain free arena list, *GetFreeArena()* is invoked. At line 6, the allocator attempts to find a suitable free arena list to pop. Then it invokes *PopStack4Arena()* (at line 8) if the *GET_NODEARENA()* returns a certain free arena list. The *PopStack4Arena()* function extracts a free arena indicated by *_ArenaList* returned at line 6.

Source 15 GetFreeArena()

```
1   arena_t* GetFreeArena(nMART_t* _nMART, size_t _rSize)
2   {
3     arena_t* arena;
4     int nodeID = GET_NODEID(_nMART, GET_CPUID());
5     _rSize = PAGE_CELLING(_rSize + SIZE_ARENA_HDR + SIZE_BLOCK_OVERHEAD*2);
6     arena_list_t* _ArenaList = GET_NODEARENA(_nMART, nodeID);
7
8     arena=PopStack4Arena(_ArenaList);
9
10    if ( NULL == arena )
11    {
12      arena = GetNewArena(&_rSize);
13      InitArena(arena, _rSize, GET_NODEID(__nMART, GET_CPUID()) );
14    }
15
16    return arena;
17  }
```

At line 12 the allocator will invoke *GetNewArena()* to increase the heap space available if the allocator failed to find a free arena.

- **FreeArena()**

FreeArena() releases a given arena which does not contain any blocks in use; it moves the released arena into the second layer. This function is invoked when the application request releasing the last block in use in the target arena.

Source 16 FreeArena()

```
1  int FreeArena(tcb_t* _tcb, block_t* _block)
2  {
3    ...
4
5    arena_t* curArena = (arena_t*)GET_ARENA_WITH_BLOCK(_block);
6
7    if ( GET_ARENA_REAL_SIZE(curArena) >= MAX_HOLD_ARENA_SIZE )
8    {
9      GET_ARENA_REAL_SIZE(curArena));
10     munmap(curArena, GET_ARENA_REAL_SIZE(curArena));
11     return TRUE;
12   }
13
14   arena_t* prevArena = GET_PREV_ARENA(curArena);
15   if ( prevArena != NULL )
16     GET_NEXT_ARENA(prevArena) = GET_NEXT_ARENA(curArena);
17   else
18     _tcb->arena = GET_NEXT_ARENA(curArena);
19
20   arena_t* nextArena = GET_NEXT_ARENA(curArena);
21   if( nextArena != NULL )
22     GET_PREV_ARENA(nextArena) = GET_PREV_ARENA(curArena);
23
24   size_t real=GET_ARENA_REAL_SIZE_WITH_HD(curArena);
25   size_t idxx = GET_NODEARENA_IDX(GET_ARENA_REAL_SIZE_WITH_HD(curArena));
26   size_t idxArr = GET_INDEX_ARRAY(curArena);
27
28   arena_list_t* _ArenaList = GET_ARENALIST(curArena, real, idxx, idxArr);
29   AddFreeArena(_ArenaList, curArena);
30
31   return TRUE;
32 }
```

At line 5, the pointer address of the current arena has been acquired to which the last free block belongs. If the arena size exceeds the maximum manageable size of the arena, it will be returned to the underlying OS immediately using the *munmap()* system call (at lines 7-12).

To insert the arena into the head of the free arena list, at lines 14-22, the allocator attempts to extract the arena from the TCB by adjusting the previous and next pointer in the TCB header if they exist.

Finally, in order to pass it to the second layer, the allocator gets some useful information to compute the index in the list containing free arenas (at lines 24-26). Then (at lines 28-29), it invokes *AddFreeArena()* to insert the free arena into a certain segregated list in the second layer of the allocator.

The block management functions

- **InitBlock()**

InitBlock() initializes the data structure for a new block. It is invoked when a block is created, and sets the block size, owner ID, and some link information.

Source 17 InitBlock()

```
1  void InitBlock(block_t* _NewBlock, u32 _size)
2  {
3    SET_BLOCK_OWNER(_NewBlock, _tcb);
4    SET_BLOCK_SIZE(_NewBlock, _size);
5    SET_FREE_BLOCK(_NewBlock);
6    SET_PPREV_BLOCK(_NewBlock, NULL);
7    GET_PREV_BLOCK(_NewBlock) = NULL;
8    GET_NEXT_BLOCK(_NewBlock) = NULL;
9  }
```

- **InsertFreeBlock()**

InsertFreeBlock() inserts a free block to the *MATRIX* to be reused later when it is no longer needed.

Source 18 InsertFreeBlock()

```
1  void InsertFreeBlock(block_t* _NewBlock)
2  {
3    int fl, sl, idx;
4    fl = GET_BLOCK_INDEX_FL(_NewBlock);
5    sl = GET_BLOCK_INDEX_SL(_NewBlock);
6    idx = GET_MATRIX_IDX(fl, sl);
7
8    GET_PREV_BLOCK(_NewBlock) = NULL;
9    GET_NEXT_BLOCK(_NewBlock) = MATRIX(_tcb, idx);
10
11   if ( MATRIX(_tcb, idx) != NULL )
```

```
12    {
13      GET_PREV_BLOCK(MATRIX(_tcb, idx)) = _NewBlock;
14      MATRIX(_tcb, idx) = _NewBlock;
15    }
16    else
17    {
18      MATRIX(_tcb, idx) = _NewBlock;
19      set_bit (sl, &(GET_SL(_tcb, fl)) );
20      set_bit (fl, &(GET_FL(_tcb)) );
21    }
22  }
```

The function gets the index *I(i, j)* from the block header (at lines 4-6). The *MATRIX* is logically a two-dimensional array, but it is just implemented using a one-dimensional array so that the index can be represented by only a single variable (at line 6). This makes it easier to implement the allocator, and the index can be easily computed as the following equation:

$$idx = (fl * c) + sl \tag{3.3}$$

c (a constant) is the maximum usable bit position in the bitmap representing the first level. For instance, the index I(1, 3) can be represented as 26 (1*23+3).

At lines 11-21 the block is inserted into the *MATRIX* according to the existence of the previous block in the certain list indexed by *idx*. The allocator sets bits to indicate the list holding a free block if the list is explicitly empty (at lines 19-20).

- **SetBlockIndex()**

SetBlockIndex() is invoked when a new block is created or when a block in use is merged. It sets the index value of the first- and second-level bitmaps, and stores them in the block header. Storing index values in the header allows less computation when it is needed later because the block size should not be changed until the block is merged.

Source 19 SetBlockIndex()

```
1   void SetBlockIndex(block_t* _block)
2   {
3     int fl, sl;
4     GetIndex4Insert(GET_BLOCK_REAL_SIZE(_block), &fl, &sl);
5
```

```
6   GET_BLOCK_INDEX_FL(_block) = fl;
7   GET_BLOCK_INDEX_SL(_block) = sl;
8 }
```

- **FindFreeBlock()**

FindFreeBlock() is the core function for searching for bits set in the first- and second-level bitmaps, and returning the equal size or larger size of a block to satisfy the request.

Source 20 FindFreeBlock()

```
1  block_t* FindFreeBlock(tcb_t *_tcb, int *_fl, int *_sl)
2  {
3    u32 _tmp = _tcb->sl[*_fl] & (~0 << *_sl);
4    block_t *_b = NULL;
5
6    if (_tmp)
7    {
8      *_sl = ls_bit(_tmp);
9      _b = MATRIX(_tcb, GET_MATRIX_IDX(*_fl,*_sl));
10   }
11   else
12   {
13     *_fl = ls_bit(_tcb->fl & (~0 << (*_fl + 1)));
14     if (*_fl > 0)
15     {
16       *_sl = ls_bit(_tcb->sl[*_fl]);
17       _b = MATRIX(_tcb, GET_MATRIX_IDX(*_fl,*_sl));
18     }
19   }
20   return _b;
21 }
```

Two bitmaps *(fl, sl)* refer to all sizes of blocks that are manageable, but the allocator does not need to search for bits indicating smaller sizes of blocks than the requested size so that lower positions of bits are ignored at line 3.

At lines 6-10 the allocator attempts to find the LSB in *tmp* to get the index of the column of *MATRIX*, if the segregated lists indicated by *tmp* are not empty. Otherwise, at lines 11-19 the allocator attempts to find a larger-size block in the upper bits in the first-level bitmap.

- **Extract4Free()**

Extract4Free() extracts a free block from a particular segregated list. This function is invoked when two or more adjacent blocks are merged.

Source 21 Extract4Free()

```
void Extract4Free(tcb_t* _tcb, block_t* _block)
{
  int _fl, _sl;

  _fl = GET_BLOCK_INDEX_FL(_block);
  _sl = GET_BLOCK_INDEX_SL(_block);

  block_t* prev = GET_PREV_BLOCK(_block);
  block_t* next = GET_NEXT_BLOCK(_block);
  if ( next != NULL )
    GET_PREV_BLOCK(next) = prev;
  if ( prev != NULL )
    GET_NEXT_BLOCK(prev) = next;

  if ( MATRIX(_tcb, GET_MATRIX_IDX(_fl,_sl)) == _block )
  {
    MATRIX(_tcb, GET_MATRIX_IDX(_fl,_sl)) = GET_NEXT_BLOCK(_block);
    if ( NULL == MATRIX(_tcb, GET_MATRIX_IDX(_fl,_sl)) )
    {
      clear_bit (_sl, &(_tcb->sl[_fl]));
      if (!_tcb -> sl[_fl])
        clear_bit (_fl, &(_tcb->fl));
    }
  }
  GET_PREV_BLOCK(_block)=NULL;
  GET_NEXT_BLOCK(_block)=NULL;
}
```

At lines 5-6 the allocator gets the values of rows and columns of the *MATRIX* from the block header. After that, in order to extract the target block from the segregated list, it adjusts the associated pointers of the previous and next blocks (at lines 8-13). The allocator clears a particular bit to 0 if the *_block* is the last one, which belongs to the list indicated by *fl* and *sl* (at line 15-24).

After this function the target block *_block* will be completely disconnected from all blocks and the MATRIX, and will be ready to be merged with adjacent blocks which have been freed already.

- **Extract4Alloc()**

Extract4Alloc() is similar to the *Extract4Free()* function. It is invoked when the application requests a memory block. It extracts a free block from a certain segregated list.

Source 22 Extract4Alloc()

```
1  void Extract4Alloc(tcb_t* _tcb, block_t* _block, int _fl, int _sl)
2  {
3    MATRIX(_tcb, GET_MATRIX_IDX(_fl,_sl)) = GET_NEXT_BLOCK(_block);
4    if ( NULL == GET_NEXT_BLOCK(_block) )
5    {
6      clear_bit (_sl, &(_tcb->sl[_fl]));
7      if (!_tcb -> sl[_fl])
8        clear_bit (_fl, &(_tcb->fl));
9    }
10   GET_PREV_BLOCK(_block)=NULL;
11   GET_NEXT_BLOCK(_block)=NULL;
12 }
```

At line 3 the allocator re-establishes the pointer of MATRIX to the header of the next block. If the target block is the last block in the list, the allocator sets a particular bit of the first- and second-level bitmap to 0 (at lines 4-9).

- **MergeFreeBlock()**

MergeFreeBlock() merges two adjacent free blocks. This function is invoked when the application requests releasing a block. Merging two adjacent blocks is extremely simple. It just needs to remove one of the headers, setting a new size to total the sum of their sizes. For instance, if a block A whose size is 1500 bytes and, physically, the next block B whose size is 1000 bytes need to be merged, the header of B is eliminated and the size of A will be changed to 2500 bytes.

Source 23 MergeFreeBlock()

```
1  void MergeFreeBlock(block_t** _block)
2  {
3    block_t* tmpBlock = GET_PNEXT_BLOCK(*_block);
4    if ( (FREE == GET_BLOCK_STATUS(tmpBlock)) )
5    {
6      Extract4Free(_tcb, tmpBlock);
7      SET_BLOCK_SIZE(*_block, COMPUTE_BLOCK_SIZE(....) );
8    }
9
```

```
10    if ( (GET_PPREV_BLOCK(*_block) != NULL) &&
11        (FREE == GET_BLOCK_STATUS(GET_PPREV_BLOCK(*_block))) )
12    {
13      tmpBlock = GET_PPREV_BLOCK(*_block);
14      Extract4Free(_tcb, tmpBlock);
15      SET_BLOCK_SIZE(tmpBlock, COMPUTE_BLOCK_SIZE(....) );
16
17      *_block = tmpBlock;
18    }
19    tmpBlock = GET_PNEXT_BLOCK(*_block);
20    GET_PPREV_BLOCK(tmpBlock) = *_block;
21  }
```

Firstly, the allocator gets a pointer to the next block physically (at line 3), with the next block being examined to see whether it is free or not. The next block will be extracted from the list if it is free (at lines 4-8).

Next, if the physically previous block is free, it will be extracted from the list to be merged (at lines 10-18). In order to merge two adjacent blocks, it is necessary to only adjust its size, as seen at lines 7 and 15.

At lines 19-20 the allocator sets two pointers to the previous and next blocks physically.

- **SplitBlock()**

The *SplitBlock()* function is invoked when a larger block than the request size is provided to satisfy the request of the memory allocation. The larger block will be split if the size of the larger block exceeds the split threshold. The split threshold is the sum of the request size, the minimum block size that the allocator is able to allocate, and the overhead of the header.

Source 24 SplitBlock()

```
1   void SplitBlock(block_t* _block, size_t _rSize)
2   {
3     block_t* remainBlock = (block_t*)((char*)&(_block->ptr.buf) + _rSize);
4
5     u32 RemainSize = GET_BLOCK_REAL_SIZE(_block) - _rSize -  SIZE_BLOCK_OVERHEAD;
6     InitBlock(remainBlock, RemainSize);
7     SetBlockIndex(remainBlock);
8     SET_PPREV_BLOCK(remainBlock, _block);
9     InsertFreeBlock(remainBlock);
10
11    SET_BLOCK_SIZE(_block, _rSize );
12  }
```

To split the larger block, it is necessary to append a new header for the split block at the end of the data of the original block. In order to set the header position for a new split block, the allocator computes the position (at line 3). After that, it computes its size, initializes the new block, and establishes some pointers to the adjacent block. At line 9 the new split block will be inserted into the *MATRIX*.

3.5 Summary

nMART is a new ccNUMA-aware dynamic memory management supporting real-time systems on ccNUMA architectures. Most application developers in the real-time domain avoid using dynamic storage allocation algorithms due to unbounded response time and fragmentation. However, *nMART* does provide bounded response time and low fragmentation, and consequently facilitates dynamic storage allocation.

As discussed above, all our design principles focus on achieving the objectives: improving the performance of allocation/de-allocation memory operations and reducing fragmentations. In order to improve the performance, *nMART* exploits a more accurate measurement of node distance, multiple heaps, which provides the per-thread-based private heap to reduce lock contentions between threads executing on the same node sharing the node-based page lists, tracks all *arena*s from where they have been taken, and manages all pages based on each ccNUMA node. In terms of reducing fragmentations, *nMART* adopts multiple maintenance strategies for different size of blocks.

nMART is based on a combination of policies, which are *best-fit*, *good-fit*, and *first-fit*. These policies are able to avoid an exhaustive search to find the most appropriate block in its data structure. Exploiting three bitmaps allows us to ensure bounded response time and low fragmentation. The allocator is able to achieve a constant search time based on each segregated list which indexes a range of block sizes. The mapping functions are based on using bit shifts, which are very efficient. Moreover, all of the functions discussed in this chapter are designed to perform their functionalities without a single loop, except for the *CheckRemoteFreeBlock()*. The

number of iterations in that loop depends on the number of free blocks, which are released by the *remote-free* technique. The only function containing a single loop is a trade-off between the predictability and the performance/space efficiency. It is able to improve the performance of allocation/deallocation operations, but it may consume more memory space due to deferred coalescing. However, as mentioned earlier, Larson [Larson and Krishnan, 1998] has observed that remote releases are in the 2% to 3% range on large and long-running applications. As a reason, the time complexity of the *nMART* can be amortized $\mathcal{O}(1)$.

We have discussed how to achieve the third layer's objective of supporting the characteristics of ccNUMA architecture systems for *nMART*. All modern operating systems have encapsulated the underlying architecture design to developers as well as end-users. Unfortunately, the hardware abstraction provided by the underlying OS is not appropriate to discover the system resources accurately, thereby making it difficult to develop predictable applications on these systems. The ACPI has been introduced to provide some transparency by hardware vendors, but it does not provide enough information to describe the underlying system exactly, especially the node distances. For this reason, we have proposed a more accurate model to measure the node distances, which is obtained via static node distance analysis. This chapter has described our new model, and shown how the model can be approximated in the Linux kernel. Experiments, discussed in Appendix B.3, have shown that our model increases the performance of the underlying system. It is also able to access the closest or closer remote nodes wherever this is possible. In the results, the performance improvement of average execution time for allocation operations varied from 5.1% to 17.4% approximately according to processor and memory affinities. The results show our model is a reasonable model to estimate the node distances, providing a closer node if it is available.

In combination, all these features make the algorithm acceptable for real-time systems, whose most important requirement is a predictable time response.

Chapter 4

Evaluation

All algorithms described in this thesis have been evaluated empirically using experiments on actual hardware. The majority of experiments are designed to show whether our objectives have been met. The chapter gives an experimental study in comparison with a representative set of dynamic memory allocators: *Best-fit*, *First-fit*, *Half-fit*, *Hoard*, *tcmalloc*, *TLSF*, and *nMART*. In order to give fair comparisons and results, we describe the experimental environment in Section 4.1 and workload models in Section 4.2. Temporal behaviour analysis in Section 4.3 gives the performance of the memory allocators. In order to compare the support for ccNUMA architecture systems, the analysis of local and remote access latencies are drawn in Section 4.3.3. Also, other evaluations are shown in the Appendices. Spatial behaviour analysis in Appendix A.1 shows the space efficiency of each algorithm. Cache behaviour analysis is shown in Appendix A.2 to illustrate the cache effect of each allocator on the experimental system.

4.1 Experimental Environment

The experimental hardware consists of two ccNUMA architecture systems. One is a four node-based system and another is an eight node-based system.

A four nodes-based ccNUMA architecture system

Figure 4.1 [Weber, 2001] (where SRI means a system request interface, XBAR indicates a crossbar, MCT denotes a local memory controller, and HT means HyperTra-

nsport^{TM} technology) shows the hardware platform, which is a Supermicro H8QME-2 serverboard consisting of nVidia MCP55Pro and AMD 8132 chipset. Our system contains SDRAM of 16G bytes, and four AMD Opteron processors. Each processor comprises four cores with all processors being connected to other processors via a HyperTransport (cHT). Each core has a 64K bytes non-shared L1 cache, 512K bytes non-shared L2 cache, and there is 2M bytes L3 shared cache memory among all cores. An individual processor comprises a ccNUMA node with up to 4G bytes of memory, so in total the system has 16G bytes main memory. The interconnection is configured with a clock frequency of 1GHz as the maximum speed supported by the system. The machine runs the Ubuntu distribution of the Linux kernel version 3.0.4. In practice, the Linux kernel treats a core as a processor so that the system has in total sixteen processors.

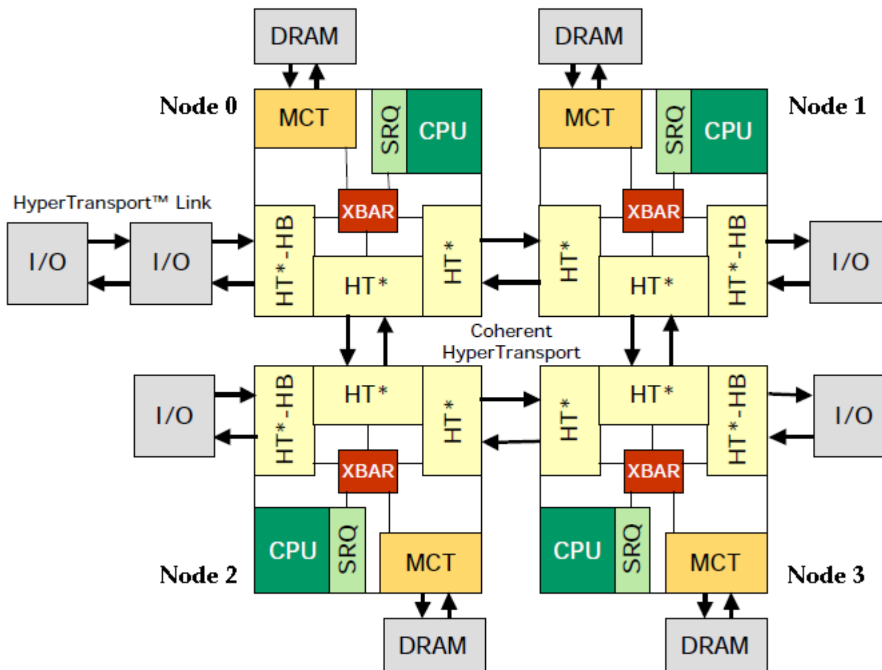

Figure 4.1: A four node-based ccNUMA architecture system

An eight nodes-based ccNUMA architecture system

In order to ensure showing the inaccurate values of the original node distance table used by all operating systems, we also show the result of an experiment to

measure the actual time taken to access remote nodes in an eight node-based cc-NUMA architecture system. We have experimentally measuring the cost of memory accesses using the same method.

The more complex architecture system contains four physical processors with 128G bytes main memory entirely. Each processor comprises eight cores, with all processors being connected to each other via HyperTransport. Each core has a 128K bytes non-shared L1 cache, 512K bytes non-shared L2 cache, and 12M bytes L3 cache memory is shared by all cores. This system is comprised of eight ccNUMA nodes instead of four nodes. This is because each four cores on the same processor have been grouped as one node, so a processor consists of two ccNUMA nodes. We have called the relationship between two nodes belonging to the same processor *sibling relationship* or *sibling nodes*. Each individual processor has 32G bytes main memory.

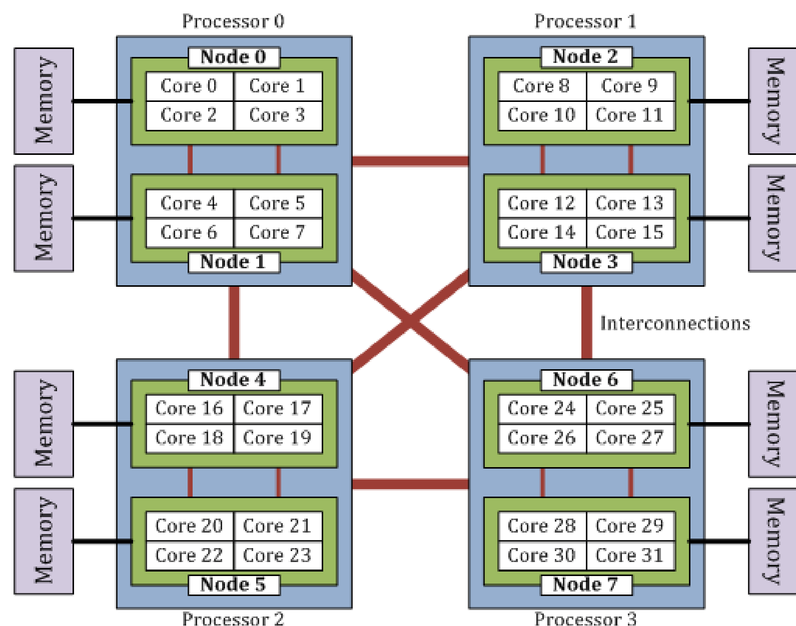

Figure 4.2: A more complex architecture system

4.2 Workload Models

In general, two different workloads are used to evaluate DSAs. Evaluating fragmentation and worst-case scenarios is performed using synthetic workloads, while

comparing average execution time is performed using real workloads. In this section, we discuss both workload models, which are used to evaluate several memory allocation algorithms as well as our prototype on nMART.

4.2.1 Real Workloads

Real workloads are more accurate than using synthetic workloads for evaluating DSAs, since synthetic workloads are generated by means of simplifying assumptions. However, it is hard to find acceptable real-time applications, which are able to evaluate the behaviour of DSAs on real-time systems. Moreover, the disparity between average cases and worst-case scenarios is very wide, as the worst case rarely occurs [Robson, 1977]. For this reason, we have chosen four applications with three test sets, which are allocation-intensive and have varying memory usage patterns; they have been used in many previous studies on DSAs [Berger et al., 2000] [Puaut, 2002] [Hasan and Chang, 2005].

cfrac *cfrac* is an application to factorize large integers by means of the continued fraction method. We have used examples distributed with its source codes.

- Test set 1 The input number, 327905606740421458831903, was the 24-digit number;
- Test set 2 The input number, 4175764634412486014593803028771, was the 31-digit number.
- Test set 3 The input number, 41757646344123832613190542166099121, was the 35-digit number.

espresso *espresso*, version 2.3, is an application which optimizes logic circuits. The input files were three of the circuit examples distributed with its source codes.

- Test set 1 The input logic circuit consisted of 7 inputs and 10 outputs (*Z5xp1.espresso*).
- Test set 2 The input logic circuit consisted of 8 inputs and 8 outputs (*mlp4.espresso*).

- Test set 3 The input logic circuit consisted of 16 inputs and 40 outputs (*largest.espresso*).

gawk Gnu *awk* is a free software implementation of the AWK interpreted programming language for data extraction and reporting. We have used the default *gawk* provided by *Ubuntu* distribution. Also, we have used the *wordfreq.awk* example, which collects statistics on how often different words appear.

- Test set 1 The input file consisted of 7,500 words and 629 lines (*prog-small-data.awk*).
- Test set 2 The input file consisted of 25,144 words and 25,144 lines (*words-small.awk*).
- Test set 3 The input file consisted of 69,965 words and 69,964 lines (*words-large.awk*).

p2c *p2c*, version 1.21alpha-07, is a tool for translating Pascal programs into C. Usually, the input file consists of a set of Pascal source files, and the output is a set of .c and .h files. We have used the Pascal source codes that are distributed with the *p2c* distribution..

- Test set 1 This input file consisted of 299 characters and 22 lines of pascal statements (fact.p).
- Test set 2 This input file consisted of 11,280 characters and 359 lines of pascal statements (cref.p).
- Test set 3 This input file consisted of 66,611 characters and 2,214 lines of pascal statements (basic.p).

There is no standard evaluation tool for multi-threaded allocators. In addition, no tools exist to stress multi-threaded performance of relatively long-running applications like real-time applications. For this reason, we have chosen four other applications used in previous studies for multi-threaded allocators [Berger et al., 2000] [Larson and Krishnan, 1998].

larson *larson* was introduced by Paul Larson at Microsoft Research [Larson and Krishnan, 1998]. It simulates a server to provide the remote freeing of memory. Each thread allocates and releases memory blocks, with some randomly chosen blocks being passed to other threads to be released. We have used the following parameters. Respectively, parameters give *sleep()* time, the minimum size of blocks, maximum size of blocks, number of blocks per thread, number of iterations, random seed, and number of threads.

- Test set 4: 1 32 32768 1000 1000 927 16

cache-scratch *cache-scratch* [Berger et al., 2000] simulates a passive false-sharing situation, and tests resilience against passive false sharing. The following parameters give the number of threads, number of iterations, requested memory size, and number of inner iterations for write operations.

- Test set 4: 32 100 8 1000000

cache-thrash *cache-thrash* [Berger et al., 2000] simulates an active false-sharing situation, and tests resilience against passive false sharing. We have used the same parameters as cache-scratch.

- Test set 4: 32 100 8 1000000

shbench *shbench* is a memory allocator stress test tool from MicroQuill [MicroQuill, 2012]. Each thread allocates and randomly releases a number of randomly sized blocks in a random order. We have used the following parameters. Parameters give the number of memory blocks and number of threads.

- Test set 4: 10000000 4

The execution of real workloads is more complex than the execution of synthetic scenarios. Sometimes, such applications require services provided by the underlying operating system. These kinds of services, such as disk I/O, incur delay when measuring performance. To minimize the impact of these services, we have turned

off process swapping to avoid pages swapped out into disks, and all service daemons, such as the X server, web server, and database server, which do not have any relationship with our experiments.

In order to ensure results are accurate, we have also used the response time obtained by measuring the processor cycles with the number of instructions executed, rather then system calls like *gettimeofday()*. Measuring processor cycles is based on a time stamp counter called *tsc*. On x86/64 architecture systems, it can be accessed using *rdtsc* instruction. This counter is usable for a variety of tasks, as it has excellent high-resolution with a low overhead, but it requires scaling of processor cycles. For instance, *Enhanced Intel SpeedStep*® (EIST) or AMD *Cool'n'Quiet*® technology may cause the processor cycle to be dynamic scaled at runtime. The impact of the technology causes the counter to be less accurate, as the length of the processor cycle may change frequently. For this reason, we have had this feature turned off in the system BIOS and disabled. Moreover, to keep the result of *tsc* accurate, the processor pipeline must be flushed before calling *rdtsc*. This is the reason why we have called the *cpuid* instruction before *rdtsc*. All results of experiments have been produced in this experimental environment.

As well as the time stamp counter, in general, modern processors are equipped with particular hardware performance counters that allow tracking system events, such as cache misses and executed instructions. In order to measure these performance counters, especially cache loads and misses and the number of instructions executed, we have employed *perf (Performance Counters for Linux)*, which is a performance analyzing tool that produces much more reasonable analysis [Nethercote and Mycroft, 2002]; it is available since Linux kernel 2.6.31, for profiling application, memory, and cache behaviours [Molnar, 2009] [Edge, 2009]. In Linux, performance counters are kernel-based subsystems providing a framework for performance analysis, including hardware-level and software-level. Unlike *valgrind* [Weidendorfer et al., 2004] [McCamant and Ernst, 2007] [Tao et al., 2008], they are capable of statistical profiling of a single processor, several threads or an entire system in both kernel and user space.

A summary of the real workload characteristics is given in tables 4.1, 4.2, 4.3, and 4.4. The last two rows of the tables are produced when using the default *glibc* allocator. Each table represents an amount of work from light to heavy in the order that is assigned to each allocator, and will be compared. In particular, table 4.4 is prepared for comparing varying allocators in a multi-threaded environment.

Except for the last two rows, processor cycles and instructions, all results in tables 4.1, 4.2 and 4.3 have been obtained using the dynamic linking loader, *dlsym()*.

Test set 1	cfrac	espresso	gawk	p2c
number of malloc()	1528	24225	4534	2675
number of free()	1230	24224	3763	611
total allocation size	29604	1693502	373559	166034
largest allocation size	284	4632	16384	1024
average allocation size	19	70	82	62
processor cycles	104699403	1407165	2155146	963251
instructions	122593809	27560262	28503704	22849408

Table 4.1: Real workload characteristics of test set 1

Test set 2	cfrac	espresso	gawk	p2c
number of malloc()	5277	59827	126794	6102
number of free()	4952	59826	126373	3372
total allocation size	108899	5285233	6245812	298792
largest allocation size	688	5448	131048	1024
average allocation size	21	88	49	49
processor cycles	1236777271	78801871	233334208	10888057
instructions	1206790730	116022605	201536776	22808989

Table 4.2: Real workload characteristics of test set 2

To the best of our knowledge, unfortunately, there is no implementation of *Best-fit*, *First-fit*, and *Half-fit* memory allocators on the Internet, which are able to replace default memory allocators provided by the underlying operating system. There are only few allocators, which cannot satisfy our standard because they are based

Test set 3	cfrac	espresso	gawk	p2c
number of malloc()	8858	1652970	351431	43936
number of free()	8518	1652969	350474	39176
total allocation size	191703	182929348	17319441	1496159
largest allocation size	1040	38496	261992	1024
average allocation size	22	111	49	34
processor cycles	4470015910	2047549487	546009716	221177907
instructions	3847942214	3334543200	545757202	168842098

Table 4.3: Real workload characteristics of test set 3

Test set 4	larson	cache-scratch	cache-thrash	shbench
number of malloc()	628256	3266	3233	2502057
number of free()	612256	3266	3233	2500017
largest allocation size	32767	2048	2048	20000
average allocation size	16293	9	9	130
processor cycles	270864287562	34543354638	33319278375	114353243368
instructions	220833518162	15610920669	15569018061	78609628514

Table 4.4: Real workload characteristics of test set 4

on an array. For this reason, we have implemented *Best-fit*, *First-fit* and *Half-fit* allocators as well as a prototype of our memory manager. The implementation of these algorithms is given in Appendix D.2.

Tables 4.5, 4.6, 4.7 and 4.8 show, for the three test sets of workloads, the results of the number of *malloc()* and *free()* calls by each of the applications. These tables show different function call counts even when these numbers are collected using the same application. For instance, in table 4.5 the number of *malloc()* calls of SET 1 varies from 1,528 to 1,530. This is because the number of function calls depends on the allocator implementation. In other words, if some functions, such as *realloc()* or *calloc()*, are implemented to invoke *malloc()* or *free()*, the call numbers may slightly increase.

cfrac						
	SET 1		SET 2		SET 3	
	# of malloc	# of free	# of malloc	# of free	# of malloc	# of free
Best	1,528	1,230	5,277	4,952	8,858	8,518
First	1,528	1,230	5,277	4,952	8,858	8,518
Half	1,528	1,230	5,277	4,952	8,858	8,518
Hoard	1,528	1,232	5,277	4,954	8,858	8,520
nMART	1,528	1,230	5,277	4,952	8,858	8,518
tcmalloc	1,539	1,237	5,288	4,959	8,869	8,525
TLSF	1,528	1,232	5,277	4,954	8,858	8,520

Table 4.5: The number of *malloc()* and *free()* calls by *cfrac*

In table 4.5 the *tcmalloc* allocator shows the highest calls recorded throughout all test sets for the allocation and de-allocation.

In table 4.6 the *Hoard* allocator shows the highest calls recorded throughout all test sets for the allocation and de-allocation.

In table 4.7 the *Hoard* allocator shows the highest calls recorded throughout test set 1 and test set 2 for the allocation and de-allocation, and *TLSF* gives the highest calls for test set 3. Regarding de-allocation, *Hoard*, *tcmalloc* and *TLSF* show higher *free()* calls than others. In *TLSF*'s cases, the number of *free()* calls exceeds

espresso						
	SET 1		SET 2		SET 3	
	# of malloc	# of free	# of malloc	# of free	# of malloc	# of free
Best	24,658	24,657	61,337	61,336	1,668,384	1,668,383
First	24,658	24,657	61,337	61,336	1,668,384	1,668,383
Half	24,658	24,657	61,337	61,336	1,668,384	1,668,383
Hoard	24,761	25,750	61,438	62,572	1,675,528	1,708,176
nMART	24,658	24,657	61,337	61,336	1,668,384	1,668,383
tcmalloc	24,730	25,292	61,405	61,019	1,675,429	1,692,061
TLSF	24,290	25,287	59,872	61,014	1,659,400	1,692,056

Table 4.6: The number of *malloc()* and *free()* calls by *espresso*

gawk						
	SET 1		SET 2		SET 3	
	# of malloc	# of free	# of malloc	# of free	# of malloc	# of free
Best	4,597	3,060	126,919	101,229	246,139	196,405
First	4,597	3,060	126,919	101,229	246,139	196,405
Half	4,597	3,060	126,919	101,229	246,139	196,405
Hoard	4,598	3,829	126,920	126,501	246,140	245,483
nMART	4,597	3,060	126,919	101,229	246,139	196,405
tcmalloc	4,564	3,770	126,850	126,380	246,099	245,411
TLSF	3,764	3,765	101,521	126,375	281,338	350,476

Table 4.7: The number of *malloc()* and *free()* calls by *gawk*

the number of *malloc()* calls because *gawk* invokes *realloc()* and *calloc()* functions repeatedly.

p2c						
	SET 1		SET 2		SET 3	
	# of malloc	# of free	# of malloc	# of free	# of malloc	# of free
Best	2,679	615	6,107	3,377	43,942	39,182
First	2,679	615	6,107	3,377	43,942	39,182
Half	2,679	615	6,107	3,377	43,942	39,182
Hoard	2,679	1,681	6,107	4,957	43,942	43,418
nMART	2,679	615	6,107	3,377	43,942	39,182
tcmalloc	2,690	1,682	6,118	4,957	43,953	43,417
TLSF	2,675	1,677	6,102	4,962	43,936	43,412

Table 4.8: The number of *malloc()* and *free()* calls by *p2c*

In table 4.8 the *tcmalloc* allocator shows the highest call number recorded for test set 1 in the allocation and de-allocation. In test set 2, *tcmalloc* for allocation and *TLSF* for de-allocation show the highest function calls. *tcmalloc* for allocation and *Hoard* for de-allocation show the highest function calls.

4.2.2 Synthetic Workloads

Since the paper by [Zorn and Grunwald, 1994], it usual to avoid using synthetic workloads to predict the behaviour of dynamic storage allocation algorithms because none of the models show good predication of the maximum blocks or the size of blocks allocated by all the applications. However, given the non-existence of real-time applications which exploit the characteristics of ccNUMA architectures with dynamic storage allocation algorithms, we have no choice but to use synthetic workloads. In order to perform an evaluation of the allocators under synthetic workloads, a load model has been designed under the combined models proposed by [Zorn and Grunwald, 1994] and [Marchand et al., 2007]. We have chosen two models which have been obtained from actual application behaviours; of the five models in the paper, the authors concluded that these two models are more accurate than others.

The main objective of the models is to abstract the behaviour of the real application in three parameters: block size (B_S), which indicates the average block size required; block holding time (B_{HT}), which indicates the time elapsed between blocks assigned to the application and being released; and block interarrival time (B_{IT}), which is the time elapsed between an allocation request and the next allocation request.

The Mean-Value Model (MEAN) The MEAN model abstracts the behaviour of an actual application using three parameters: the mean of B_S, B_{HT} and B_{IT}. These parameters are used to generate random values from zero to twice the mean with a uniform distribution. A variant of the model tracks the mean and variance of each parameter to generate samples from three normal distributions from tracked values [Zorn and Grunwald, 1994] [Puaut, 2002].

Cumulative Distribution Functions (CDF) This model constructs the actual CDF using the B_S, B_{HT} and B_{IT} values, which observed data. These functions with a uniform distribution in [0, 1[are used to abstract the behaviour of applications. The following is an example of CDF, which indicates that there is an equal distribution of size blocks between a range from 128 to 512 and a range from 512 to 4k bytes:

$$CDF(s) = \begin{cases} 0.0 & if \quad s < 128 \\ 0.33 & if \quad 128 \leq s < 512 \\ 0.66 & if \quad 512 \leq s < 4096 \\ 1.0 & if \quad s \geq 4096 \end{cases}$$

The experimental results are presented in Appendix C. In particular, the results in tables C.1, C.2, C.3, C.4, C.5, C.6, C.7, C.8, C.9, C.10, C.11, C.12 and C.13 are collected during each application execution. In order to collect statistics accurately, we modified and used our implementation of the *First-fit* allocator.

As seen in table C.1, the result shows that *cfrac* requests small-size memory blocks throughout all test sets, with its standard deviations being also small. Except for test set 1, the holding time exceeded the interarrival time. It shows that the amount of memory usage is increasing linearly because of releasing memory blocks rarely.

In tables 4.1, 4.2 and 4.3, *espresso* is a memory-intensive application, With the *cfrac* model, it requests and releases lots of memory blocks. None of the holding time exceeds the interarrival time throughout all test sets. In particular, the holding time of test sets 1 and 2 is also small compared to other application sets.

Unlike other applications, *gawk* sometimes requires larger-size memory blocks up to 256k bytes. Except for test set 1, it also requires lots of memory during application execution, with its holding time exceeding the interarrival time. In our observation, the application does not release memory completely during its execution, and expects that the underlying operating system will release memory blocks when the program terminates.

In tables 4.1, 4.2 and 4.3, *p2c* also does not release memory blocks completely, even though its interarrival time is higher than the holding time. The responsibility of releasing remains with the operating system. It requests small-sizes memory blocks up to 1k bytes compared to the memory-intensive applications: *espresso* and *gawk*.

4.2.2.1 MEAN-Value Model

The model proposed by [Zorn and Grunwald, 1994] characterizes the behaviour of an actual application, as seen in table C.1, which gives the mean values (and standard deviations) of three parameters, which were obtained during the execution of each application.

In the table, it shows the mean of block sizes requested, the mean of interarrival time elapsed, and the mean of block holding time elapsed on each application. Note that $\mathrm{B_S}$ indicates the mean of block sizes requested, $\mathrm{B_{IT}}$ denotes the mean of interarrival time elapsed, and $\mathrm{B_{HT}}$ indicates the mean of block holding time elapsed. Columns starting “Std.” beside each metric denote the standard deviation of each metric.

Using Table C.1, we have generated a workload under the [Zorn and Grunwald, 1994] approach. However, modern processors are faster than at the time the paper was published, so that the interarrival time and holding time are mostly too small and the maximum of alive memory blocks is only 37 blocks in all execution times.

Moreover, we need to consider the characteristics of real-time applications, in that the lifetime is typically longer than general-purpose applications' lifetime possibly as much as days, months or even years. Also, ccNUMA architecture systems are more complex and powerful than uniprocessor or SMP systems so that they are running as servers, such as a database and web servers. Characteristics of such servers tend towards applications requesting larger-size memory blocks and keeping them during their lifetimes. Generally, many servers allocate a huge amount of memory at their start-up time to provide a memory pool internally. The main objective of the memory pool is to keep track of all memory allocations and to release all allocated blocks automatically. In one case, *Oracle* allocated over 350M bytes of memory at start-up time to be shared globally. This is the most common usage, called plateau (discussed in 2.1.3), for many service servers [Oracle Inc., 2013] [Sybase Inc., 2013] [Apache Software Foundation, 2013]. For these reasons, we have designed a new workload model based on the original MEAN and CDF models to reflect modern architectures and to target applications. The proposed synthetic workload model generates random numbers under the following premises:

- The size of memory block (B_S) for allocation is randomly generated from a uniform distribution in a specified range from 1 to twice the size of the mean of block sizes in Table C.1. In the original approach, the size of the memory block starts from 0, but there is no meaning of zero size of allocation, so we have adjusted the value from zero to 1 for allocation.

- The size of a large amount of memory is randomly generated in a range from 256M bytes to 512M bytes, will be allocated once at start-up time, and be alive until termination of the application.

- The interarrival time (B_{IT}) is determined using a uniform distribution in a specified range from 0 to twice the value of the mean of the interarrival time in the table.

- The holding time (B_{HT}) is determined using a uniform distribution in a specific range from 0 to twice the value of the mean of the holding time in the table.

- The number of memory requests is fixed at allocations of 100,000 times, to better reflect block size distribution.

- To measure remote memory access latencies, we have read and written meaningless data many times, iteratively ranging from 30 to 50, onto the large amount of memory, which was allocated at the application's start-up time.

The following table 4.9 has been generated using the above premises.

	Mean	Min	Max
B_S	55	1	110
B_{IT}	80,338	0	160,675
B_{HT}	146,824	0	293,648

Table 4.9: The MEAN of B_S, B_{IT} and B_{HT} generated

4.2.2.2 The CDF Model

Similar to the MEAN model, we have designed the workload model of CDF, as follows. The cumulative percentage and frequency of B_S, B_{IT} and B_{HT} of all applications are shown in tables C.2, C.3, C.4, C.5, C.6, C.7, C.8, C.9, C.10, C.11, C.12 and C.13.

- The size of memory block (B_S) for allocation is randomly generated from a uniform distribution in each distinct range, as seen in Table 4.10.

- The size of a large amount of memory blocks has randomly generated in a range from 256M bytes to 512M bytes, will be allocated once at start-up time, and be alive until termination of its application.

- The interarrival time (B_{IT}) is determined using a uniform distribution in each specified range, as seen in the table.

- The holding time (B_{HT}) is determined using a uniform distribution in each distinct range, as seen in the table.

- The number of memory requests is fixed as allocations of 100,000 times to better reflect block size distribution.

- To measure remote memory access latencies, we have read and written meaningless data many times, iteratively ranging from 30 to 50, onto the large amount of memory block which was allocated at application's start-up time.

Based on these premises, we have generated table 4.10, which gives the number of occurrences of each distinct size, holding time and interarrival time.

$$\mathrm{B_S}(x) = \begin{cases} 0.00 & x < 8 \\ 0.08 & x < 16 \\ 0.37 & x < 32 \\ 0.44 & x < 64 \\ 0.89 & x < 128 \\ 0.94 & x < 256 \\ 0.97 & x < 512 \\ 0.99 & x < 1k \\ 1.0 & x \geq 2k \end{cases} \quad \mathrm{B_{IT}}(x) = \begin{cases} 0.00 & x < 256 \\ 0.05 & x < 512 \\ 0.64 & x < 1k \\ 0.75 & x < 2k \\ 0.91 & x < 4k \\ 0.95 & x < 8k \\ 0.98 & x < 16k \\ 0.99 & x < 32k \\ 1.00 & x \geq 64k \end{cases} \quad \mathrm{B_{HT}}(s) = \begin{cases} 0.00 & x < 8 \\ 0.42 & x < 16 \\ 0.49 & x < 32 \\ 0.55 & x < 64 \\ 0.61 & x < 128 \\ 0.67 & x < 256 \\ 0.72 & x < 512 \\ 0.77 & x < 1k \\ 0.81 & x < 2k \\ 0.83 & x < 4k \\ 0.87 & x < 8k \\ 0.90 & x < 16k \\ 0.93 & x < 32k \\ 0.95 & x < 64k \\ 0.96 & x < 128k \\ 0.97 & x < 256k \\ 0.98 & x < 512k \\ 0.99 & x < 1m \\ 1.00 & x \geq 2m \end{cases}$$

Table 4.10: The CDF model of $\mathrm{B_S}$, $\mathrm{B_{IT}}$ and $\mathrm{B_{HT}}$ generated

In summary, we have evaluated all algorithms empirically using real workloads (four test sets) and synthetic workloads (MEAN-value and CDF model) on actual hardware, the four node-based ccNUMA architecture system.

4.3 Temporal Behaviour Analysis

In this section, we will discuss the temporal behaviour of the memory allocator algorithms by comparing the longest execution time and the total execution time elapsed for allocation and de-allocation of each allocator. Also, local and remote

access latencies for each allocators are evaluated. As previously discussed, benchmarks for allocators on ccNUMA architecture systems lag behind the introduction of new hardware. In our case, the situation is exacerbated because there is no model to evaluate memory management algorithms for real-time applications supporting ccNUMA architectures. For these reason, we have used synthetic workloads to evaluate local and remote access latencies of each algorithms. The longest execution time depends on the worst case execution time in practice and the total execution time indicates the performance of memory management algorithms. Local and remote access latencies depend on the support efficiency of ccNUMA architecture systems.

4.3.1 The Longest Execution Time

During allocation, memory allocators search for an unused memory block to satisfy the request. In the worst case, it fails and the allocator must request an increase in heap space from the underlying operating system, splitting the return space. However, in normal cases, allocators just provide a free block without any additional operations. In de-allocation cases, coalescing will occur if the memory block is the only block used in a memory pool. Sometimes, coalescing occurs between free memory pools, while in normal cases a memory block will just be released via setting a bit that indicates its state. Of course, additional operations spend more time. Practically, the worst case allocation and de-allocation scenarios spend more time allocating or de-allocating memory blocks, so we can determine the worst case execution time as well as measure the longest execution time for allocation and de-allocation.

This experiment is executed using different real applications with test sets 1 to 3. Each application was executed 1,000 times with the same allocator repeatedly; therefore, each application was executed 7,000 times with seven different allocators totally. In the individual execution, we measure the longest execution time of allocation and de-allocation in each application execution. For instance, in test set 1 of *cfrac* in Table 4.5, the application invokes the malloc() function 1,528 times. During 1,528 function calls, we collect the longest execution time, accumulating it as the application is executed 1,000 times. After that, we collect statistics of the

average of the thousand longest execution times, as well as the minimum and the maximum longest execution time in the thousand longest execution times.

***cfrac* Results** Table 4.11, Table 4.12 and Table 4.13 show execution times obtained for allocation and de-allocation on *cfrac* with each allocator. The execution times of test set 1 analysis is given in Table 4.11. The execution times of test set 2 is given in Table 4.12. The execution times of test set 3 is given in Table 4.13.

Results Analysis The following shows the impact of allocation and de-allocation on time; note that *Stdev.s()* in tables represents the standard deviation. As we mentioned in Section 4.2, we use processor cycles to measure execution time. Therefore, the unit of execution time is the number of processor cycles, but it can be converted to a time value by dividing 2010 (processor clock (MHz)). For example, the average time of *tcmalloc* for allocation in test set 1 (8,333,072.35) in table 4.11 is equal to 4.15 msec:

- Average time: The average longest execution time for allocation and de-allocation is shown in Table 4.11, Table 4.12 and Table 4.13, respectively, for all test sets. The average time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators. The average time of *tcmalloc* for allocation in test set 1 is 8,333,072.35 compared to 212,368.34 of *First-fit*, which shows the shortest344 average of the longest execution time. The average time of *tcmalloc* for allocation in test set 2 is 8,318,732.25 compared to 372,394.14 of *First-fit*, which shows the shortest average of the longest execution time. The average time of *tcmalloc* for allocation in test set 3 is 8,307,108.05 compared to 456,267.67 of *Best-fit*, which shows the shortest average of the longest execution time.

  In the de-allocation case, the average longest execution time of *Hoard* for de-allocation in test set 1 is longer than the values for other allocators. The average time of *nMART* for de-allocation in test sets 2 and 3 is longer than the values for other allocators. The average time of *Hoard* for de-allocation in test set 1 is 34,112.69 compared to 18,129.51 of *Best-fit*, which shows the shortest

average of the longest execution time. The average time of *nMART* for de-allocation in test set 2 is 133,066.01 compared to 48,122.62 of *First-fit*, which shows the shortest average of the longest execution time. The average time of *nMART* for de-allocation in test set 3 is 198,831.43 compared to 79,151.61 of *tcmalloc*, which shows the shortest average of the longest execution time.

- Min: The minimum longest execution time for allocation and de-allocation is shown in Table 4.11, Table 4.12 and Table 4.13, respectively, for all test sets. The minimum longest execution time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators. The minimum time of *tcmalloc* for allocation in test set 1 is 7,704,663 compared to 59,456 of *Half-fit*, which shows the shortest execution time. The minimum time of *tcmalloc* for allocation in test set 2 is 7,722,250 compared to 65,618 of *First-fit*, which shows the shortest execution time. The minimum time of *tcmalloc* for allocation in test set 3 is 7,719,497 compared to 109,404 of *First-fit*, which shows the shortest execution time.

  In the de-allocation case, the minimum longest execution time of *Hoard* for de-allocation in test sets 1 and 2 is longer than the values for other allocators. The minimum longest execution time of *tcmalloc* for de-allocation in test set 3 is longer than the values for other allocators. The minimum time of *Hoard* for de-allocation in test set 1 is 7,502 compared to 590 of *Half-fit*, which shows the shortest execution time. The minimum time of *Hoard* for de-allocation in test set 2 is 7,407 compared to 743 of *Half-fit*, which shows the shortest execution time. The minimum time of *tcmalloc* for de-allocation in test set 3 is 21,837 compared to 1,068 of *First-fit*, which shows the shortest execution time.

- Max: The maximum longest execution time for allocation and de-allocation is shown in Table 4.11, Table 4.12 and Table 4.13, respectively, for all test sets. The maximum longest execution time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators. The maximum time of *tcmalloc* for allocation in test set 1 is 9,778,916 compared to 768,668 of *First-fit*, which shows the shortest execution time. The maximum time of *tcmalloc*

for allocation in test set 2 is 9,123,311 compared to 1,030,918 of *nMART*, which shows the shortest execution time. The maximum time of *tcmalloc* for allocation in test set 3 is 9,423,513 compared to 1,199,017 of *nMART*, which shows the shortest execution time.

In the de-allocation case, the maximum longest execution time of *First-fit* for de-allocation in test set 1 is longer than the values for other allocators. The maximum longest execution time of *Best-fit* for de-allocation in test set 2 is longer than the values for other allocators. The maximum longest execution time of *Half-fit* for de-allocation in test set 3 is longer than the values for other allocators. The maximum time of *First-fit* for de-allocation in test set 1 is 873,224 compared to 385,717 of *nMART*, which shows the shortest execution time. The maximum time of *Best-fit* for de-allocation in test set 2 is 1,081,580 compared to 525,676 of *Hoard*, which shows the shortest execution time. The maximum time of *Half-fit* for de-allocation in test set 3 is 1,203,819 compared to 583,552 of *nMART*, which shows the shortest execution time.

cfrac: Test Set 1					
		Average time	Min	Max	Stdev.s()
Best	malloc	214,628.21	62,189	5,120,950	215,825.32
	free	18,129.51	791	492,877	68,650.87
First	malloc	212,368.34	61,479	768,668	155,153.45
	free	20,403.42	747	873,224	78,840.46
Half	malloc	220,548.67	59,456	974,629	164,180.54
	free	30,197.94	590	659,880	72,069.75
Hoard	malloc	3,340,860.87	2,807,576	4,426,567	467,996.55
	free	34,112.69	7,502	603,470	63,650.84
nMART	malloc	328,468.70	106,314	932,996	173,098.01
	free	32,936.93	816	385,717	89,598.65
tcmalloc	malloc	8,333,072.35	7,704,663	9,778,916	396,598.48
	free	29,139.50	6,637	571,560	54,936.97
TLSF	malloc	284,857.75	106,894	1,189,523	195,152.25
	free	29,634.66	625	653,955	79,929.18

Table 4.11: The average *malloc()/free()* time of Set 1 of *cfrac*

cfrac: Test Set 2					
		Average time	Min	Max	Stdev.s()
Best	malloc	375,499.07	66,987	1,213,794	175,446.25
	free	51,937.21	942	1,081,580	113,614.27
First	malloc	372,394.14	65,618	1,127,776	176,117.26
	free	48,122.62	930	561,142	106,420.45
Half	malloc	399,144.46	73,996	4,969,239	226,092.57
	free	87,924.02	743	792,488	129,416.55
Hoard	malloc	3,410,029.91	2,817,640	4,181,108	480,268.48
	free	59,236.17	7,407	525,676	98,130.32
nMART	malloc	565,514.03	271,949	1,030,918	214,086.68
	free	133,066.10	1,044	625,891	182,361.49
tcmalloc	malloc	8,318,732.25	7,722,250	9,123,311	397,159.83
	free	56,876.74	6,777	1,031,921	103,091.05
TLSF	malloc	448,946.68	112,614	1,346,470	240,276.95
	free	50,886.25	1,278	907,798	105,336.00

Table 4.12: The average *malloc()/free()* time of Set 2 of *cfrac*

cfrac: Test Set 3					
		Average time	Min	Max	Stdev.s()
Best	malloc	456,267.67	136,423	1,201,060	163,813.70
	free	110,602.86	1,128	1,167,753	151,492.62
First	malloc	470,091.62	109,404	5,185,056	224,624.07
	free	99,639.90	1,068	1,090,271	141,349.51
Half	malloc	473,908.76	136,420	1,263,521	170,373.70
	free	135,897.87	19,987	1,203,819	165,655.78
Hoard	malloc	3,365,642.86	2,779,007	4,523,809	473,715.43
	free	86,592.17	7,728	1,143,234	132,026.54
nMART	malloc	633,478.87	367,286	1,199,017	178,467.04
	free	198,831.43	19,857	583,552	216,111.72
tcmalloc	malloc	8,307,108.05	7,719,497	9,423,513	402,948.43
	free	79,151.61	21,837	603,932	100,009.80
TLSF	malloc	570,979.39	139,536	1,683,677	240,433.16
	free	85,303.66	1,225	639,988	139,356.43

Table 4.13: The average *malloc()/free()* time of Set 3 of *cfrac*

***espresso* Results** Table 4.14, Table 4.15 and Table 4.16 show execution times obtained for allocation and de-allocation on *espresso* with each allocator. The execution times of test set 1 analysis is given in Table 4.14. The execution times of test set 2 is given in Table 4.15. The execution times of test set 3 is given in Table 4.16.

Results Analysis The following shows the effect of allocation and de-allocation on time. The unit is the number of processor cycles:

- Average time: The average longest time for allocation and de-allocation is shown in Table 4.14, Table 4.15 and Table 4.16, respectively, for all test sets. The average time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators. The average time of *tcmalloc* for allocation in test set 1 is 8,502,336.53 compared to 287,067.58 of *First-fit*, which shows the shortest average of the longest execution time. The average time of *tcmalloc* for allocation in test set 2 is 8,353,023.89 compared to 376,699.75 of *First-fit*, which shows the shortest average of the longest execution time. The average time of *tcmalloc* for allocation in test set 3 is 8,357,972.13 compared to 708,740.49 of *First-fit*, which shows the shortest average of the longest execution time.

  In the de-allocation case, the average longest execution time of *Half-fit* for de-allocation throughout all tests is longer than the values for other allocators, while the average longest execution time of *tcmalloc* for de-allocation throughout all tests is shorter than the values for other allocators. The average time of *Hoard* for de-allocation in test set 1 is 281,605.67 compared to 202,217.10 of *tcmalloc*, which shows the minimum average of the longest execution time. The average time of *Half-fit* for de-allocation in test set 2 is 398,143.54 compared to 324,335.71 of *tcmalloc*, which shows the minimum average of the longest execution time. The average time of *Half-fit* for de-allocation in test set 3 is 775,582.14 compared to 654.561.72 of *tcmalloc*, which shows the minimum average of the longest execution time.

- Min: The minimum longest execution time for allocation and de-allocation is shown in Table 4.14, Table 4.15 and Table 4.16, respectively, for all test sets.

The minimum longest execution time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators, while *First-fit* shows the shortest of the longest execution time for de-allocation throughout all test sets. The minimum time of *tcmalloc* for allocation in test set 1 is 7,733,015 compared to 49,839 of *First-fit*, which shows the shortest execution time. The minimum time of *tcmalloc* for allocation in test set 2 is 7,715,054 compared to 51,002 of *First-fit*, which shows the shortest execution time. The minimum time of *tcmalloc* for allocation in test set 3 is 7,762,354 compared to 432,900 of *First-fit*, which shows the shortest execution time.

In the de-allocation case, the minimum longest execution time of *tcmalloc* for de-allocation in test sets 1 and 2 is longer than the values for other allocators. The minimum longest execution time of *Half-fit* for de-allocation in test set 3 is longer than the values for other allocators. The minimum time of *tcmalloc* for de-allocation in test set 1 is 17,688 compared to 926 of *Best-fit*, which shows the shortest execution time. The minimum time of *tcmalloc* for de-allocation in test set 2 is 19,772 compared to 2,500 of *First-fit*, which shows the shortest execution time. The minimum time of *Half-fit* for de-allocation in test set 3 is 447,165 compared to 379,366 of *tcmalloc*, which shows the shortest execution time.

- Max: The maximum longest execution time for allocation and de-allocation is shown in Table 4.14, Table 4.15 and Table 4.16, respectively, for all test sets. The maximum longest execution time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators. The maximum time of *tcmalloc* for allocation in test set 1 is 9,658,963 compared to 641,681 of *nMART*, which shows the shortest execution time. The maximum time of *tcmalloc* for allocation in test set 2 is 9,564,810 compared to 780,534 of *nMART*, which shows the shortest execution time. The maximum time of *tcmalloc* for allocation in test set 3 is 14,656,388 compared to 4,973,663 of *nMART*, which shows the shortest execution time.

  In the de-allocation case, the maximum longest execution time of *First-fit* for

de-allocation in test set 1 is longer than the values for other allocators. The maximum longest execution time of *TLSF* for de-allocation in test set 2 is longer than the values for other allocators. The maximum longest execution time of *tcmalloc* for de-allocation in test set 3 is longer than the values for other allocators. The maximum time of *First-fit* for de-allocation in test set 1 is 4,987,469 compared to 554,217 of *nMART*, which shows the shortest execution time. The maximum time of *TLSF* for de-allocation in test set 2 is 4,944,250 compared to 617,614 of *nMART*, which shows the shortest execution time. The maximum time of *tcmalloc* for de-allocation in test set 3 is 5,327,664 compared to 1,284,990 of *nMART*, which shows the shortest execution time.

espresso: Test Set 1					
		Average time	Min	Max	Stdev.s()
Best	malloc	316,893.83	52,417	1,357,885	177,761.96
	free	237,349.59	926	1,346,874	208,581.09
First	malloc	287,069.58	49,839	1,313,421	185,084.73
	free	226,211.48	1,345	4,987,469	254,061.81
Half	malloc	342,056.87	51,578	1,186,206	170,846.11
	free	281,605.67	1,973	1,129,344	190,141.45
Hoard	malloc	3,354,796.59	2,805,003	4,270,118	475,756.75
	free	220,609.53	5,222	1,105,986	211,036.72
nMART	malloc	279,446.70	90,360	641,681	168,623.42
	free	253,164.17	2,104	554,217	204,669.98
tcmalloc	malloc	8,502,336.53	7,733,015	9,658,963	322,049.44
	free	202,217.10	17,688	1,099,139	228,887.61
TLSF	malloc	306,888.86	73,707	1,318,821	211,380.87
	free	213,621.84	7,055	1,050,580	211,705.12

Table 4.14: The average *malloc()/free()* time of Set 1 of *espresso*

gawk **Results** Table 4.17, Table 4.18 and Table 4.19 show execution times obtained for allocation and de-allocation on *gawk* with each allocator. The execution time of test set 1 analysis is given in Table 4.17. The execution times of test set 2 is given in Table 4.18. The execution times of test set 3 is given in Table 4.19.

espresso: Test Set 2					
		Average time	Min	Max	Stdev.s()
Best	malloc	425,596.04	55,163	1,189,763	140,490.61
	free	349,901.25	2,681	1,121,580	160,051.01
First	malloc	376,699.75	51,002	1,080,488	156,193.87
	free	353,697.72	2,500	1,688,041	180,930.93
Half	malloc	434,869.05	53,786	1,277,975	139,581.64
	free	398,143.54	3,682	1,209,649	149,553.36
Hoard	malloc	3,567,896.06	2,790,053	5,281,388	468,266.79
	free	348,448.14	5,372	1,501,541	188,430.56
nMART	malloc	383,483.43	95,733	780,534	132,510.72
	free	333,060.00	3,277	617,614	195,047.00
tcmalloc	malloc	8,353,023.89	7,715,054	9,564,810	396,285.24
	free	324,335.71	19,772	1,400,689	220,596.95
TLSF	malloc	390,315.28	82,438	1,274,815	165,170.34
	free	355,818.81	7,890	4,944,250	232,909.88

Table 4.15: The average *malloc()/free()* time of Set 2 of *espresso*

espresso: Test Set 3					
		Average time	Min	Max	Stdev.s()
Best	malloc	866,891.28	466,864	5,160,904	502,923.82
	free	702,400.54	429,080	4,926,442	279,784.84
First	malloc	708,740.49	432,900	5,079,030	313,798.56
	free	695,962.75	399,317	5,293,355	312,095.34
Half	malloc	851,022.24	471,749	5,073,612	427,797.40
	free	775,582.14	447,165	5,124,217	391,869.31
Hoard	malloc	3,253,162.83	2,763,749	5,024,183	471,538.76
	free	731,693.56	425,342	5,218,295	378,408.28
nMART	malloc	861,857.73	495,565	4,973,663	808,102.26
	free	744,368.17	451,469	1,284,990	254,618.95
tcmalloc	malloc	8,357,972.13	7,762,354	14,656,388	435,706.87
	free	654,561.72	379,366	5,327,664	335,666.84
TLSF	malloc	781,175.95	455,932	5,046,530	399,757.62
	free	730,979.13	426,102	5,093,244	447,688.98

Table 4.16: The average *malloc()/free()* time of Set 3 of *espresso*

Results Analysis The following shows the effect of allocation and de-allocation on time. The unit is the number of processor cycles:

- Average time: The average longest time for allocation and de-allocation is shown in Table 4.17, Table 4.18 and Table 4.19, respectively, for all test sets. The average time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators. The average time of *tcmalloc* for allocation in test set 1 is 8,578,420.04 compared to 299,584.07 of *First-fit*, which shows the minimum average of the longest execution time. The average time of *tcmalloc* for allocation in test set 2 is 11,741,468.32 compared to 676,179.63 of *Half-fit*, which shows the minimum average of the longest execution time. The average time of *tcmalloc* for allocation in test set 3 is 11,939,380.85 compared to 760,253.67 of *First-fit*, which shows the minimum average of the longest execution time.

  In the de-allocation case, the average time of *Hoard* for de-allocation throughout test sets 1 and 2 is longer than the values for other allocators. The average time of *TLSF* for de-allocation throughout test set 3 is longer than the values for other allocators. The average time of *Hoard* for de-allocation in test set 1 is 91,364.24 compared to 38,383.73 of *tcmalloc*, which shows the minimum average of the longest execution time. The average time of *Hoard* for de-allocation in test set 2 is 542,598.87 compared to 441,309.22 of *First-fit*, which shows the minimum average of the longest execution time. The average time of *TLSF* for de-allocation in test set 3 is 643,570.56 compared to 528,380.67 of *Best-fit*, which shows the minimum average of the longest execution time.

- Min: The minimum longest execution time for allocation and de-allocation is shown in Table 4.17, Table 4.18 and Table 4.19, respectively, for all test sets. The minimum longest execution time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators. The minimum time of *tcmalloc* for allocation in test set 1 is 7,760,483 compared to 61,677 of *Best-fit*, which shows the shortest execution time. The minimum time of *tcmalloc* for allocation in test set 2 is 10,786,211 compared to 356,474 of *Half-fit*, which

shows the shortest execution time. The minimum time of *tcmalloc* for allocation in test set 3 is 10,792,026 compared to 494,305 of *First-fit*, which shows the shortest execution time.

In the de-allocation case, the minimum longest execution time of *Hoard* for de-allocation in test sets 1 and 2 is longer than the values for other allocators. The minimum longest execution time of *nMART* for de-allocation in test set 3 is longer than the values for other allocators. The minimum time of *Hoard* for de-allocation in test set 1 is 34,110 compared to 559 of *TLSF*, which shows the shortest execution time. The minimum time of *Hoard* for de-allocation in test set 2 is 235,306 compared to 2,715 of *TLSF*, which shows the shortest execution time. The minimum time of *nMART* for de-allocation in test set 3 is 383,311 compared to 19,738 of *Best-fit*, which shows the shortest execution time.

- Max: The maximum longest execution time for allocation and de-allocation is shown in Table 4.17, Table 4.18 and Table 4.19, respectively, for all test sets. The maximum longest execution time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators. The maximum time of *tcmalloc* for allocation in test set 1 is 9,612,900 compared to 1,062,006 of *nMART*, which shows the shortest execution time. The maximum time of *tcmalloc* for allocation in test set 2 is 31,532,613 compared to 1,530,469 of *nMART*, which shows the shortest execution time. The maximum time of *tcmalloc* for allocation in test set 3 is 37,213,136 compared to 2,192,928 of *nMART*, which shows the shortest execution time.

  In the de-allocation case, the maximum longest execution time of *Best-fit* for de-allocation in test set 1 is longer than the values for other allocators. The maximum longest execution time of *Hoard* for de-allocation in test set 2 is longer than the values for other allocators. The maximum longest execution time of *Half-fit* for de-allocation in test set 3 is longer than the values for other allocators. The maximum time of *Best-fit* for de-allocation in test set 1 is 1,136,916 compared to 561,612 of *nMART*, which shows the shortest execution

time. The maximum time of *Hoard* for de-allocation in test set 2 is 5,162,480 compared to 630,654 of *nMART*, which shows the shortest execution time. The maximum time of *Half-fit* for de-allocation in test set 3 is 5,202,690 compared to 1,042,077 of *nMART*, which shows the shortest execution time.

gawk: Test Set 1					
		Average time	Min	Max	Stdev.s()
Best	malloc	306,953.30	61,677	1,163,320	169,140.66
	free	53,577.29	3,569	1,136,916	115,949.07
First	malloc	299,584.07	69,820	1,141,155	168,365.80
	free	50,509.65	3,576	660,294	106,255.84
Half	malloc	319,088.10	67,220	5,001,510	226,514.10
	free	66,156.24	3,118	1,123,960	142,402.00
Hoard	malloc	3,398,034.02	2,786,993	4,324,692	475,182.68
	free	91,364.24	34,110	1,012,183	120,372.42
nMART	malloc	478,174.90	198,038	1,062,006	213,122.78
	free	38,852.23	607	561,612	120,575.34
tcmalloc	malloc	8,578,420.04	7,760,483	9,612,900	273,143.78
	free	38,383.73	6,208	733,301	103,279.23
TLSF	malloc	413,712.61	126,014	1,294,939	224,968.66
	free	41,143.60	559	750,538	127,552.98

Table 4.17: The average *malloc()*/*free()* time of Set 1 of *gawk*

p2c **Results** Table 4.20, Table 4.21 and Table 4.22 show execution times obtained for allocation and de-allocation on *p2c* with each allocator. The execution time of test set 1 analysis is given in Table 4.20. The execution times of test set 2 is given in Table 4.21. The execution times of test set 3 is given in Table 4.22.

Results Analysis The following shows the effect of allocation and de-allocation on time. The unit is the number of processor cycles:

- Average time: The average longest times for allocation and de-allocation is shown in Table 4.20, Table 4.21 and Table 4.22, respectively, for all test sets. The average time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators. The average time of *tcmalloc* for allocation

gawk: Test Set 2					
		Average time	Min	Max	Stdev.s()
Best	malloc	680,430.67	359,788	5,221,906	200,719.17
	free	446,963.64	4,333	5,070,667	204,261.50
First	malloc	691,628.79	392,423	5,216,926	237,014.75
	free	441,309.22	19,793	1,243,975	155,979.47
Half	malloc	676,179.63	356,474	5,284,900	200,245.38
	free	475,240.41	3,885	4,932,226	185,967.19
Hoard	malloc	3,363,651.74	2,803,385	4,339,793	461,334.05
	free	542,598.87	235,306	5,162,480	268,426.83
nMART	malloc	934,034.77	594,950	1,530,469	194,351.11
	free	465,438.33	241,495	630,654	83,076.73
tcmalloc	malloc	11,731,468.32	10,786,211	31,532,613	920,187.02
	free	483,435.10	22,949	5,100,050	219,520.47
TLSF	malloc	929,580.24	463,123	1,924,966	211,810.62
	free	517,904.13	2,715	1,174,107	136,193.51

Table 4.18: The average *malloc()/free()* time of Set 2 of *gawk*

gawk: Test Set 3					
		Average time	Min	Max	Stdev.s()
Best	malloc	785,513.37	503,558	5,234,231	259,795.24
	free	528,380.67	19,738	1,267,111	136,384.23
First	malloc	760,253.67	494,305	5,242,546	277,609.92
	free	536,186.47	19,873	5,095,696	191,572.73
Half	malloc	767,245.01	496,771	5,073,434	212,932.14
	free	569,403.67	250,628	5,202,690	249,462.83
Hoard	malloc	3,354,691.81	2,784,054	4,972,763	474,551.79
	free	623,393.88	269,168	1,381,371	182,584.12
nMART	malloc	1,261,376.33	931,757	2,192,928	278,863.45
	free	607,860.87	383,311	1,042,077	164,418.96
tcmalloc	malloc	11,939,380.85	10,792,026	37,213,136	1,602,598.31
	free	601,423.14	29,149	1,627,890	159,608.13
TLSF	malloc	1,349,688.49	847,136	5,175,018	385,390.41
	free	643,570.56	371,705	5,120,053	199,556.89

Table 4.19: The average *malloc()/free()* time of Set 3 of *gawk*

in test set 1 is 8,348,306.46 compared to 296,586.89 of *Best-fit*, which shows the minimum average of the longest execution time. The average time of *tcmalloc* for allocation in test set 2 is 8,367,276.31 compared to 404,693.67 of *TLSF*, which shows the minimum average of the longest execution time. The average time of *tcmalloc* for allocation in test set 3 is 8,375,375.20 compared to 632,419.60 of *First-fit*, which shows the minimum average of the longest execution time.

In the de-allocation case, the average longest execution time of *Hoard* for de-allocation throughout test set 1 is longer than the values for other allocators. The average longest execution time of *Half-fit* for de-allocation throughout test sets 2 and 3 is longer than the values for other allocators. The average time of *Hoard* for de-allocation in test set 1 is 36,704.45 compared to 14,255.77 of *nMART*, which shows the minimum average of the longest execution time. The average time of *Half-fit* for de-allocation in test set 2 is 117,247.01 compared to 34,729.50 of *nMART*, which shows the minimum average of the longest execution time. The average time of *Half-fit* for de-allocation in test set 3 is 371,065.01 compared to 276,477.62 of *tcmalloc*, which shows the minimum average of the longest execution time.

- Min: The minimum longest execution time for allocation and de-allocation is shown in Table 4.20, Table 4.21 and Table 4.22, respectively, for all test sets. The minimum longest execution time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators. The minimum time of *tcmalloc* for allocation in test set 1 is 7,676,006 compared to 47,640 of *First-fit*, which shows the shortest execution time. The minimum time of *tcmalloc* for allocation in test set 2 is 7,749,307 compared to 49,848 of *First-fit*, which shows the shortest execution time. The minimum time of *tcmalloc* for allocation in test set 3 is 7,723,173 compared to 103,537 of *TLSF*, which shows the shortest execution time.

  In the de-allocation case, the minimum longest execution time of *Hoard* for de-allocation in test sets 1 and 2 is longer than the values for other allocators.

The minimum longest execution time of *tcmalloc* for de-allocation in test set 3 is longer than the values for other allocators. The minimum time of *Hoard* for de-allocation in test set 1 is 11,481 compared to 921 of *TLSF*, which shows the shortest execution time. The minimum time of *Hoard* for de-allocation in test set 2 is 11,732 compared to 922 of *TLSF*, which shows the shortest execution time. The minimum time of *tcmalloc* for de-allocation in test set 3 is 27,142 compared to 12,084 of *Hoard*, which shows the shortest execution time.

- Max: The maximum longest execution time for allocation and de-allocation is shown in Table 4.20, Table 4.21 and Table 4.22, respectively, for all test sets. The maximum longest execution time of *tcmalloc* for allocation throughout all tests is longer than the values for other allocators. The maximum time of *tcmalloc* for allocation in test set 1 is 9,674,195 compared to 1,057,949 of *First-fit*, which shows the shortest execution time. The maximum time of *tcmalloc* for allocation in test set 2 is 9,345,380 compared to 914,276 of *nMART*, which shows the shortest execution time. The maximum time of *tcmalloc* for allocation in test set 3 is 10,233,936 compared to 1,107,103 of *nMART*, which shows the shortest execution time.

  In the de-allocation case, the maximum longest execution time of *TLSF* for de-allocation in test sets 1 and 3 is longer than the values for other allocators. The maximum longest execution time of *Half-fit* for de-allocation in test set 2 is longer than the values for other allocators. The maximum time of *TLSF* for de-allocation in test set 1 is 1,060,541 compared to 361,204 of *nMART*, which shows the shortest execution time. The maximum time of *Half-fit* for de-allocation in test set 2 is 5,114,481 compared to 502,192 of *nMART*, which shows the shortest execution time. The maximum time of *TLSF* for de-allocation in test set 3 is 5,205,056 compared to 703,563 of *nMART*, which shows the shortest execution time.

In summary, a type of memory allocator, which uses a per-thread heap with multiple size classes of small sizes of free blocks, usually shows worse performance in the worst case scenarios than a type of memory allocator using a global heap

p2c: Test Set 1					
		Average time	Min	Max	Stdev.s()
Best	malloc	296,586.89	48,093	1,143,657	232,264.09
	free	18,908.12	2,332	656,752	85,218.86
First	malloc	323,072.56	47,640	1,057,949	229,765.80
	free	17,987.45	2,319	675,839	82,684.92
Half	malloc	353,847.11	49,727	1,273,113	229,273.95
	free	22,448.45	1,007	683,932	56,582.74
Hoard	malloc	3,330,697.84	2,752,022	4,705,648	469,676.24
	free	36,704.45	11,481	732,691	94,437.62
nMART	malloc	400,715.37	95,749	1,069,550	278,448.26
	free	14,255.77	1,289	361,204	65,615.55
tcmalloc	malloc	8,348,306.46	7,676,006	9,674,195	392,031.99
	free	31,716.12	6,620	693,072	99,092.67
TLSF	malloc	309,124.02	86,378	1,255,863	243,522.17
	free	28,895.32	921	1,060,541	122,478.76

Table 4.20: The average *malloc()/free()* time of Set 1 of *p2c*

p2c: Test Set 2					
		Average time	Min	Max	Stdev.s()
Best	malloc	429,851.42	53,694	1,248,437	219,375.56
	free	61,592.67	2,278	783,132	149,519.53
First	malloc	434,348.29	49,848	1,168,484	232,081.91
	free	59,414.01	2,383	1,067,850	152,547.75
Half	malloc	485,641.41	53,530	1,268,345	208,298.36
	free	117,247.01	1,307	5,114,481	258,111.10
Hoard	malloc	3,304,142.17	2,769,887	4,324,017	458,917.05
	free	74,201.62	11,732	1,128,328	158,403.59
nMART	malloc	529,538.10	151,604	914,276	233,210.28
	free	34,729.50	1,267	502,192	88,432.83
tcmalloc	malloc	8,367,276.31	7,749,307	9,345,380	379,348.60
	free	67,964.26	6,767	998,013	158,403.55
TLSF	malloc	404,693.67	92,683	1,210,880	252,325.06
	free	67,885.66	922	875,079	172,547.88

Table 4.21: The average *malloc()/free()* time of Set 2 of *p2c*

p2c: Test Set 3					
		Average time	Min	Max	Stdev.s()
Best	malloc	680,409.91	388,870	5,000,315	211,483.50
	free	321,406.10	19,808	4,970,134	263,644.65
First	malloc	632,419.60	144,270	1,259,328	148,124.80
	free	307,941.78	19,831	1,193,373	212,794.77
Half	malloc	652,864.04	159,291	1,222,243	145,474.44
	free	371,065.10	19,983	1,179,346	195,712.12
Hoard	malloc	3,295,687.61	2,766,864	4,285,355	447,843.58
	free	303,209.61	12,084	1,221,773	229,451.99
nMART	malloc	749,627.47	374,411	1,107,103	172,329.75
	free	340,936.80	20,321	703,563	194,154.58
tcmalloc	malloc	8,375,375.20	7,723,173	10,233,936	388,132.71
	free	276,477.62	27,142	1,192,790	223,185.43
TLSF	malloc	664,806.50	103,537	5,067,908	265,226.95
	free	368,255.21	19,726	5,205,056	296,480.66

Table 4.22: The average *malloc()/free()* time of Set 3 of *p2c*

because they perform more operations to maintain multiple heaps and the various small size classes. They also prepare some space for pre-defined size of classes at the first execution of the algorithms. For these reasons, overall, *tcmalloc* and *Hoard* show worse performance throughout all test sets. *nMART* avoids the overheads by allocating a 4k block (as a per-thread heap) and splitting any allocation requests to small sizes when required.

4.3.2 Total Execution Time

As with the above experiments, this experiment is executed using different real applications with test sets 1 to 3. Each application was executed 1,000 times with the same allocator repeatedly; therefore, each application was executed 7,000 times with seven different allocators in total. In the individual execution, we measure the total execution time of allocation and de-allocation in each application execution. For instance, in test set 1 of *cfrac* in Table 4.5, the application invokes the malloc() function 1,528 times. During 1,528 function calls, we collect each execution time for

allocation and de-allocation practically, accumulating it for the whole application lifetime. After that, we collect statistics of the average of a thousand execution times spent, as well as the minimum and the maximum total execution time in a thousand total execution times as the application is executed 1,000 times.

***cfrac* Results** Table 4.23, Table 4.24 and Table 4.25 show measured execution times for allocation and de-allocation on *cfrac* with each allocator. The execution times of test set 1 analysis is given in Table 4.23. The execution time of test set 2 is given in Table 4.24. The execution times of test set 3 is given in Table 4.25.

Results Analysis The following shows the effect of allocation and de-allocation on time. The unit is the number of processor cycles:

- Average time: The average total execution time for allocation and de-allocation is shown in Table 4.23, Table 4.24 and Table 4.25, respectively, for all test sets. The average time of *tcmalloc* for allocation throughout test sets 1 and 2 is longer than the values for other allocators. The average time of *Half-fit* for allocation in test set 3 is longer than the values for other allocators. The average time of *tcmalloc* for allocation in test set 1 is 9,068,004.11 compared to 1,362,989.48 of *TLSF*, which shows the minimum average of the total execution time. The average time of *tcmalloc* for allocation in test set 2 is 11,056,006.17 compared to 4,235,375.31 of *TLSF*, which shows the minimum average of the total execution time. The average time of *Half-fit* for allocation in test set 3 is 23,641,909.73 compared to 6,355,733.38 of *Hoard*, which shows the minimum average of the total execution time.

  In the de-allocation case, the average total execution time of *Half-fit* for de-allocation throughout all test sets is longer than the values for other allocators, while the average total execution time of *tcmalloc* for de-allocation throughout all test sets is shorter than the values for other allocators. The average time of *Half-fit* for de-allocation in test set 1 is 465,183.38 compared to 278,470.79 of *tcmalloc*, which shows the minimum average of the total execution time. The average time of *Half-fit* for de-allocation in test set 2 is 1,891,632.57

compared to 883,677.97 of *tcmalloc*, which shows the minimum average of the total execution time. The average time of *Half-fit* for de-allocation in test set 3 is 3,318,209.85 compared to 1,422,679.27 of *tcmalloc*, which shows the minimum average of the total execution time.

- Min: The minimum total execution time for allocation and de-allocation is shown in Table 4.23, Table 4.24 and Table 4.25, respectively, for all test sets. The minimum total execution time of *tcmalloc* for allocation throughout test sets 1 and 2 is longer than the values for other allocators. The minimum total execution time of *Half-fit* for allocation throughout test set 3 is longer than the values for other allocators. The minimum time of *tcmalloc* for allocation in test set 1 is 8,313,746 compared to 981,488 of *TLSF*, which shows the shortest execution time. The minimum time of *tcmalloc* for allocation in test set 2 is 9,951,086 compared to 3,178,937 of *TLSF*, which shows the shortest execution time. The minimum time of *Half-fit* for allocation in test set 3 is 21,672,745 compared to 5,602,864 of *Hoard*, which shows the shortest execution time.

  In the de-allocation case, the minimum total execution time of *Half-fit* for de-allocation throughout all test sets is longer than the values for other allocators, while the average total execution time of *tcmalloc* for de-allocation throughout all test sets is shorter than the values for other allocators. The minimum time of *Half-fit* for de-allocation in test set 1 is 422,931 compared to 226,191 of *tcmalloc*, which shows the shortest execution time. The minimum time of *Half-fit* for de-allocation in test set 2 is 1,745,561 compared to 786,504 of *tcmalloc*, which shows the shortest execution time. The minimum time of *Half-fit* for de-allocation in test set 3 is 3,056,451 compared to 1,338,329 of *tcmalloc*, which shows the shortest execution time.

- Max: The maximum total execution time for allocation and de-allocation is shown in Table 4.23, Table 4.24 and Table 4.25, respectively, for all test sets. The maximum total execution time of *tcmalloc* for allocation throughout test set 1 is longer than the values for other allocators. The maximum total execution time of *Best-fit* for allocation throughout test set 2 is longer than the

values for other allocators. The maximum total execution time of *Half-fit* for allocation throughout test set 3 is longer than the values for other allocators. The maximum time of *tcmalloc* for allocation in test set 1 is 10,601,363 compared to 2,863,908 of *TLSF*, which shows the shortest execution time. The maximum time of *Best-fit* for allocation in test set 2 is 18,812,924 compared to 6,496,255 of *Hoard*, which shows the shortest execution time. The maximum time of *Half-fit* for allocation in test set 3 is 29,816,957 compared to 8,334,062 of *Hoard*, which shows the shortest execution time.

In the de-allocation case, the maximum total execution time of *First-fit* for de-allocation in test set 1 is longer than the values for other allocators. The maximum total execution time of *Half-fit* for de-allocation in test sets 2 and 3 is longer than the values for other allocators. The maximum time of *First-fit* for de-allocation in test set 1 is 1,216,252 compared to 723,791 of *nMART*, which shows the shortest execution time. The maximum time of *Half-fit* for de-allocation in test set 2 is 2,634,833 compared to 1,850,802 of *Hoard*, which shows the shortest execution time. The maximum time of *Half-fit* for de-allocation in test set 3 is 4,581,187 compared to 1,948,382 of *tcmalloc*, which shows the shortest execution time.

***espresso* Results** Table 4.26, Table 4.27 and Table 4.28 show measured execution times for allocation and de-allocation on *espresso* with each allocator. The execution time of test set 1 analysis is given in Table 4.26. The execution times of test set 2 is given in Table 4.27. The execution times of test set 3 is given in Table 4.28.

Results Analysis The following shows the effect of allocation and de-allocation on time. The unit is the number of processor cycles:

- Average time: The average total execution time for allocation and de-allocation is shown in Table 4.26, Table 4.27 and Table 4.28, respectively, for all test sets. The average time of *tcmalloc* for allocation throughout test set 1 is longer than the values for other allocators. The average time of *Half-fit* for allocation in

cfrac: Test Set 1					
		Average time	Min	Max	Stdev.s()
Best	malloc	2,219,635.33	1,824,604	8,168,481	452,869.46
	free	336,111.18	312,963	825,239	70,182.49
First	malloc	2,193,971.88	1,773,319	8,735,528	434,222.92
	free	338,530.64	313,589	1,216,252	80,779.73
Half	malloc	2,354,737.36	1,940,285	8,915,013	415,209.69
	free	465,183.38	422,931	1,093,790	77,064.64
Hoard	malloc	3,921,714.99	3,340,193	5,010,571	460,134.45
	free	343,168.98	260,694	963,845	76,565.61
nMART	malloc	2,282,401.37	1,793,824	3,110,846	371,477.45
	free	356,567.87	315,572	723,791	97,805.53
tcmalloc	malloc	9,068,004.11	8,313,746	10,601,363	446,531.90
	free	278,470.79	226,191	820,776	58,585.40
TLSF	malloc	1,362,989.48	981,488	2,863,908	326,534.16
	free	304,354.39	265,916	945,942	83,957.60

Table 4.23: The average of total *malloc()/free()* time for *cfrac* test set 1

cfrac: Test Set 2					
		Average time	Min	Max	Stdev.s()
Best	malloc	10,112,049.11	8,946,018	18,812,924	851,785.41
	free	1,337,827.13	1,264,884	2,354,727	121,555.36
First	malloc	10,005,855.94	8,836,405	18,081,181	864,049.31
	free	1,332,742.83	1,265,523	1,900,858	115,795.34
Half	malloc	10,640,629.54	9,491,472	18,337,499	852,773.19
	free	1,891,632.57	1,745,561	2,634,833	140,133.28
Hoard	malloc	5,222,969.72	4,532,544	6,496,255	470,055.34
	free	1,240,848.76	1,020,360	1,850,802	166,613.59
nMART	malloc	5,719,923.60	5,448,601	7,008,062	564,250.27
	free	1,615,873.70	1,309,851	2,120,325	232,490.95
tcmalloc	malloc	11,056,006.17	9,951,086	13,301,670	607,346.38
	free	883,677.97	786,504	1,853,460	108,585.57
TLSF	malloc	4,235,375.31	3,178,937	7,217,319	565,490.01
	free	1,096,864.76	1,029,125	1,970,071	115,199.39

Table 4.24: The average of total *malloc()/free()* time for *cfrac* test set 2

cfrac: Test Set 3					
		Average time	Min	Max	Stdev.s()
Best	malloc	22,537,314.81	20,812,953	28,194,513	1,044,212.73
	free	2,386,671.52	2,215,621	3,455,138	170,016.09
First	malloc	22,432,607.76	20,483,759	28,408,566	1,083,020.43
	free	2,365,372.65	2,212,800	3,396,221	160,552.25
Half	malloc	23,651,909.73	21,672,745	29,816,957	1,058,659.01
	free	3,318,209.85	3,056,451	4,581,187	196,463.43
Hoard	malloc	6,355,733.38	5,602,864	8,334,062	519,346.51
	free	1,938,006.30	1,747,350	3,280,756	212,613.66
nMART	malloc	8,938,137.73	6,632,436	13,391,493	763,575.67
	free	3,187,592.87	2,703,907	3,985,800	297,058.23
tcmalloc	malloc	12,844,142.80	11,490,354	15,389,955	704,011.56
	free	1,422,679.27	1,338,329	1,948,382	113,484.78
TLSF	malloc	7,227,047.45	5,683,482	12,468,744	798,793.16
	free	1,929,864.22	1,785,515	3,000,045	170,032.99

Table 4.25: The average of total *malloc()/free()* time for *cfrac* test set 3

test sets 2 and 3 is longer than the values for other allocators. The average time of *tcmalloc* for allocation in test set 1 is 14,670,130.96 compared to 6,549,822.43 of *TLSF*, which shows the minimum average of the total execution time. The average time of *Half-fit* for allocation in test set 2 is 30,834,373.92 compared to 15,156,754.04 of *TLSF*, which shows the minimum average of the total execution time. The average time of *Half-fit* for allocation in test set 3 is 862,967,947.65 compared to 302,131,117.47 of *tcmalloc*, which shows the minimum average of the total execution time.

In the de-allocation case, the average total execution time of *Half-fit* for de-allocation throughout all test sets is longer than the values for other allocators, while the average total execution time of *tcmalloc* for de-allocation throughout all test sets is shorter than the values for other allocators. The average time of *Half-fit* for de-allocation in test set 1 is 9,015,318.13 compared to 4,182,505.25 of *tcmalloc*, which shows the minimum average of the total execution time. The average time of *Half-fit* for de-allocation in test set 2 is 22,783,161.78

compared to 9,886,048.68 of *tcmalloc*, which shows the minimum average of the total execution time. The average time of *Half-fit* for de-allocation in test set 3 is 410,607,470.70 compared to 267,100,185.80 of *tcmalloc*, which shows the minimum average of the total execution time.

- Min: The minimum total execution time for allocation and de-allocation is shown in Table 4.26, Table 4.27 and Table 4.28, respectively, for all test sets. The minimum total execution time of *tcmalloc* for allocation throughout test set 1 is longer than the values for other allocators. The minimum total execution time of *Half-fit* for allocation throughout test sets 2 and 3 is longer than the values for other allocators. The minimum time of *tcmalloc* for allocation in test set 1 is 13,172,530 compared to 5,991,619 of *TLSF*, which shows the shortest execution time. The minimum time of *Half-fit* for allocation in test set 2 is 29,339,010 compared to 14,201,387 of *TLSF*, which shows the shortest execution time. The minimum time of *Half-fit* for allocation in test set 3 is 839,115,005 compared to 293,516,411 of *tcmalloc*, which shows the shortest execution time.

  In the de-allocation case, the minimum total execution time of *Half-fit* for de-allocation throughout all test sets is longer than the values for other allocators, while the average total execution time of *tcmalloc* for de-allocation throughout all test sets is shorter than the values for other allocators. The minimum time of *Half-fit* for de-allocation in test set 1 is 8,509,403 compared to 3,907,238 of *tcmalloc*, which shows the shortest execution time. The minimum time of *Half-fit* for de-allocation in test set 2 is 21,737,415 compared to 9,286,535 of *tcmalloc*, which shows the shortest execution time. The minimum time of *Half-fit* for de-allocation in test set 3 is 602,756,763 compared to 260,335,795 of *tcmalloc*, which shows the shortest execution time.

- Max: The maximum total execution time for allocation and de-allocation is shown in Table 4.26, Table 4.27 and Table 4.28, respectively, for all test sets. The maximum total execution time of *tcmalloc* for allocation throughout test set 1 is longer than the values for other allocators. The maximum total execu-

tion time of *Half-fit* for allocation throughout test sets 2 and 3 is longer than the values for other allocators. The maximum time of *tcmalloc* for allocation in test set 1 is 17,586,676 compared to 8,281,692 of *nMART*, which shows the shortest execution time. The maximum time of *Half-fit* for allocation in test set 2 is 33,140,180 compared to 16,870,230 of *TLSF*, which shows the shortest execution time. The maximum time of *Half-fit* for allocation in test set 3 is 955,106,435 compared to 311,128,073 of *tcmalloc*, which shows the shortest execution time.

In the de-allocation case, the maximum total execution time of *First-fit* for de-allocation in test set 1 is longer than the values for other allocators. The maximum total execution time of *Half-fit* for de-allocation in test sets 2 and 3 is longer than the values for other allocators, while the average total execution time of *tcmalloc* for de-allocation throughout all test sets is shorter than the values for other allocators. The maximum time of *First-fit* for de-allocation in test set 1 is 11,292,341 compared to 5,941,112 of *tcmalloc*, which shows the shortest execution time. The maximum time of *Half-fit* for de-allocation in test set 2 is 24,919,366 compared to 11,810,741 of *tcmalloc*, which shows the shortest execution time. The maximum time of *Half-fit* for de-allocation in test set 3 is 631,444,518 compared to 277,782,572 of *tcmalloc*, which shows the shortest execution time.

***gawk* Results** Table 4.29, Table 4.30 and Table 4.31 show measured execution times for allocation and de-allocation on *gawk* with each allocator. The execution time of test set 1 analysis is given in Table 4.29. The execution times of test set 2 is given in Table 4.30. The execution times of test set 3 is given in Table 4.31.

Results Analysis The following shows the effect of allocation and de-allocation on time. The unit is the number of processor cycles:

- Average time: The average total execution time for allocation and de-allocation is shown in Table 4.29, Table 4.30 and Table 4.31, respectively, for all test sets. The average time of *tcmalloc* for allocation throughout test set 1 is longer than

espresso: Test Set 1					
		Average time	Min	Max	Stdev.s()
Best	malloc	10,328,312.53	9,644,153	12,938,000	442,094.10
	free	6,537,953.84	6,159,023	7,859,629	325,639.11
First	malloc	7,550,581.13	7,046,240	9,951,857	391,775.81
	free	6,586,638.37	6,237,763	11,292,341	356,683.61
Half	malloc	12,640,430.64	11,832,613	14,711,244	450,860.45
	free	9,015,318.13	8,509,403	10,561,103	367,310.15
Hoard	malloc	9,661,677.64	8,647,227	11,854,106	583,821.88
	free	5,764,900.01	5,373,793	7,156,634	328,415.41
nMART	malloc	7,441,275.90	6,970,673	8,281,692	407,161.98
	free	6,017,620.63	5,595,553	7,274,812	424,721.58
tcmalloc	malloc	14,670,130.96	13,172,530	17,586,676	629,824.20
	free	4,182,505.25	3,907,238	5,941,112	299,195.90
TLSF	malloc	6,549,822.43	5,991,619	8,630,935	414,204.54
	free	5,628,939.71	5,300,274	6,989,488	305,383.76

Table 4.26: The average of total *malloc()/free()* time for *espresso* test set 1

espresso: Test Set 2					
		Average time	Min	Max	Stdev.s()
Best	malloc	26,379,216.30	24,796,207	29,205,872	730,736.19
	free	16,038,254.18	15,228,243	18,188,727	453,431.82
First	malloc	17,584,439.04	16,660,367	20,295,355	528,887.86
	free	16,314,920.34	15,502,503	17,999,487	486,695.20
Half	malloc	30,834,373.92	29,339,010	33,140,180	691,691.49
	free	22,783,161.78	21,737,415	24,919,366	556,367.15
Hoard	malloc	16,936,101.18	15,407,714	19,277,000	595,232.68
	free	13,759,930.21	12,901,015	16,110,194	494,866.33
nMART	malloc	16,765,343.77	15,947,878	17,625,455	464,955.15
	free	14,440,379.77	13,783,375	15,349,685	450,898.30
tcmalloc	malloc	20,737,105.33	19,371,462	23,924,919	604,706.05
	free	9,886,048.68	9,286,535	11,810,741	426,149.73
TLSF	malloc	15,156,754.04	14,201,387	16,879,230	505,696.63
	free	13,673,867.85	12,884,928	18,494,313	484,922.80

Table 4.27: The average of total *malloc()/free()* time for *espresso* test set 2

espresso: Test Set 3					
		Average time	Min	Max	Stdev.s()
Best	malloc	843,847,646.09	824,496,036	885,794,752	9,927,614.48
	free	434,108,268.84	425,839,310	444,569,488	2,856,866.65
First	malloc	466,844,500.37	456,103,209	483,537,298	3,285,092.48
	free	435,319,612.56	426,407,147	453,382,335	3,051,587.22
Half	malloc	862,967,947.65	839,115,005	955,106,435	10,279,981.45
	free	612,756,957.28	602,756,763	631,444,518	3,845,383.12
Hoard	malloc	367,371,952.02	356,880,799	384,458,752	4,446,923.37
	free	410,607,470.70	401,136,569	433,555,377	5,391,360.14
nMART	malloc	462,057,571.57	455,916,470	475,354,206	4,047,006.58
	free	410,712,786.90	405,434,101	417,533,722	2,710,206.44
tcmalloc	malloc	302,131,117.47	293,516,411	311,128,073	2,447,882.64
	free	267,100,185.80	260,335,795	277,782,572	2,151,746.14
TLSF	malloc	414,795,447.03	405,313,569	437,529,858	3,370,184.19
	free	374,455,818.16	366,084,603	392,430,367	3,293,050.02

Table 4.28: The average of total *malloc()/free()* time for *espresso* test set 3

the values for other allocators. The average time of *Half-fit* for allocation in test sets 2 and 3 is longer than the values for other allocators. The average time of *tcmalloc* for allocation in test set 1 is 11,265,358.83 compared to 3,219,690.93 of *TLSF*, which shows the minimum average of the total execution time. The average time of *Half-fit* for allocation in test set 2 is 93,536,655.11 compared to 33,022,243.59 of *Hoard*, which shows the minimum average of the total execution time. The average time of *Half-fit* for allocation in test set 3 is 178,801,586.82 compared to 59,322,385.67 of *Hoard*, which shows the minimum average of the total execution time.

In the de-allocation case, the average total execution time of *Half-fit* for de-allocation throughout test sets 1 and 2 is longer than the values for other allocators. The average total execution time of *TLSF* for de-allocation throughout test set 3 is longer than the values for other allocators, while the average total execution time of *tcmalloc* for de-allocation throughout all test sets is shorter than the values for other allocators. The average time of *Half-fit* for

de-allocation in test set 1 is 1,133,643.32 compared to 619,214.35 of *tcmalloc*, which shows the minimum average of the total execution time. The average time of *Half-fit* for de-allocation in test set 2 is 33,975,761.58 compared to 20,018,277.09 of *tcmalloc*, which shows the minimum average of the total execution time. The average time of *TLSF* for de-allocation in test set 3 is 72,968,984.60 compared to 39,058,631.20 of *tcmalloc*, which shows the minimum average of the total execution time.

- Min: The minimum total execution time for allocation and de-allocation is shown in Table 4.29, Table 4.30 and Table 4.31, respectively, for all test sets. The minimum total execution time of *tcmalloc* for allocation throughout test set 1 is longer than the values for other allocators. The minimum of total execution time of *Half-fit* for allocation throughout test sets 2 and 3 is longer than the values for other allocators. The minimum time of *tcmalloc* for allocation in test set 1 is 9,877,625 compared to 2,567,746 of *TLSF*, which shows the shortest execution time. The minimum time of *Half-fit* for allocation in test set 2 is 88,465,725 compared to 30,342,070 of *Hoard*, which shows the shortest execution time. The minimum time of *Half-fit* for allocation in test set 3 is 169,767,938 compared to 55,160,997 of *Hoard*, which shows the shortest execution time.

  In the de-allocation case, the minimum total execution time of *Half-fit* for de-allocation throughout test sets 1 and 2 is longer than the values for other allocators. The minimum total execution time of *TLSF* for de-allocation throughout test set 3 is longer than the values for other allocators, while the average total execution time of *tcmalloc* for de-allocation throughout all test sets is shorter than the values for other allocators. The minimum time of *Half-fit* for de-allocation in test set 1 is 1,034,898 compared to 575,811 of *tcmalloc*, which shows the shortest execution time. The minimum time of *Half-fit* for de-allocation in test set 2 is 32,049,395 compared to 18,679,242 of *tcmalloc*, which shows the shortest execution time. The minimum time of *TLSF* for de-allocation in test set 3 is 67,495,459 compared to 36,333,583 of *tcmalloc*, which shows the shortest execution time.

- Max: The maximum total execution time for allocation and de-allocation is shown in Table 4.29, Table 4.30 and Table 4.31, respectively, for all test sets. The maximum total execution time of *tcmalloc* for allocation throughout test set 1 is longer than the values for other allocators. The maximum total execution time of *Half-fit* for allocation throughout test sets 2 and 3 is longer than the values for other allocators. The maximum time of *tcmalloc* for allocation in test set 1 is 13,714,334 compared to 5,181,362 of *nMART*, which shows the shortest execution time. The maximum time of *Half-fit* for allocation in test set 2 is 111,6135,733 compared to 37,357,520 of *Hoard*, which shows the shortest execution time. The maximum time of *Half-fit* for allocation in test set 3 is 203,002,792 compared to 66,089,745 of *Hoard*, which shows the shortest execution time.

  In the de-allocation case, the maximum total execution time of *Half-fit* for de-allocation in test sets 1 and 2 is longer than the values for other allocators. The maximum total execution time of *TLSF* for de-allocation in test set 3 is longer than the values for other allocators, while the average total execution time of *tcmalloc* for de-allocation throughout all test sets is shorter than the values for other allocators. The maximum time of *Half-fit* for de-allocation in test set 1 is 2,182,718 compared to 1,220,199 of *nMART*, which shows the shortest execution time. The maximum time of *Half-fit* for de-allocation in test set 2 is 39,902,071 compared to 25,450,369 of *tcmalloc*, which shows the shortest execution time. The maximum time of *TLSF* for de-allocation in test set 3 is 84,677,618 compared to 43,690,615 of *tcmalloc*, which shows the shortest execution time.

***p2c* Results** Table 4.32, Table 4.33 and Table 4.34 show measured execution times for allocation and de-allocation on *p2c* with each allocator. The execution time of test set 1 analysis is given in Table 4.32. The execution times of test set 2 is given in Table 4.33. The execution times of test set 3 is given in Table 4.34.

Results Analysis The following shows the effect of allocation and de-allocation on time. The unit is the number of processor cycles:

gawk: Test Set 1					
		Average time	Min	Max	Stdev.s()
Best	malloc	4,004,168.65	3,326,082	5,581,117	319,674.13
	free	784,970.30	712,115	1,874,873	127,229.80
First	malloc	3,875,239.83	3,266,832	6,052,402	320,310.49
	free	798,982.80	722,974	1,431,708	113,666.49
Half	malloc	4,597,032.68	3,956,949	9,619,640	375,456.03
	free	1,133,643.32	1,034,898	2,182,718	158,332.48
Hoard	malloc	5,633,024.10	4,774,550	7,534,736	529,870.93
	free	1,026,283.37	930,000	2,014,151	143,145.47
nMART	malloc	4,350,778.03	3,625,369	5,181,362	424,619.84
	free	723,030.60	657,718	1,220,199	130,595.64
tcmalloc	malloc	11,265,358.83	9,877,625	13,714,334	577,914.92
	free	619,214.35	575,811	1,336,600	106,386.76
TLSF	malloc	3,219,690.93	2,567,746	6,899,505	431,800.94
	free	788,654.38	721,705	1,672,622	133,518.00

Table 4.29: The average of total *malloc()/free()* time for *gawk* test set 1

gawk: Test Set 2					
		Average time	Min	Max	Stdev.s()
Best	malloc	77,652,202.14	72,090,100	92,744,466	2,520,010.46
	free	23,874,023.79	22,408,569	28,630,504	715,956.77
First	malloc	91,630,822.58	85,161,733	103,113,256	2,435,126.61
	free	24,688,437.70	23,266,066	27,449,853	701,456.81
Half	malloc	93,536,655.11	88,465,725	116,135,733	2,462,966.22
	free	33,975,761.58	32,049,395	39,902,071	856,844.32
Hoard	malloc	33,022,243.59	30,342,070	37,357,520	1,095,715.25
	free	30,583,551.06	28,345,035	38,402,113	1,078,810.53
nMART	malloc	65,622,199.33	61,257,440	81,409,187	2,710,888.32
	free	23,528,674.20	21,858,185	27,575,051	1,420,528.59
tcmalloc	malloc	63,961,514.05	58,154,229	104,167,706	2,738,466.07
	free	20,018,277.09	18,679,242	25,450,369	710,908.08
TLSF	malloc	59,626,468.82	53,367,232	75,364,016	2,565,426.78
	free	26,278,313.25	23,877,076	29,828,019	1,079,320.69

Table 4.30: The average of total *malloc()/free()* time for *gawk* test set 2

gawk: Test Set 3					
		Average time	Min	Max	Stdev.s()
Best	malloc	146,307,777.65	138,162,948	164,115,947	3,757,203.22
	free	46,289,994.15	43,311,540	50,282,595	1,074,525.93
First	malloc	152,033,034.24	143,053,557	167,062,603	3,993,567.21
	free	47,813,010.74	45,367,890	52,591,632	1,046,272.40
Half	malloc	178,801,586.82	169,767,938	203,002,792	4,236,118.20
	free	66,094,928.58	62,662,780	72,664,234	1,335,986.67
Hoard	malloc	59,322,385.67	55,160,997	66,089,745	1,651,487.67
	free	59,063,328.81	54,682,072	65,231,703	1,680,567.01
nMART	malloc	171,283,594.57	158,802,351	202,901,719	5,179,193.61
	free	65,091,000.33	60,527,727	78,087,514	4,207,347.75
tcmalloc	malloc	100,261,536.18	92,410,314	160,213,948	4,524,142.97
	free	39,058,631.20	36,333,583	43,690,615	1,103,773.99
TLSF	malloc	157,745,105.13	142,956,387	192,059,833	5,972,851.08
	free	72,968,984.60	67,495,459	84,677,618	2,488,447.24

Table 4.31: The average of total *malloc()/free()* time for *gawk* test set 3

- Average time: The average total execution time for allocation and de-allocation is shown in Table 4.32, Table 4.33 and Table 4.34, respectively, for all test sets. The average time of *tcmalloc* for allocation throughout test sets 1 and 2 is longer than the values for other allocators. The average time of *Best-fit* for allocation in test set 3 is longer than the values for other allocators. The average time of *tcmalloc* for allocation in test set 1 is 10,259,259.52 compared to 2,271,061.37 of *TLSF*, which shows the minimum average of the total execution time. The average time of *tcmalloc* for allocation in test set 2 is 11,645,160.86 compared to 4,042,385.27 of *TLSF*, which shows the minimum average of the total execution time. The average time of *Best-fit* for allocation in test set 3 is 105,703,752.07 compared to 14,042,696.28 of *Hoard*, which shows the minimum average of the total execution time.

  In the de-allocation case, the average total execution time of *Hoard* for de-allocation throughout test set 1 is longer than the values for other allocators. The average total execution time of *Half-fit* for de-allocation throughout test

sets 2 and 3 is longer than the values for other allocators. The average time of *Hoard* for de-allocation in test set 1 is 350,507.46 compared to 169,927.37 of *First-fit*, which shows the minimum average of the total execution time. The average time of *Half-fit* for de-allocation in test set 2 is 1,333,855.13 compared to 874,377.20 of *tcmalloc*, which shows the minimum average of the total execution time. The average time of *Half-fit* for de-allocation in test set 3 is 13,434,286.86 compared to 7,216,987.22 of *tcmalloc*, which shows the minimum average of the total execution time.

- Min: The minimum total execution time for allocation and de-allocation is shown in Table 4.32, Table 4.33 and Table 4.34, respectively, for all test sets. The minimum total execution time of *tcmalloc* for allocation throughout test sets 1 and 2 is longer than the values for other allocators. The minimum total execution time of *Best-fit* for allocation throughout test set 3 is longer than the values for other allocators. The minimum time of *tcmalloc* for allocation in test set 1 is 9,400,304 compared to 1,875,452 of *TLSF*, which shows the shortest execution time. The minimum time of *Half-fit* for allocation in test set 2 is 10,544,113 compared to 3,376,732 of *TLSF*, which shows the shortest execution time. The minimum time of *Best-fit* for allocation in test set 3 is 101,096,233 compared to 12,791,094 of *Hoard*, which shows the shortest execution time.

  In the de-allocation case, the minimum total execution time of *Hoard* for de-allocation throughout test set 1 is longer than the values for other allocators. The minimum total execution time of *Half-fit* for de-allocation throughout test sets 2 and 3 is longer than the values for other allocators. The minimum time of *Hoard* for de-allocation in test set 1 is 304,415 compared to 150,019 of *Best-fit*, which shows the shortest execution time. The minimum time of *Half-fit* for de-allocation in test set 2 is 1,181,016 compared to 789,064 of *tcmalloc*, which shows the shortest execution time. The minimum time of *Half-fit* for de-allocation in test set 3 is 12,542,273 compared to 6,702,335 of *tcmalloc*, which shows the shortest execution time.

- Max: The maximum total execution time for allocation and de-allocation is shown in Table 4.32, Table 4.33 and Table 4.34, respectively, for all test sets. The maximum total execution time of *tcmalloc* for allocation throughout test sets 1 and 2 is longer than the values for other allocators. The maximum total execution time of *Best-fit* for allocation throughout test set 3 is longer than the values for other allocators. The maximum time of *tcmalloc* for allocation in test set 1 is 12,397,525 compared to 4,221,169 of *TLSF*, which shows the shortest execution time. The maximum time of *tcmalloc* for allocation in test set 2 is 13,421,021 compared to 6,268,721 of *TLSF*, which shows the shortest execution time. The maximum time of *Best-fit* for allocation in test set 3 is 117,884,428 compared to 16,393,020 of *Hoard*, which shows the shortest execution time.

  In the de-allocation case, the maximum total execution time of *TLSF* for de-allocation in test set 1 is longer than the values for other allocators. The maximum total execution time of *Half-fit* for de-allocation in test sets 2 and 3 is longer than the values for other allocators. The maximum time of *TLSF* for de-allocation in test set 1 is 1,383,458 compared to 516,279 of *nMART*, which shows the shortest execution time. The maximum time of *Half-fit* for de-allocation in test set 2 is 6,330,245 compared to 1,482,982 of *nMART*, which shows the shortest execution time. The maximum time of *Half-fit* for de-allocation in test set 3 is 15,711,530 compared to 9,218,633 of *tcmalloc*, which shows the shortest execution time.

In summary, it is hard to specify which allocation algorithm shows the best performance for all cases. As we see in the results, nothing is the best in both the allocation and de-allocation cases. Also, different results are obtained between less memory-intensive test and memory-intensive tests. Conventional algorithms show good performance for less memory-intensive test (Test set 1), while more complex algorithms, *tcmalloc* and *Hoard*, show better performance for memory-intensive test (Test set 3). *tcmalloc* shows the best performance in all de-allocation cases. Overall, *Best-fit* and *Half-fit* show worse performance in memory-intensive test sets. *nMART* shows better performance for memory-intensive tests compared to conventional al-

p2c: Test Set 1					
		Average time	Min	Max	Stdev.s()
Best	malloc	3,536,707.14	2,940,174	5,164,199	401,214.49
	free	170,251.83	150,019	1,180,812	90,282.79
First	malloc	3,535,246.34	2,924,904	5,044,031	391,932.90
	free	169,927.37	150,808	829,227	83,369.14
Half	malloc	4,592,039.95	3,965,106	6,640,588	393,094.02
	free	225,270.20	197,772	883,503	59,014.28
Hoard	malloc	4,988,660.57	4,275,551	6,658,776	474,745.61
	free	350,507.46	304,415	1,078,272	98,245.17
nMART	malloc	3,348,603.30	2,802,990	4,664,745	403,704.74
	free	171,917.43	152,574	516,279	65,593.32
tcmalloc	malloc	10,259,259.52	9,400,304	12,397,525	470,488.62
	free	306,990.85	273,705	990,499	101,966.95
TLSF	malloc	2,271,061.37	1,875,452	4,221,169	333,791.27
	free	330,286.36	290,193	1,383,458	124,011.26

Table 4.32: The average of total *malloc()/free()* time for *p2c* test set 1

p2c: Test Set 2					
		Average time	Min	Max	Stdev.s()
Best	malloc	6,551,038.67	5,713,344	8,960,900	465,389.62
	free	927,219.61	851,201	1,855,954	159,373.98
First	malloc	6,378,660.84	5,454,360	8,549,224	477,481.88
	free	932,289.52	859,685	1,935,571	161,270.39
Half	malloc	8,253,528.02	7,184,759	10,606,201	520,793.58
	free	1,333,855.13	1,181,016	6,330,245	274,887.27
Hoard	malloc	5,909,893.87	5,180,354	7,649,035	483,369.98
	free	1,012,280.91	909,421	2,087,321	174,517.94
nMART	malloc	6,065,802.73	5,344,974	7,053,384	432,010.73
	free	954,985.07	857,972	1,482,982	110,828.26
tcmalloc	malloc	11,645,160.86	10,544,113	13,421,021	540,093.17
	free	874,377.20	789,064	2,325,030	176,367.93
TLSF	malloc	4,042,385.27	3,376,732	6,268,721	454,395.55
	free	1,072,134.79	980,492	2,190,368	179,519.62

Table 4.33: The average of total *malloc()/free()* time for *p2c* test set 2

p2c: Test Set 3					
		Average time	Min	Max	Stdev.s()
Best	malloc	105,703,752.07	101,096,233	117,884,428	1,611,412.11
	free	9,634,891.93	8,892,125	14,960,253	505,587.35
First	malloc	25,478,882.70	22,635,424	28,974,644	992,066.61
	free	10,735,226.45	9,961,231	13,419,804	479,526.87
Half	malloc	34,599,243.03	31,233,053	40,697,240	1,366,546.33
	free	13,434,286.86	12,542,273	15,711,530	513,908.55
Hoard	malloc	14,042,696.28	12,791,094	16,393,020	678,372.73
	free	8,656,893.30	8,059,247	10,531,219	419,070.60
nMART	malloc	18,349,030.63	16,809,797	20,504,014	793,589.33
	free	10,575,562.93	9,797,885	11,483,243	461,995.54
tcmalloc	malloc	22,053,727.28	20,269,696	25,431,394	751,976.81
	free	7,216,987.22	6,702,335	9,218,633	422,493.03
TLSF	malloc	16,994,133.75	14,713,285	22,510,016	956,598.68
	free	8,836,206.46	8,040,846	13,917,636	542,513.04

Table 4.34: The average of total *malloc()/free()* time for *p2c* test set 3

location algorithms.

4.3.3 Remote Access Latencies Analysis

As we discussed in Section 4.2.2, no model exists to evaluate remote access latencies on ccNUMA architecture systems regarding memory allocation algorithms. For this reason, we have used the synthetic workloads model generated in tables C.1 and 4.10.

4.3.3.1 The Results of MEAN-Value Model

This experiment was executed using the data generated with each allocation algorithm. An application was executed 100 times with each allocator repeatedly so that the application was executed 700 times with seven different allocators totally. In this experiment, we measure the execution time of an application request to allocate memory under the interarrival time and to hold memory blocks based on Table 4.9. It created a varying number of threads ranging from two to sixty-four. In the local

case, the application was executed on node 1; it requested allocating a varying size of memory blocks from the local node only and writing data onto the memory block to ensure pages in. However, in the remote case, the application created the same number of threads to the local cases and was executed on node 1 (the same as the local case), but it allocated a varying size of memory blocks from only the remote node, which was chosen automatically by the underlying operating system. In order to ensure pages in, it writes meaningless data onto the memory block.

Table 4.35, 4.36, 4.37, 4.38, 4.39 and 4.40 show the results of execution time with each allocation algorithm in both cases.

Two Threads Case Table 4.35 gives the average of execution times (unit is second) elapsed in local and remote access cases as well as the standard deviation with each allocator. The application created two threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution times elapsed for local accesses with each allocator is shown in Table 4.35 for the MEAN model. The average execution times of *TLSF* is lower than the values for other allocators, while the *First-fit* shows higher execution time relatively. The average execution time of *TLSF* in the MEAN model is 4.08 seconds compared to 6.91 seconds of *First-fit*. This is due to the differences in the implementations. As seen in Appendix A.1.2, *TLSF* invokes a library wrapper function less often to require more space, and *Hoard* and *tcmalloc* use the *mmap()* system call for larger memory blocks to bypass the request of memory allocation to the underlying operating system, but *TLSF* uses the *sbrk()* function to have more memory space without any size class distinction. Also, our implementations of *Best-fit*, *First-fit* and *Half-fit* invoke the *sbrk()* function call more, so their management efficiency is worse. In detail, *TLSF* obtains 10k bytes as a minimum-size memory block, with unused memory space being reserved when it invokes *sbrk()*, while our implementation just obtains the exact-size memory block requested. However, *TLSF* and *Half-fit* show an uneven performance relatively, as seen in the

standard deviation.

	Local		Remote	
	Average	Stdev	Average	Stdev
Best	6.6389	0.1057	7.3977	0.0640
First	6.9135	0.1413	7.4281	0.5119
Half	6.8153	0.3509	7.3425	0.2742
Hoard	6.5890	0.2208	7.3829	0.3091
nMART	5.1158	0.2830	5.9431	0.4850
tcmalloc	6.3901	0.2026	7.3935	0.3966
TLSF	4.0811	0.3798	7.0840	0.4805

Table 4.35: The execution time of MEAN-Value model with two threads

- Average of remote case: The average execution times elapsed for remote accesses with each allocator is shown in Table 4.35 respectively for the MEAN model. The average time of *nMART* is lower than the values for other allocators, while the *First-fit* shows higher execution time relatively. The average of the execution time of *nMART* in the MEAN model is 5.94 seconds compared to 7.43 seconds of *First-fit*. The average execution time of remote access of*TLSF* increases significantly compared to its local case. Also, its performance and *First-fit* performance are relatively uneven.

Four Threads Case Table 4.36 gives the average execution time (unit is second) elapsed in local and remote access cases as well as the standard deviation with each allocator. The application created four threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution times elapsed of local accesses with each allocator is shown in Table 4.36 respectively for the MEAN model. The average execution time of *TLSF* is lower than the values for other allocators, while the *Hoard* shows higher execution time relatively. The average

	Local		Remote	
	Average	Stdev	Average	Stdev
Best	6.3045	0.1683	6.8881	0.1516
First	5.9749	0.3419	6.8068	0.2668
Half	6.4812	0.5264	7.4000	0.5649
Hoard	6.6679	0.5615	7.3885	0.4822
nMART	4.5191	0.5491	5.3346	0.7689
tcmalloc	6.1237	0.1159	6.8478	0.1526
TLSF	3.2924	0.5914	6.2848	1.0704

Table 4.36: The execution time of MEAN-Value model with four threads

of the execution time of *TLSF* in the MEAN model is 3.29 seconds compared to 6.67 seconds of *Hoard*. *Half-fit*, *Hoard* and *TLSF* show a relatively uneven performance, as seen in the standard deviation.

- Average of remote case: The average execution times elapsed of remote accesses with each allocator is shown in Table 4.36 respectively for the MEAN model. The average time of *nMART* is lower than the values for other allocators, while the *Half-fit* shows higher execution time relatively. The average of the execution time of *nMART* in the MEAN model is 5.33 seconds compared to 7.40 seconds of *Half-fit*. However, the average execution time of remote access of*TLSF* increases significantly compared to its local case. Also, its performance is relatively uneven.

Eight Threads Case Table 4.37 gives the average execution time (unit is second) elapsed in local and remote access cases as well as the standard deviation with each allocator. The application created eight threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution times elapsed of local accesses with each allocator is shown in Table 4.37 respectively for the MEAN model. The average execution time of *TLSF* is lower than the values for other allo-

cators, while the *Hoard* shows higher execution time relatively. The average of the execution time of *TLSF* in the MEAN model is 3.76 seconds compared to 6.36 seconds of *Hoard*. *Half-fit* shows an uneven performance relatively, as seen in the standard deviation.

	Local		Remote	
	Average	Stdev	Average	Stdev
Best	5.8451	0.3080	6.6151	0.3288
First	6.1348	0.3719	6.6689	0.2177
Half	6.1928	0.6568	6.4728	0.3166
Hoard	6.3591	0.3658	6.6722	0.1728
nMART	5.1682	0.3220	5.6198	0.5585
tcmalloc	6.0175	0.1753	6.5425	0.2530
TLSF	3.7646	0.2166	6.7519	0.6057

Table 4.37: The execution time of MEAN-Value model with eight threads

- Average of remote case: The average execution times elapsed of remote accesses with each allocator is shown in Table 4.37 respectively for the MEAN model. The average time of *nMART* is lower than the values for other allocators, while the *TLSF* shows higher execution time relatively. The average of the execution time of *nMART* in the MEAN model is 5.62 seconds compared to 6.75 seconds of *TLSF*. Also, the average execution time of remote access of*TLSF* increases significantly compared to its local case and others, and shows an uneven performance relatively.

Sixteen Threads Case Table 4.38 gives the average execution times elapsed (unit is second) in local and remote access cases as well as the standard deviation with each allocator. The application created sixteen threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution times elapsed of local accesses with each allocator is shown in Table 4.38 respectively for the MEAN model.

The average execution time of *TLSF* is lower than the values for other allocators, while the *First-fit* shows higher execution time relatively. The average of the execution time of *TLSF* in the MEAN model is 3.88 seconds compared to 6.17 seconds of *First-fit*. All allocators except *TLSF* show an uneven performance relatively, as seen in the standard deviation.

	Local		Remote	
	Average	Stdev	Average	Stdev
Best	6.1565	0.5857	6.8781	0.3270
First	6.1699	0.5013	6.7909	0.2925
Half	6.0922	0.5886	6.5189	0.2452
Hoard	6.1151	0.5544	6.5744	0.4342
nMART	4.8258	0.2020	5.9348	0.2203
tcmalloc	5.6457	0.6970	6.6678	0.2603
TLSF	3.8823	0.2462	7.2862	0.4319

Table 4.38: The execution time of MEAN-Value model with sixteen threads

- Average of remote case: The average execution times elapsed of remote accesses with each allocator is shown in Table 4.38 respectively for the MEAN model. The average time of *nMART* is lower than the values for other allocators, while the *TLSF* shows higher execution time relatively. The average of the execution time of *nMART* in the MEAN model is 5.93 seconds compared to 7.29 seconds of *TLSF*. Also, the average execution time of remote access of*TLSF* increases significantly compared to its local case and others. *Hoard* and *TLSF* show an uneven performance relatively.

Thirty-two Threads Case Table 4.39 gives the average execution times elapsed (unit is second) in local and remote access cases as well as the standard deviation with each allocator. The application created thirty-two threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution times elapsed of local accesses with each allocator is shown in Table 4.39 respectively for the MEAN model. The average execution time of *TLSF* is lower than the values for other allocators, while the *Best-fit* shows higher execution time relatively. The average of the execution time of *TLSF* in the MEAN model is 3.76 seconds compared to 5.85 seconds of *Best-fit*. *Best-fit*, *First-fit*, *Half-fit* and *Hoard* show an uneven performance relatively, as seen in the standard deviation.

	Local		Remote	
	Average	Stdev	Average	Stdev
Best	5.8541	1.0396	6.6619	0.7130
First	5.7748	0.6880	6.6971	0.5723
Half	5.6066	0.8405	6.7986	0.9280
Hoard	5.6891	0.6380	6.7006	0.3901
nMART	4.4028	0.0651	5.1337	0.0905
tcmalloc	5.8395	0.2146	6.8555	0.3132
TLSF	3.7640	0.2259	7.6787	0.3679

Table 4.39: The execution time of MEAN-Value model with thirty-two threads

- Average of remote case: The average execution times elapsed of remote accesses with each allocator is shown in Table 4.39 respectively for the MEAN model. The average time of *nMART* is lower than the values for other allocators, while the *TLSF* shows higher execution time relatively. The average of the execution time of *nMART* in the MEAN model is 5.13 seconds compared to 7.68 seconds of *TLSF*. Also, the average execution time of remote access of*TLSF* increases significantly compared to its local case and others. *Best-fit*, *First-fit* and *Half-fit* show an uneven performance relatively.

Sixty-four Threads Case Table 4.40 gives the average execution times elapsed (unit is second) in local and remote access cases as well as the standard deviation with each allocator. The application created sixty-four threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution time elapsed of local accesses with each allocator is shown in Table 4.40 respectively for the MEAN model. The average execution time of *TLSF* is lower than the values for other allocators, while the *Best-fit* shows higher execution time relatively. The average of the execution time of *TLSF* in the MEAN model is 4.63 seconds compared to 5.79 seconds of *Best-fit*. *Best-fit*, *First-fit*, *Half-fit* and *Hoard* show an uneven performance relatively, as seen in the standard deviation.

	Local		Remote	
	Average	Stdev	Average	Stdev
Best	5.7895	0.8419	6.2835	1.0447
First	5.7215	0.8983	6.4076	1.2089
Half	5.2989	1.4567	6.4857	1.1155
Hoard	5.6645	0.7300	6.5553	0.8613
nMART	4.3491	0.4385	5.0684	0.0573
tcmalloc	5.4834	0.2084	6.6359	0.3388
TLSF	4.6261	0.3296	6.5536	0.2329

Table 4.40: The execution time of MEAN-Value model with sixty-four threads

- Average of remote case: The average execution times elapsed of remote accesses with each allocator is shown in Table 4.40 respectively for the MEAN model. The average time of *nMART* is lower than the values for other allocators, while the *tcmalloc* shows higher execution time relatively. The average of the execution time of *nMART* in the MEAN model is 6.28 seconds compared to 5.07 seconds of *tcmalloc*. The average execution time of remote access of *TLSF* increases significantly compared to its local case and others. *Best-fit*, *First-fit*, *Half-fit* and *Hoard* show an uneven performance relatively.

In summary, for local memory access cases, *TLSF* shows better performance, with all numbers of threads ranging from two to sixty-four; otherwise, *First-fit* and *Best-fit* show the worst performance when the application uses a large number of threads. For remote memory access cases, *First-fit* and *Best-fit* show better

performance than *TLSF*, which shows the worst performance using large number of threads ranging from eight to thirty-two. Especially, *nMART* shows the best performance for remote memory access, with all number of threads ranging from two to sixty-four. For local memory access, it also shows better performance compared to others except *TLSF*.

In summary, for local memory access cases, *TLSF* shows better performance, with all numbers of threads ranging from two to sixty-four, and *nMART* shows better performance compared to others except *TLSF*; otherwise, *First-fit* and *Best-fit* show the worst performance when the application uses a large number of threads. In the remote memory access cases, *First-fit* and *Best-fit* show better performance than *TLSF*, which shows the worst performance using a large number of threads ranging from eight to thirty-two. *nMART* shows the best performance for remote memory access, with all number of threads ranging from two to sixty-four. It shows that our algorithm, specially the third layer functionalities, is able to improve performance of remote memory access.

4.3.3.2 The Results of CDF Model

This experiment was executed using the data generated with each allocation algorithm. An application was executed 100 times with each allocator repeatedly so that the application was executed 700 times with seven different allocators totally. In this experiment, we measure the execution time of an application request to allocate memory under the interarrival time and to hold memory blocks based on Table 4.10. It created a varying number of threads ranging from two to sixty-four. In the local case, the application was executed on node 3; it requested allocating a varying size of memory blocks from the local node only and writing data onto the memory block to ensure pages in. However, in the remote case, the application created the same number of threads to the local cases, with it being executed on node 3 (the same as the local case), but it allocated a varying size of memory blocks from only the remote node, which was chosen automatically by the underlying operating system. In order to ensure pages in, it writes meaningless data onto the memory block.

Tables 4.41, 4.42, 4.43, 4.44, 4.45 and 4.46 show the results of execution times

with each allocation algorithm in both cases.

Two Threads Case Table 4.41 gives the average execution times elapsed (unit is second) in local and remote access cases as well as the standard deviation with each allocator. The application created two threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution times elapsed of local accesses with each allocator is shown in Table 4.41 respectively for the CDF model. The average execution time of *TLSF* is lower than the values for other allocators, while the *tcmalloc* shows higher execution time relatively. *TLSF* shows improved performance compared to others around 57%. The average of the execution time of *TLSF* in the CDF model is 3.84 seconds compared to 6.64 seconds of *tcmalloc*. It was caused by the implementation difference. As seen in Appendix A.1.2, *TLSF* invokes a library wrapper function less to have more space, and *Hoard* and *tcmalloc* use the *mmap()* system call for larger memory blocks to bypass the request of memory allocation to the underlying operating system, but *TLSF* uses the *sbrk()* function to have more memory space without any size class distinction. Also, our implementations of *Best-fit*, *First-fit* and *Half-fit* invoke the *sbrk()* function call more, as their management efficiency is worse. In detail, *TLSF* obtains 10k bytes as a minimum-size memory block, with unused memory space being reserved when it invokes *sbrk()*, while our implementation just obtains the exact-size memory block requested. *Hoard* shows an uneven performance relatively, as seen in the standard deviation.

- Average of remote case: The average execution times elapsed of remote accesses with each allocator is shown in Table 4.41 respectively for the CDF model. The average time of *nMART* is lower than the values for other allocators, while the *Half-fit* shows higher execution time relatively. The average of the execution time of *nMART* in the CDF model is 6.04 seconds compared to 8.71 seconds of *Half-fit*. The average execution time of remote access of *TLSF*

increases significantly compared to its local case and others. Also, its performance is uneven relatively.

	Local access time		Remote access time	
	Average	Stdev	Average	Stdev
Best	5.9057	0.0302	6.7465	0.4893
First	6.3614	0.2851	7.6078	0.5846
Half	6.4654	0.2241	8.7072	0.0414
Hoard	6.1715	1.0505	7.5149	0.4544
nMART	5.5737	0.6063	6.0446	0.4803
tcmalloc	6.6380	0.1576	8.0231	0.4389
TLSF	3.8354	0.4308	6.9286	0.8189

Table 4.41: The execution time of CDF model with two threads

Four Threads Case Table 4.42 gives the average executions time elapsed (unit is second) in local and remote access cases as well as the standard deviation with each allocator. The application created four threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution times elapsed of local accesses with each allocator is shown in Table 4.42 respectively for the CDF model. The average execution time of *TLSF* is lower than the values for other allocators, while the *Half-fit* shows higher execution time relatively. The average of the execution time of *TLSF* in the CDF model is 3.30 seconds compared to 6.42 seconds of *Half-fit*. *Hoard* shows an uneven performance relatively, as seen in the standard deviation.

- Average of remote case: The average execution times elapsed of remote accesses with each allocator is shown in Table 4.42 respectively for the CDF model. The average time of *nMART* is lower than the values for other allocators, while the *Half-fit* shows higher execution time relatively. The average of the execution time of *nMART* in the CDF model is 5.64 seconds compared to

8.21 seconds of *Half-fit*. However, the average execution time of remote access of*TLSF* increases significantly compared to its local case and others. Also, its performance is uneven relatively.

	Local access time		Remote access time	
	Average	Stdev	Average	Stdev
Best	6.2488	0.1175	7.4000	0.1951
First	6.4106	0.2021	7.7403	0.5730
Half	6.4180	0.1435	8.2129	0.4779
Hoard	5.7518	1.1971	7.9046	0.7157
nMART	5.0215	0.6239	5.6373	0.7667
tcmalloc	6.1712	0.1223	7.5282	0.1899
TLSF	3.3036	0.6212	6.1508	0.8814

Table 4.42: The execution time of CDF model with four threads

Eight Threads Case Table 4.43 gives the average execution time elapsed (unit is second) in local and remote access cases as well as the standard deviation with each allocator. The application created eight threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution times elapsed of local accesses with each allocator is shown in Table 4.43 respectively for the CDF model. The average execution time of *TLSF* is lower than the values for other allocators, while the *First-fit* shows higher execution time relatively. The average of the execution time of *TLSF* in the CDF model is 3.73 seconds compared to 6.86 seconds of *First-fit*. *Half-fit* shows an uneven performance relatively, as seen in the standard deviation.

- Average of remote case: The average execution times elapsed of remote accesses with each allocator is shown in Table 4.43 respectively for the CDF model. The average time of *nMART* is lower than the values for other allocators, while the *Half-fit* shows higher execution time relatively. The average of

the execution time of *nMART* in the CDF model is 5.68 seconds compared to 7.98 seconds of *Half-fit*. However, the average execution time of remote access of*TLSF* increases significantly compared to its local case and others. *Hoard* shows an uneven performance relatively.

	Local access time		Remote access time	
	Average	Stdev	Average	Stdev
Best	5.8954	0.3234	7.2477	0.3181
First	6.8640	0.6344	7.5226	0.4851
Half	6.7173	1.0284	7.9788	0.4252
Hoard	5.9764	0.8866	7.2421	0.6281
nMART	4.8253	0.3649	5.6759	0.6011
tcmalloc	5.7272	0.3054	7.2314	0.3112
TLSF	3.7300	0.2774	6.9798	0.3953

Table 4.43: The execution time of CDF model with eight threads

Sixteen Threads Case Table 4.44 gives the average execution times elapsed (unit is second) in local and remote access cases as well as the standard deviation with each allocator. The application created sixteen threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution time elapsed of local accesses with each allocator is shown in Table 4.44 respectively for the CDF model. The average execution time of *TLSF* is lower than the values for other allocators, while the *Half-fit* shows higher execution time relatively. The average of the execution time of *TLSF* in the CDF model is 3.74 seconds compared to 5.82 seconds of *Half-fit*. *Half-fit* and *First-fit* show an uneven performance relatively, as seen in the standard deviation.

- Average of remote case: The average execution time elapsed of remote accesses with each allocator is shown in Table 4.44 respectively for the CDF model. The average time of *nMART* is lower than the values for other allocators, while the

First-fit shows higher execution time relatively. The average of the execution time of *nMART* in the CDF model is 5.93 seconds compared to 7.80 seconds of *First-fit*. However, the average execution time of remote access of *TLSF* increases significantly compared to its local case and others. *Best-fit*, *First-fit* and *Hoard* show an uneven performance relatively.

	Local access time		Remote access time	
	Average	Stdev	Average	Stdev
Best	5.7641	0.3679	7.4050	0.6060
First	5.8179	0.6913	7.7964	0.7187
Half	5.8192	0.6645	7.5818	0.5230
Hoard	5.6188	0.4695	7.4572	0.6212
nMART	4.4986	0.2674	5.9343	0.2520
tcmalloc	5.5445	0.3397	7.0841	0.4480
TLSF	3.7425	0.3693	7.4842	0.3074

Table 4.44: The execution time of CDF model with sixteen threads

Thirty-two Threads Case Table 4.45 gives the average execution times elapsed (unit is second) in local and remote access cases as well as the standard deviation with each allocator. The application created thirty-two threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution times elapsed of local accesses with each allocator is shown in Table 4.45 respectively for the CDF model. The average execution time of *TLSF* is lower than the values for other allocators, while the *First-fit* shows higher execution time relatively. The average of the execution time of *TLSF* in the CDF model is 3.66 seconds compared to 5.98 seconds of *First-fit*. *Best-fit*, *First-fit* and *Half-fit* show an uneven performance relatively, as seen in the standard deviation.

- Average of remote case: The average execution times elapsed of remote accesses with each allocator is shown in Table 4.45 respectively for the CDF

model. The average time of *nMART* is lower than the values for other allocators, while the *TLSF* shows higher execution time relatively. The average of the execution time of *nMART* in the CDF model is 7.58 seconds compared to 5.12 seconds of *TLSF*. The average execution time of remote access of *TLSF* increases significantly compared to its local case and others. *Best-fit*, *First-fit*, *Half-fit* and *Hoard* show an uneven performance relatively.

	Local access time		Remote access time	
	Average	Stdev	Average	Stdev
Best	5.9188	0.6715	7.5774	0.7946
First	5.9845	0.6335	7.3445	0.8834
Half	5.7718	0.7998	7.4435	0.7959
Hoard	5.7381	0.5245	7.3596	0.7406
nMART	4.4497	0.1619	5.1217	0.0998
tcmalloc	5.8253	0.2567	7.4613	0.3817
TLSF	3.6567	0.3670	7.8142	0.2646

Table 4.45: The execution time of CDF model with thirty-two threads

Sixty-four Threads Case Table 4.46 gives the average execution times elapsed (unit is second) in local and remote access cases as well as the standard deviation with each allocator. The application created sixty-four threads at start-up time. Each thread allocates and holds memory blocks and writes meaningless data.

- Average of local case: The average execution times elapsed of local accesses with each allocator is shown in Table 4.46 respectively for the CDF model. The average execution time of *TLSF* is lower than the values for other allocators, while the *Hoard* shows higher execution time relatively. The average of the execution time of *TLSF* in the CDF model is 4.76 seconds compared to 6.32 seconds of *Hoard*. *Best-fit*, *First-fit*, *Half-fit* and *Hoard* show an uneven performance relatively, as seen in the standard deviation.
- Average of remote case: The average execution times elapsed of remote accesses with each allocator is shown in Table 4.46 respectively for the CDF

model. The average time of *nMART* is lower than the values for other allocators, while the *First-fit* shows higher execution time relatively. The average of the execution time of *nMART* in the CDF model is 6.06 seconds compared to 7.54 seconds of *First-fit*. However, the average execution time of remote access of *TLSF* increases significantly compared to its local case and others. *Best-fit*, *First-fit*, *Half-fit* and *Hoard* show an uneven performance relatively.

	Local access time		Remote access time	
	Average	Stdev	Average	Stdev
Best	5.6636	1.1111	7.1593	0.9167
First	5.3398	1.2511	7.5382	1.3146
Half	5.2340	0.9361	7.1182	0.9562
Hoard	6.3177	1.1185	7.2804	0.9164
nMART	5.6742	0.0472	6.0579	0.0524
tcmalloc	5.4136	0.2975	7.2075	0.4350
TLSF	4.7615	0.3585	6.8852	0.0455

Table 4.46: The execution time of CDF model with sixty-four threads

In summary, for local memory access cases, *TLSF* shows better performance, with all number of threads ranging from two to sixty-four; otherwise, *First-fit*, *Half-fit* and even *Hoard* show the worst performance when the application uses a large number of threads. For remote memory access cases, all allocator algorithms, except for *Best-fit* and *Half-fit*, show better performance. *Half-fit* shows the worst performance in a small number of thread cases ranging from two to eight, and *First-fit* and *TLSF* show the worst performance in a large number of thread cases. Regarding the performance of *nMART*, it shows the best performance for remote memory access, with all number of threads ranging from two to sixty-four. For local memory access, it also shows better performance compared to others except *TLSF*.

4.4 Summary

Developing a memory allocation algorithm is a trade-off between time efficiency and space efficiency. Many allocation algorithms supporting multi-threaded envi-

ronments use more memory space due to using multiple heaps, but they show better performance. Otherwise, memory allocators, which do not consider multi-threaded environments, waste less memory space, but they show worse performance.

The primary objectives of the thesis were to develop a new memory allocation algorithm that combines real-time and distributed shared memory architecture systems, specially, ccNUMA systems. It requires providing bounded response time with maximizing node-locality in support of the target systems. In order to provide evidences in support of the thesis hypothesis, this chapter has empirically conducted temporal behaviour analysis of memory allocation algorithms. It evaluates the longest execution time and the total execution time obtained in allocation and de-allocation operations to analyze the worst-case execution time and the average performance of algorithms in practice. The local and remote access latencies analyses were also provided to compare the efficiency of ccNUMA architecture support. Also, additional evaluation in terms of spatial behaviour and cache behaviour analysis are provided in Appendix A.

Overall, our prototype shows acceptable performance in experiments compared to all algorithms in support of single-threaded applications. Also, it showed better performance in terms of multi-threaded circumstances, especially on remote memory accesses, and shows consistent and stable performance; however it can degrade its performance if the *remote free* occurs frequently. Its overheads are not noticeable because a complete trigger condition depends on the design of application and it is only observed less than 3% in long-running applications [Larson and Krishnan, 1998].

Chapter 5

Conclusions And Future Work

Despite memory allocation algorithms being studied extensively over the last five decades, limited attention has been focused on their use in real-time systems. Many high-performance algorithms do not satisfy the requirements of real-time systems, and the few allocators supporting real-time systems do not scale well on multi-processor systems. ccNUMA architecture systems are a result of the demand of high-performance system, and they provide a systematic, scalable design. Real-time systems have also been increasing in size and complexity, with the consequence that they need computing platforms that consist of multiprocessors with many cores.

For non-real-time applications, ccNUMA architecture systems do not introduce any new problems beyond those architectures that have multiple processors. Cache coherence protocols allow application developers not to consider the characteristics of the platform. Also, the underlying operating systems provide hardware abstraction to allow developers not to be concerned with the locations where applications execute. However, application developers, who require predictability and high-performance fully need to understand the characteristics of the system and the abstractions. With storage allocation, an architecture-aware algorithm can avoid suffering from unexpected memory access latencies and exploit local node or closer nodes as often as possible without hardware-specific knowledge.

In this thesis, we have shown that many operating systems do not use an accurate measurement for the distances between nodes in a ccNUMA architecture. We have introduced a more accurate measure for node distances and established a new node

distance table with values obtained after evaluating the real node distance; this guarantees that where remote accesses are required, the access will be to the closest node possible. This provides better temporal performance. Also, it provides more transparency to real-time applications to make them aware of the hardware resources available. Application developers do not need to consider the hardware specific design to access the closest node.

In terms of the memory allocation algorithms. Multiple heaps are able to reduce lock contentions between threads and provide better temporal performance in a multi-threaded environment. A single heap with a global lock mechanism to ensure data consistency is one of the major reasons for performance degradation due to the contention for the lock. A global lock also results in poor scalability. Multiple heap scales much better than a global heap.

Search strategies, *exact-fit* mechanisms for small sizes of blocks, *good-fit* and *first-fit* mechanisms for ordinary sizes of blocks, used in the algorithm are able to provide temporal guarantees. These mechanisms with *bitmapped-fit* are able to avoid exhaustive search in the list when looking for a free block. Also, these mechanisms can be implemented in a constant execution time.

In terms of spatial behaviour, using multiple strategies for different sizes of blocks consumes less space when compared to the use of a single strategy. Splitting size ranges into fine-grains reduces internal fragmentation. Also, tracking the memory block profile prevents the *blowup* phenomenon, which resulted in unbounded space consumption. Tracking is also able to maximize node data locality on the target platform caused by the *first touch* policy of modern operating systems.

A prototype implementing our design principles has been evaluated empirically using experiments on actual hardware. Results show that the algorithm can be used by real-time applications on soft real-time systems to improve their performance. Particularly, the algorithm shows the best performance for remote access memory cases between other algorithms. Regarding spatial behaviour, Appendix A shows good memory consumption. It shows less memory consumption compared to other algorithms for multi-threaded applications such as *tcmalloc* and *Hoard*.

Overall, we have developed a ccNUMA-aware memory management algorithm

that provides good performance (amortized $\mathcal{O}(1)$) for soft real-time systems on ccNUMA architecture systems in practice. The potentially unbounded remote free operations means that the algorithm cannot be used in hard real-time systems.

5.1 Contributions

In this thesis, the major motivation is based on the argument that many memory allocation algorithms widely used in the development of both general-purpose and real-time systems on ccNUMA architecture systems will increasingly prove inefficient, as such systems become larger and more complex. In order to provide stable, high-performance, and scalability on such systems, it is necessary to recognise and fully discover the underlying hardware design; current dynamic allocation algorithms cannot achieve this and result in unexpected memory access latencies.

The key contributions of this thesis come from the development of a new ccNUMA-aware memory allocation algorithm that can be used in the implementation of high-performance real-time applications. In detail, the first of these contributions is that we present a new model that allows measuring real node distances to reflect the underlying hardware design more accurately than in previous systems. The problem of the encapsulation of the underlying architecture design, which is provided by modern operating systems, inhibits the discovery of the system resources accurately, thereby leading to making it difficult to develop predictable applications on real-time systems. The new model enables discovering and recognizing exactly how the underlying hardware is organised in order to minimize unexpected memory access latencies.

The second contributions is that we show a new ccNUMA-aware memory allocation algorithm, which ensures maximum node-locality and provides bounded response time as well as small fragmentation. Through it, we show that we can still design much better memory allocation algorithms supporting ccNUMA systems, despite there being many high-performance allocation algorithms already in existence. Also, the algorithm provides transparencies of the underlying architecture design for real-time application developers, maximizes ccNUMA node-locality to applications,

and minimizes the probability of the worst-case access to remote nodes.

Finally, existing memory allocation algorithms, which can be used to support ccNUMA systems, are analysed and shown that they do not facilitate efficient memory allocation and deallocation; however, a prototype implementation of *nMART* has shown much better performance regarding remote memory access. As there are no standard evaluation methods or tools for memory allocation algorithms, we have design evaluation methods for our experiments along with measuring the performance of remote memory access. An empirical analysis evaluates the performance of memory allocation algorithms using both real workloads and synthetic workloads. Our experimental results show that the prototype achieves its objectives of maximizing node-locality, scalability, bounded response time as well as low fragmentation.

5.2 The Hypothesis Revisited

In Chapter 1, we presented the following thesis hypothesis.

> *The ability of a dynamic storage allocation algorithm can be enhanced to meet the requirements of soft real-time systems with a small bounded execution-time on ccNUMA architecture systems.*

The main objective of dynamic storage allocation algorithms is to satisfy allocation/deallocation requests from applications. With supporting cache coherence protocols, ccNUMA architecture systems do not introduce any new problems for non-real-time dynamic storage allocation algorithms. However, for real-time applications, the algorithms require a complete understanding of the characteristics of the underlying architecture to meet their timing constraints. It is, therefore, an essential requirement to recognize and discover the underlying architecture design. Our approach exploits a new node distance table and measuring model, and a "closest node-oriented allocation" policy to reduce the latency of remote memory access. These are incorporated into the *nMART* dynamic storage allocation algorithm to improve performance and predictability on the target systems.

Our experiments have been empirically conducted using synthetic and real workloads on actual hardware. The longest execution time and the total execution time

have been measured to allow comparisons with the worst-case execution time and the average performance of *nMART* with other competing algorithms described in this thesis. In particular, the access latencies of local and remote are evaluated to compare the ccNUMA support of the target systems.

Reducing the latency of remote memory access is the most important factor to achieve better performance. Our algorithm shows good performance in experiments in support of single-threaded and multi-threaded applications. In terms of the remote latency, our approach shows consistent and stable performance among all algorithms. Overall, our algorithm is able to improve performance on the target architecture systems.

5.3 Future Work

The study presented in this thesis points to several areas for future work. Firstly, our implementation is a user-level memory allocation allocator; although we modified the Linux kernel to maximize accessing the local node and to minimize the distance when remote access is required. Our approach is based on a static node distance table, which re-measures real node distances and is used at kernel compile-time. However, we believe that these limitations can be improved by adopting a technique used in the "load average" of the *top* command in the Unix environment. This measures the processor utilization, as an instantaneous snapshot, with all demand for the processors and compiles the statistics. After that, it provides every 1, 5 and 15 minutes load-average values. Using the same approach, the new model can measure the values of real node distance between nodes every a few minutes and re-establishes a new node distance table at system run-time.

Synchronization for resource sharing is an important consideration in the design of a memory allocation algorithm supporting concurrency and scalability. Regarding synchronization techniques, many well-known solutions have been introduced, such as *wait-free* and *lock-free* algorithms; however, for real-time systems, it is necessary to consider schedulability along with synchronization. Unfortunately, most implementations of these techniques are based on CAS-primitives, which suffered from

a (potentially unbounded) number of retries. However, we believe a detailed and more thorough research of synchronization along with schedulability in the context of memory allocators is needed. The use of *wait-free* and *lock-free* algorithms would remove the necessity of the remote freeing of blocks mentioned in Section 3.4.4 which causes the potential for unbounded execution times.

Typically, it is impossible to specify the sequence of allocation and de-allocation requests of all individual applications. This makes it difficult to predict the amount of power consumed by memory operations; however, nowadays, low power consumption is also one of the important issues in real-time systems. Currently, most researches concentrate on hardware optimizations to reduce power dissipation; there are few studies on alternative approaches involving optimizations at the compiler-level, instruction-level, and source code-level [Ortiz and Santiago, 2008] [Mehta et al., 1997]. The use of these techniques can have an impact on the optimization and predictability of memory allocation. This is another area for future study.

Applications of real-time systems can be considered mission-critical. Only authorized and fully analysed applications can be executed on these systems to satisfy their constraints; however, this is a slightly different environment to embedded systems, such as smart phones. Many attack techniques have been introduced, along with their corresponding detection and prevention techniques [Polishchuk et al., 2007] [Robertson et al., 2003] [Chen et al., 2005] [Cowan et al., 2000], which suffer from additional overheads on memory operations and are difficult to analyse. We believe more thorough research on the detection and prevention techniques is needed.

Finally, many modern high-level programming languages (e.g., Java, C# and C++11) support garbage collection, which is responsible for automatic recycling of unreferenced regions of memory. Using a garbage collector is still a challenge in the real-time domain due to its unpredictable delays. However, there are now real-time garbage collectors that have adequate soft real-time performance. The integration of these with our memory allocation approach is another area of future work.

Appendix A

Additional Evaluation of Spatial and Cache Behaviour of memory Allocators

In this appendix we show the result of our experiments to determine the spatial and cache behaviour of a variety of memory allocation algorithms, including nMART.

A.1 Spatial Behaviour Analysis

In this section, we will discuss the spatial behaviour of memory allocator algorithms by comparing the total size of blocks allocated by each allocator and the total amount of virtual memory used by each allocators.

A.1.1 Total Size of Memory Blocks

The total size of memory blocks requested and provided are related to the internal fragmentation. In Section 2.2.1, we discussed how to calculate this internal fragmentation.

This experiment is executed using different real applications with test sets 1 to 3. Each application was executed 1,000 times with the same allocator repeatedly so that each application was executed 7,000 times with seven different allocators totally. In the individual executions, we measure the total size of blocks requested

and the actual total size of blocks provided to each application's execution. For instance, in test set 1 of *cfrac* in Table 4.5, the application invokes the malloc() function 1,528 times. During 1,528 function calls, we collect the size of blocks requested and provided, accumulating it as the application is executed 1,000 times. After that, we collect statistics of the average of a thousand total sizes of blocks requested and provided, calculating the internal fragmentation using equation 2.1.

cfrac **Results** Table A.1 shows the total size of blocks obtained for allocation on *cfrac* with each allocator. The block sizes of the analysis of test sets 1, 2 and 3 are given in Table A.1.

Results Analysis The following shows the effect of allocation on the internal fragmentation. Note that *Req* in tables represents the total size of blocks requested, *Prov* means the total size of blocks provided, and *Frag* denotes the internal fragmentation obtained using equation 2.1.

The total sizes of blocks for allocation are shown in Table A.1 respectively for all test sets. The internal fragmentation of *Hoard* throughout all test sets is lower than the values for other allocators, while the *Half-fit* shows the highest internal fragmentation throughout all test sets. The internal fragmentation of *Hoard* in test set 1 is 21.0% compared to 53.53% of *Half-fit*. The internal fragmentation of *Hoard* in test set 2 is 12.72% compared to 55.99% of *Half-fit*. The internal fragmentation of *Hoard* in test set 3 is 7.6% compared to 53.37% of *Half-fit*.

espresso **Results** Table A.2 shows the total size of blocks obtained for allocation on *espresso* with each allocator. The block sizes of the analysis of test sets 1, 2 and 3 are given in Table A.2.

Results Analysis The following shows the effect of allocation on the internal fragmentation:

The total sizes of blocks for allocation are shown in Table A.2 respectively for all test sets. The internal fragmentation of *tcmalloc* throughout test set 1 is lower than the values for other allocators. The internal fragmentation of *TLSF* throughout test

cfrac						
	SET 1		SET 2		SET 3	
	Req/Prov	Frag	Req/Prov	Frag	Req/Prov	Frag
Best	29,604 / 57,500	48.51%	108,899 / 199,524	45.42%	191,703 / 333,408	42.50%
First	29,604 / 57,500	48.51%	108,899 / 199,556	45.43%	191,703 / 333,440	42.51%
Half	29,604 / 63,712	53.53%	108,899 / 247,424	55.99%	191,703 / 411,136	53.37%
Hoard	29,604 / 37,472	21.00%	108,899 / 124,776	12.72%	191,703 / 207,472	7.60%
nMART	29,604 / 43,528	31.99%	108,899 / 142,712	23.69%	191,703 / 232,896	17.69%
tcmalloc	29,606 / 40,464	26.83%	108,901 / 142,840	23.76%	191,705 / 237,192	19.18%
TLSF	29,604 / 42,688	30.65%	108,899 / 142,464	23.56%	191,703 / 236,800	19.04%

Table A.1: The total block sizes requested/provided by *cfrac*

sets 2 and 3 is lower than the values for other allocators, while the *Half-fit* shows the highest internal fragmentation throughout all test sets. The internal fragmentation of *tcmalloc* in test set 1 is 8.27% compared to 45.21% of *Half-fit*. The internal fragmentation of *TLSF* in test set 2 is 6.96% compared to 44.44% of *Half-fit*. The internal fragmentation of *TLSF* in test set 3 is 3.09% compared to 38.59% of *Half-fit*.

espresso						
	SET 1		SET 2		SET 3	
	Req/Prov	Frag	Req/Prov	Frag	Req/Prov	Frag
Best	1,755,782 / 2,218,544	20.86%	5,490,249 / 6,507,076	15.63%	186,382,660 / 207,689,252	10.26%
First	1,755,782 / 2,144,128	18.11%	5,490,249 / 6,320,100	13.13%	186,382,660 / 203,939,380	8.61%
Half	1,755,782 / 3,204,352	45.21%	5,490,249 / 9,881,728	44.44%	186,382,660 / 303,505,312	38.59%
Hoard	1,693,198 / 1,916,640	11.66%	5,284,473 / 6,025,288	12.30%	182,915,092 / 199,398,696	8.27%
nMART	1,693,198 / 1,847,186	8.34%	5,284,473 / 5,724,624	7.69%	182,915,092 / 189,395,918	3.42%
tcmalloc	1,693,200 / 1,845,816	8.27%	5,284,475 / 5,828,584	9.34%	182,915,094 / 199,631,608	8.37%
TLSF	1,693,198 / 1,848,368	8.39%	5,284,473 / 5,679,488	6.96%	182,915,092 / 188,751,200	3.09%

Table A.2: The total block sizes requested/provided by *espresso*

***gawk* Results** Table A.3 shows the total size of blocks obtained for allocation on *gawk* with each allocator. The block sizes of the analysis of test sets 1, 2 and 3 are given in Table A.3.

Results Analysis The following shows the effect of allocation on the internal fragmentation:

The total sizes of blocks for allocation are shown in Table A.3 respectively for all

test sets. The internal fragmentation of *TLSF* throughout test set 1 is lower than the values for other allocators. The internal fragmentation of *Hoard* throughout test sets 2 and 3 is lower than the values for other allocators, while the *Half-fit*, again, shows the highest internal fragmentation throughout all test sets. The internal fragmentation of *TLSF* in test set 1 is 9.84% compared to 41.65% of *Half-fit*. The internal fragmentation of *Hoard* in test set 2 is 11.61% compared to 45.53% of *Half-fit*. The internal fragmentation of *Hoard* in test set 3 is 11.16% compared to 45.94% of *Half-fit*.

gawk						
	SET 1		SET 2		SET 3	
	Req/Prov	Frag	Req/Prov	Frag	Req/Prov	Frag
Best	385,298 / 490,272	21.41%	6,255,070 / 9,495,428	34.13%	12,034,858 / 18,274,412	34.14%
First	385,298 / 491,008	21.53%	6,255,070 / 9,431,028	33.68%	12,034,858 / 18,716,396	35.70%
Half	385,298 / 660,320	41.65%	6,255,070 / 11,482,944	45.53%	12,034,858 / 22,260,736	45.94%
Hoard	373,559 / 442,064	15.50%	6,245,812 / 7,065,832	11.61%	12,025,925 / 13,536,112	11.16%
nMART	350,243 / 397,800	11.96%	5,651,828 / 6,669,459	15.26%	15,654,509 / 18,348,213	14.68%
tcmalloc	350,245 / 414,352	15.47%	5,651,830 / 6,974,440	18.96%	10,858,671 / 13,516,672	19.66%
TLSF	350,243 / 388,464	9.84%	5,651,828 / 6,607,712	14.47%	15,654,509 / 18,316,288	14.53%

Table A.3: The total block sizes requested/provided by *gawk*

***p2c* Results** Table A.4 shows the total size of blocks obtained for allocation on *p2c* with each allocator. The block sizes of the analysis of test sets 1, 2 and 3 are given in Table A.4.

Results Analysis The following shows the effect of allocation on the internal fragmentation. Note that all test sets produced the same values of the total size of blocks as well as the internal fragmentation, but test set 3 of *p2c* produced different values during the experiment. *R_STD* in the table for test set 3 means the standard deviation of the total size of block requests, and *P_STD* denotes the standard deviation of the total size of blocks provided:

The total sizes of blocks for allocation are shown in Table A.4 respectively for all test sets. The internal fragmentation of *Hoard* throughout all test sets is lower than the values for other allocators, while the *Half-fit* shows the highest internal fragmentation throughout all test sets. The internal fragmentation of *Hoard* in test

set 1 is 6.42% compared to 47.13% of *Half-fit*. The internal fragmentation of *Hoard* in test set 2 is 5.86% compared to 46.45% of *Half-fit*. The internal fragmentation of *Hoard* in test set 3 is 3.02% compared to 52.26% of *Half-fit*.

In summary, *Hoard* shows the best efficiency on the internal fragmentation throughout *cfrag*, *gawk* and *p2c* application, whereas *TLSF* shows less internal fragmentation for *espresso*. On the whole, *nMART* shows better efficiency on the internal fragmentation throughout all applications compared to conventional algorithms. Also, it shows better efficiency on the fragmentation throughout *esspresso* and *gawk* application compared to *Hoard* and *tcmalloc*.

A.1.2 Total Amount of Virtual Memory Usage

In general, memory allocators request more memory space from the underlying operating system when remaining memory space is not enough to satisfy applications' requests, even if the total amount of memory unused is larger than the requested size because of external fragmentation. The external fragmentation affects the actual total amount of memory used. The total amount of memory used can be determined indirectly via monitoring the virtual memory usage.

Typically, it is harder to find out the external fragmentation in practice than finding the internal fragmentation because it is necessary to analyze virtual memory usage. In Linux, there are two approaches to determine how much memory is used by a process. We have decided to use the *proc* special file system, which provides information about all running processes, rather than using some APIs such as *getrusage()* provided by the OS; this is because using APIs incurs an overhead for the application and it may miss some of the memory allocated if the application allocats memory through the mmap() system call. Through */proc/pid/status* (*pid* is the process ID) we can collect information regarding memory, especially peak virtual memory and data area usage, and current virtual memory usage, and so on.

In order to obtain the peak virtual memory usage and data area usage, we have executed test sets 1 to 4 with the application provided in appendix D.1.3. The application iteratively read the */proc/pid/status* file every 0.1 msec and prints the collected and parsed results. We accumulate the results in another log file using a

p2c							
	SET 1		SET 2		SET 3		
	Req/Prov	fragment	Req/Prov	fragment	Req/Prov	fragment	R_STD/P_STD
Best	166,334 / 211,032	21.18%	299,441 / 382,420	21.70%	1,497,258.32 / 2302056.00	34.96%	1.09 / 0.00
First	166,334 / 211,064	21.19%	299,441 / 383,524	21.92%	1,497,258.22 / 2019864.00	25.87%	0.92 / 0.00
Half	166,334 / 314,592	47.13%	299,441 / 559,232	46.45%	1,497,258.22 / 3136576.00	52.26%	0.91 / 0.00
Hoard	166,034 / 177,432	6.42%	298,792 / 317,400	5.86%	1,496,159.22 / 1542800.45	3.02%	0.92 / 1.84
nMART	166,034 / 181,474	8.51%	298,792 / 323,765	7.71%	1,496,159.27 / 1763320.30	15.15%	0.92 / 4.78
tcmalloc	166,036 / 179,464	7.48%	298,794 / 323,648	7.68%	1,496,161.21 / 1750744.85	14.54%	0.90 / 3.59
TLSF	166,034 / 184,208	9.87%	298,792 / 333,728	10.47%	1,496,159.22 / 1776850.69	15.80%	0.92 / 11.04

Table A.4: The total block sizes requested/provided by *p2c*

redirection of standard output. For the experiment, we need to modify some of the application's source code because some test sets finish their functionalities before reading the *status* file in *proc* often enough. In order to wait enough time for the results to be collected, we have just added the *sleep()* function to wait three seconds at the end of the *main()* function of each application of the test sets.

The experiment is executed using different real applications with test sets 1 to 4. Each application was executed 100 times with the same allocator repeatedly so that each application was executed 700 times with seven different allocators totally. In the individual execution, we read the */proc/pid/status* file every 0.1 msec to determine the peak virtual memory usage in each application execution practically.

Test Set 1 **Results** Table A.5 shows the virtual memory usage obtained for test set 1 with each allocator. The memory usage of the *cfrac*, *espresso*, *gawk* and *p2c* analysis is given in Table A.5.

Results Analysis The following shows the behaviour of allocation on the total amount of virtual memory usage. Note that *Peak* indicates the peak virtual memory usage, which is the peak of the aggregate amount of virtual memory usage of data, stack and code segments. *Data* means the size of the data segment of application. The unit of the table is Kilobyte (kB):

- *cfrac*: The peak and data of virtual memory usage of *cfrac* with each allocator are shown in Table A.5. The amount of virtual memory of *First-fit* and *Best-fit* in the set is lower than the values of other allocators. In the peak usage case, the amount of memory of *Best-fit* and *First-fit* in the set is 8,904 kB compared to 18,832 kB of *tcmalloc*, which has the highest amount of peak virtual memory usage. In the data usage case, the amount of memory of *Best-fit* and *First-fit* is 136 kB compared to 2,528 kB of *tcmalloc*.

- *espresso*: The peak and data of virtual memory usage of *espresso* with each allocator are shown in Table A.5. The amount of virtual memory of *First-fit* and *Best-fit* in the set is lower than the values for other allocators. In the

peak usage case, the amount of *Best-fit* and *First-fit* in test set 1 is 6,480 kB compared to 19,000 kB of *tcmalloc*, which has the highest amount of peak virtual memory usage. In the data usage case, the amount of memory of *Best-fit* and *First-fit* is 128 kB compared to 2,588 kB of *Hoard*.

- *gawk*: The peak and data of virtual memory usage of *gawk* with each allocator are shown in Table A.5. The amount of virtual memory of *First-fit* and *Best-fit* in the set is lower than the values for other allocators. In the peak usage case, the amount of *Best-fit* and *First-fit* in the set is 13,140 kB compared to 22,844 kB of *tcmalloc*, which has the highest amount of virtual memory. In the data usage case, the amount of memory of *Best-fit*, *First-fit*, and *TLSF* is 392 kB compared to 3,056 kB of *Hoard*.

- *p2c*: The peak and data of virtual memory usage of *p2c* with each allocator are shown in Table A.5. The amount of virtual memory of *First-fit* and *Best-fit* in the set is lower than the values for other allocators. In the peak usage case, the amount of *Best-fit* and *First-fit* in test set 1 is 7,040 kB compared to 19,416 kB of *tcmalloc*, which has the highest amount of virtual memory. In the data usage case, the amount of memory of *Best-fit* and *First-fit* is 300 kB compared to 2,564 kB of *tcmalloc*.

SET 1								
	cfrac		espresso		gawk		p2c	
	Peak	Data	Peak	Data	Peak	Data	Peak	Data
Best	8,904	136	6,480	128	13,140	392	7,040	300
First	8,904	136	6,480	128	13,140	392	7,040	300
Half	8,912	140	6,500	144	13,216	464	7,124	380
Hoard	17,528	1,432	18,848	2,588	21,068	3,056	18,748	2,104
nMART	9,048	276	6,532	176	13,284	532	7,180	436
tcmalloc	18,832	2,528	19,000	2,532	22,844	2,560	19,416	2,564
TLSF	8,928	156	6,492	136	13,144	392	7,048	304

Table A.5: The size of virtual memory provided for test set 1

Test Set 2 **Results** Table A.6 shows the virtual memory usage obtained for test set 2 with each allocator. The memory usage of the *cfrac*, *espresso*, *gawk* and *p2c* analysis is given in Table A.6.

Results Analysis The following shows the behaviour of allocation on the total amount of virtual memory usage. The unit of the table is Kilobyte (kB):

- *cfrac*: The peak and data of virtual memory usage of *cfrac* with each allocator are shown in Table A.6. The amount of virtual memory of *First-fit* and *Best-fit* in the set is lower than the values of other allocators. In the peak usage case, the amount of memory of *Best-fit* and *First-fit* in the set is 9,064 kB compared to 18,832 kB of *tcmalloc*, which has the highest amount of peak virtual memory usage. In the data usage case, the amount of memory of *Best-fit* and *First-fit* is 296 kB compared to 2,528 kB of *tcmalloc*.

- *espresso*: The peak and data of virtual memory usage of *espresso* with each allocator are shown in Table A.6. The amount of virtual memory of *First-fit* and *Best-fit* in the set is lower than the values for other allocators. In the peak usage case, the amount of *Best-fit* and *First-fit* in test set 2 is 6,472 kB compared to 19,000 kB of *tcmalloc*, which has the highest amount of peak virtual memory usage. In the data usage case, the amount of memory of *Best-fit* and *First-fit* is 120 kB compared to 2,844 kB of *Hoard*.

- *gawk*: The peak and data of virtual memory usage of *gawk* with each allocator are shown in Table A.6. The amount of virtual memory of *TLSF* in the set is lower than the values for other allocators. In the peak usage case, the amount of *TLSF* is 16,976 kB compared to 26,944 kB of *tcmalloc*, which has the highest amount of virtual memory. In the data usage case, the amount of memory of *TLSF* is 4,224 kB compared to 6,772 kB of *Hoard*.

- *p2c*: The peak and data of virtual memory usage of *p2c* with each allocator are shown in Table A.6. The amount of virtual memory of *Best-fit* in the set is lower than the values for other allocators. In the peak usage case, the

amount of *Best-fit* is 7,140 kB compared to 19,416 kB of *tcmalloc*, which has the highest amount of virtual memory. In the data usage case, the amount of memory of *Best-fit* is 400 kB compared to 2,564 kB of *tcmalloc*.

SET 2								
	cfrac		espresso		gawk		p2c	
	Peak	Data	Peak	Data	Peak	Data	Peak	Data
Best	9,064	296	6,472	120	17,152	4,404	7,140	400
First	9,064	296	6,472	120	17,140	4,392	7,144	404
Half	9,088	316	6,488	132	17,892	5,140	7,244	500
Hoard	17,720	1,624	19,100	2,844	24,852	6,772	18,812	2,168
nMART	9,520	748	6,512	156	18,356	5,604	7,400	656
tcmalloc	18,832	2,528	19,000	2,532	26,944	6,656	19,416	2,564
TLSF	9,080	308	6,484	128	16,976	4,224	7,148	404

Table A.6: The size of virtual memory provided for test set 2

***Test Set 3* Results** Table A.7 shows the virtual memory usage obtained for test set 3 with each allocator. The memory usage of the *cfrac*, *espresso*, *gawk* and *p2c* analysis is given in Table A.7.

Results Analysis The following shows the behaviour of allocation on the total amount of virtual memory usage. The unit of the table is Kilobyte (kB):

- *cfrac*: The peak and data of virtual memory usage of *cfrac* with each allocator are shown in Table A.7. The amount of virtual memory of *First-fit* and *Best-fit* in the set is lower than the values of other allocators. In the peak usage case, the amount of memory of *Best-fit* and *First-fit* in the set is 9,212 kB compared to 18,832 kB of *tcmalloc*, which has the highest amount of peak virtual memory usage. In the data usage case, the amount of memory of *Best-fit* and *First-fit* is 444 kB compared to 2,528 kB of *tcmalloc*.
- *espresso*: The peak and data of virtual memory usage of *espresso* with each allocator are shown in Table A.7. The amount of virtual memory of *First-fit*

and *Best-fit* in the set is lower than the values for other allocators. In the peak usage case, the amount of *Best-fit* and *First-fit* in test set 3 is 6,844 kB compared to 20,436 kB of *Hoard*, which has the highest amount of peak virtual memory usage. In the data usage case, the amount of memory of *Best-fit* and *First-fit* is 492 kB compared to 4,140 kB of *Hoard*.

- *gawk*: The peak and data of virtual memory usage of *gawk* with each allocator are shown in Table A.7. The amount of virtual memory of *TLSF* in the set is lower than the values for other allocators. In the peak usage case, the amount of *TLSF* is 24,080 kB compared to 33,100 kB of *tcmalloc*, which has the highest amount of virtual memory. In the data usage case, the amount of memory of *TLSF* is 11,328 kB compared to 13,872 kB of *Hoard*.

- *p2c*: The peak and data of virtual memory usage of *p2c* with each allocator are shown in Table A.7. The amount of virtual memory of *Best-fit* and *First-fit* in the set is lower than the values for other allocators. In the peak usage case, the amount of *Best-fit* and *First-fit* is 7,592 kB compared to 19,416 kB of *tcmalloc*, which has the highest amount of virtual memory. In the data usage case, the amount of memory of *Best-fit* and *First-fit* is 852 kB compared to 2,564 kB of *tcmalloc*.

SET 3								
	cfrac		espresso		gawk		p2c	
	Peak	Data	Peak	Data	Peak	Data	Peak	Data
Best	9,212	444	6,844	492	24,588	11,840	7,592	852
First	9,212	444	6,844	492	24,568	11,820	7,592	852
Half	9,244	472	6,968	612	26,500	13,748	7,780	1,036
Hoard	17,848	1,752	20,436	4,140	31,884	13,872	19,196	2,552
nMART	9,968	1,196	7,120	764	27,664	12,912	8,280	1,236
tcmalloc	18,832	2,528	20,024	3,556	33,100	12,816	19,416	2,564
TLSF	9,228	456	6,860	504	24,080	11,328	7,600	856

Table A.7: The size of virtual memory provided for test set 3

***Test Set 4* Results** Table A.8 shows the virtual memory usage obtained for test set 4 with each allocator. The memory usage of the *cache-scratch*, *cache-thrash*, *larson* and *shbench* analysis is given in Table A.8.

Results Analysis The following shows the behaviour of allocation on the total amount of virtual memory usage. The unit of the table is Kilobyte (kB):

- *cache-scratch*: The peak and data of virtual memory usage of *cache-scratch* with each allocator are shown in Table A.8. The amount of virtual memory of *First-fit* and *Best-fit* in the set is lower than the values of other allocators. In the peak usage case, the amount of memory of *Best-fit* and *First-fit* in the set is 268,516 kB compared to 283,604 kB of *Hoard*, which has the highest amount of peak virtual memory usage. In the data usage case, the amount of memory of *Best-fit* and *First-fit* is 262,356 kB compared to 266,068 kB of *Hoard*.

- *cache-thrash*: The peak and data of virtual memory usage of *cache-thrash* with each allocator are shown in Table A.8. The amount of virtual memory of *First-fit* and *Best-fit* in the set is lower than the values for other allocators. In the peak usage case, the amount of *Best-fit* and *First-fit* in test set 4 is 268,516 kB compared to 283,604 kB of *Hoard*, which has the highest amount of peak virtual memory usage. In the data usage case, the amount of memory of *Best-fit* and *First-fit* is 262,356 kB compared to 267,476 kB of *Hoard*.

- *larson*: The peak and data of virtual memory usage of *larson* with each allocator are shown in Table A.8. The amount of virtual memory of *Best-fit* in the set is lower than the values for other allocators. In the peak usage case, the amount of *Best-fit* is 414,420 kB compared to 541,200 kB of *Hoard*, which has the highest amount of virtual memory. In the data usage case, the amount of memory of *Best-fit* is 406,084 kB compared to 525,324 kB of *Hoard*.

- *shbench*: The peak and data of virtual memory usage of *shbench* with each allocator are shown in Table A.8. The amount of virtual memory of *Best-fit* and *First-fit* in the set is lower than the values for other allocators. In the

peak usage case, the amount of *Best-fit* and *First-fit* is 39,320 kB compared to 54,436 kB of *Hoard*, which has the highest amount of virtual memory. In the data usage case, the amount of memory of *Best-fit* and *First-fit* is 33,160 kB compared to 38,308 kB of *Hoard*.

	cache-scratch		cache-thrash		larson		shbench	
	Peak	Data	Peak	Data	Peak	Data	Peak	Data
Best	268,516	262,356	268,516	262,356	414,420	406,084	39,320	33,160
First	268,516	262,356	268,516	262,356	507,624	499,308	39,320	33,160
Half	268,532	262,368	268,528	262,364	498,856	490,512	39,472	33,308
Hoard	283,604	266,068	283,604	267,476	541,200	525,324	54,436	38,308
nMART	268,536	262,372	268,536	262,372	418,604	409,576	39,428	33,264
tcmalloc	281,068	264,796	281,068	264,796	511,704	502,600	51,580	35,308
TLSF	268,536	262,372	268,524	262,360	417,752	409,800	39,348	33,184

Table A.8: The size of virtual memory provided for test set 4

In summary, a type of memory allocator, which exploits multiple heaps with segregated size class, used more memory space, while memory allocators using a global heap consumed less memory space. In the *Hoard* case, it has the highest usage of memory space under very memory-intensive applications. Overall, *nMART* used less memory space throughout all applications compared to *tcmalloc* and *Hoard*, which exploits multiple heaps with segregated size classes. But, it consumed more memory space compared to others, which exploit a global heap.

A.2 Cache Behaviour Analysis

In this section, we will analyze cache behaviour of applications with each allocator. In particular, active false-sharing analysis has been conducted using *cache-thrash* and passive false-sharing analysis has been conducted using *cache-scratch*. Such applications, provided by [Berger et al., 2000], produced contention caused by the cache-coherent protocol.

Unlike the above experiments, the experiment for test set 4 has been conducted

using *perf* to analyze cache behaviour, processor cycles, and instructions executed. We have employed an option provided to execute each application repeatedly and to produce average and the standard deviation of events for results. Each application was executed 400 times with the same allocator repeatedly under the option so that each application was executed 2,800 times with seven different allocators totally. As the result, we collected the average processor cycles and instructions executed, L1 cache misses and L3 cache misses, and time elapsed produced by *perf*.

***cache-scratch* Results** Table A.9 gives analysis of the average number of processor cycles obtained and the average number of instructions executed, L1 cache misses and L3 cache misses, and the average time elapsed on *cache-scratch* with each allocator.

Results Analysis The following shows the behaviour of *cache-scratch* on the performance events; note that *L1/L3 misses* in tables represent the L1 cache misses ratio and L3 cache misses ratio. The time elapsed means the time spent executing the application. The values in parenthesis denote the standard deviation for each element:

- Processor cycles: The average number of processor cycles for each application execution is shown in Table A.9 for test set 4. The number of processor cycles of *nMART* is lower than the values for other allocators, while *tcmalloc* consumed the highest processor cycles. The number of processor cycles of *nMART* in test set 4 is 19,189,732,268 compared to 155,372,798,443 of *tcmalloc*.

- Instructions: The average values of the number of instructions executed were uniformly distributed compared to the number of processor cycles. The number of executed instructions of *nMART* is lower than the values of other allocators, while *tcmalloc* executed the highest instructions. The number of executed instructions of *nMART* is 15,422,955,446 compared to 15,774,177,620 of *tcmalloc*.

- L1/L3 misses: The L1 cache misses ratio of *Hoard* and nMART is lower than

the values of other allocators, while the ratio of *tcmalloc* is higher than others as well as the L3 cache misses ratio. The L1 cache misses ratio of *Hoard* and *nMART* is 0.03% compared to 0.87% of *tcmalloc* and the L3 cache misses ratio of *nMART* is 3.25% compared to 23.40% of *tcmalloc*.

- Time elapsed: The average time elapsed of *nMART*, showing the best cache efficiency, is lower than the values of other allocators, while the average time elapsed of *tcmalloc* is higher than others. The average time elapsed of *nMART* is 1.21s compared to 6.53s of *tcmalloc*.

cache-scratch				
	Processor cycles	Instructions	L1/L3 misses	time elapsed
Best	30,482,111,629 (±0.30%)	15,713,624,520 (±0.06%)	0.20% / 16.73%	1.69 (±0.23%)
First	35,927,307,175 (±0.33%)	15,747,837,791 (±0.06%)	0.28% / 20.51%	1.98 (±0.30%)
Half	35,193,500,238 (±0.33%)	15,681,244,019 (±0.07%)	0.29% / 21.24%	2.08 (±0.34%)
Hoard	20,517,658,433 (±0.05%)	15,429,706,988 (±0.03%)	0.03% / 3.39%	1.25 (±0.07%)
nMART	19,189,732,268 (±0.05%)	15,422,955,446 (±0.03%)	0.03% / 3.25%	1.21 (±0.06%)
tcmalloc	155,372,798,443 (±0.18%)	15,774,177,620 (±0.03%)	0.87% / 23.40%	6.53 (±0.23%)
TLSF	33,933,050,107 (±0.27%)	15,666,639,647 (±0.05%)	0.25% / 17.91%	1.85 (±0.22%)

Table A.9: The states of event counters for *cache-scratch* in test set 4

***cache-thrash* Results** Table A.10 gives analysis of the average number of processor cycles obtained and the average number of instructions executed, L1 cache misses and L3 cache misses, and the average time elapsed on *cache-thrash* with each allocator.

Results Analysis The following shows the behaviour of *cache-thrash* on the performance events:

- Processor cycles: The average of the number of processor cycles for each application execution is shown in Table A.10 for test set 4. The number of processor cycles of *nMART* is lower than the values for other allocators, while *tcmalloc* consumed the highest processor cycles relatively. The number of processor cycles of *nMART* in test set 4 is 19,213,068,983 compared to 156,506,009,375 of *tcmalloc*.

- Instructions: The average values of the number of instructions executed were uniformly distributed compared to the number of processor cycles relatively. The number of executed instructions of *Hoard* is lower than the values of other allocators, while *Half-fit* executed the highest instructions. The number of executed instructions of *Hoard* is 15,431,756,220 compared to 15,808,405,681 of *Half-fit*.

- L1/L3 misses: The L1 cache misses ratio of *nMART* is lower than the values of other allocators, while the ratio of *tcmalloc* is higher than others as well as the L3 cache misses ratio. The L1 cache misses ratio of *nMART* is 0.03% compared to 0.86% of *tcmalloc* and the L3 cache misses ratio of *nMART* is 3.27% compared to 23.18% of *tcmalloc*.

- Time elapsed: The average time elapsed of *nMART*, showing the best cache efficiency, is lower than the values of other allocators, while the average time elapsed of *tcmalloc* is higher than others. The average time elapsed of *nMART* is 1.20s compared to 6.60s of *tcmalloc*.

cache-thrash				
	Processor cycles	Instructions	L1/L3 misses	time elapsed
Best	34,644,781,603 (±0.35%)	15,651,870,681 (±0.07%)	0.25% / 19.87%	2.07(±0.38%)
First	36,393,746,791 (±0.32%)	15,656,292,856 (±0.08%)	0.28% / 22.25%	2.00 (±0.32%)
Half	39,006,174,358 (±0.36%)	15,808,405,681 (±0.08%)	0.32% / 20.13%	2.12 (±0.36%)
Hoard	20,458,064,897 (±0.13%)	15,431,756,220 (±0.03%)	0.05% / 5.04%	1.47 (±0.40%)
nMART	19,213,068,983 (±0.13%)	15,435,817,243 (±0.03%)	0.03% / 3.27%	1.20 (±0.38%)
tcmalloc	156,506,009,375 (±0.20%)	15,781,686,255 (±0.03%)	0.86% / 23.18%	6.60 (±0.25%)
TLSF	31,333,536,707 (±0.34%)	15,555,412,372 (±0.05%)	0.22% / 18.64%	1.97 (±0.42%)

Table A.10: The states of event counters for *cache-thrash* in test set 4

***larson* Results** Table A.11 gives analysis of the average number of processor cycles obtained and the average number of instructions executed, L1 cache misses and L3 cache misses, and the average time elapsed on *larson* with each allocator.

Results Analysis The following shows the behaviour of *larson* on the performance events:

- Processor cycles: The average number of processor cycles for each application execution is shown in Table A.11 for test set 4. The number of processor cycles of *tcmalloc* is lower than the values for other allocators, while *Best-fit* consumed the highest processor cycles. The number of processor cycles of *tcmalloc* in test set 4 is 28,414,722,204 compared to 81,165,969,173 of *Best-fit*.

- Instructions: The average values of the number of instructions executed were uniformly distributed compared to the number of processor cycles relatively. The number of executed instructions of *Hoard* is lower than the values of other allocators, while *Best-fit* executed the highest instructions. The number of executed instructions of *Hoard* is 915,019,352 compared to 33,908,239,390 of *Best-fit*.

- L1/L3 misses: The L1 cache misses ratio of *First-fit* is lower than the values of other allocators, while the ratio of *Hoard* is higher than others. The L3 cache misses ratio of *Hoard* is lower than the values of other allocators, while the ratio of *Best-fit* is higher than others. The L1 cache misses ratio of *First-fit* is 1.00% compared to 8.99% of *Hoard* and the L3 cache misses ratio of *Hoard* is 3.17% compared to 38.52% of *Best-fit*.

- Time elapsed: The average time elapsed of *Half-fit* is lower than the values of other allocators, while the average time elapsed of *Best-fit* is higher than others. The average time elapsed of *Half-fit* is 6.36s compared to 29.58s of *Best-fit*.

***shbench* Results** Table A.12 gives analysis of the average number of processor cycles obtained and the average number of instructions executed, L1 cache misses and L3 cache misses, and the average time elapsed on *shbench* with each allocator.

Results Analysis The following shows the behaviour of *shbench* on the performance events:

- Processor cycles: The average of the number of processor cycles for each application execution is shown in Table A.12 for test set 4. The number of processor

larson				
	Processor cycles	Instructions	L1/L3 misses	time elapsed
Best	81,165,969,173 (±2.16%)	33,908,239,390 (±4.39%)	4.19% / 38.52%	29.58 (±0.23%)
First	34,028,174,151 (±0.04%)	17,162,890,262 (±0.16%)	1.00% / 16.10%	8.20 (±0.02%)
Half	30,346,204,027 (±0.03%)	4,180,181,485 (±0.27%)	1.20% / 6.94%	6.36 (±0.01%)
Hoard	28,574,320,199 (±0.28%)	915,019,352 (±0.31%)	8.99% / 3.17%	6.47 (±0.05%)
nMART	30,393,247,975 (±0.08%)	19,689,032,680 (±0.15%)	1.01% / 11.97%	7.07 (±0.02%)
tcmalloc	28,414,722,204 (±0.03%)	6,871,857,910 (±0.29%)	2.96% / 12.81%	6.55 (±0.01%)
TLSF	30,153,147,298 (±0.11%)	19,489,664,548 (±0.13%)	1.02% / 12.01%	7.10 (±0.05%)

Table A.11: The states of event counters for *larson* in test set 4

cycles of *tcmalloc* is lower than the values for other allocators, while *Half-fit* consumed the highest processor cycles relatively. The number of processor cycles of *tcmalloc* in test set 4 is 501,271,115 compared to 25,105,925,676 of *Half-fit*.

- Instructions: The average values of the number of instructions executed were uniformly distributed compared to the number of processor cycles relatively. The number of executed instructions of *tcmalloc* is lower than the values of other allocators, while *TLSF* executed the highest instructions. The number of executed instructions of *tcmalloc* is 792,931,505 compared to 6,295,724,312 of *TLSF*.

- L1/L3 misses: The L1 cache misses ratio of *nMART* is lower than the values of other allocators, while the ratio of *TLSF* is higher than others. The L3 cache misses ratio of *Hoard* is lower than the values of other allocators, while the ratio of *Half-fit* is higher than others. The L1 cache misses ratio of *nMART* is 0.02% compared to 3.07% of *TLSF* and the L3 cache misses ratio of *Hoard* is 2.62% compared to 27.62% of *Half-fit*.

- Time elapsed: The average time elapsed of *tcmalloc* is lower than the values of other allocators, while the average time elapsed of *Half-fit* is higher than others. The average time elapsed of *tcmalloc* is 0.35s compared to 3.96s of *Half-fit*.

shbench				
	Processor cycles	Instructions	L1/L3 misses	time elapsed
Best	20,373,588,162 (±0.18%)	4,969,045,856 (±0.26%)	0.74% / 25.92%	3.25 (±0.10%)
First	19,972,972,141 (±0.23%)	4,806,444,040 (±0.29%)	0.77% / 25.94%	3.20 (±0.13%)
Half	25,105,925,676 (±0.57%)	5,551,456,633 (±0.51%)	0.73% / 27.62%	3.96 (±0.35%)
Hoard	1,675,713,167 (±0.35%)	1,458,146,485 (±0.17%)	1.39% / 2.62%	0.51 (±0.28%)
nMART	960,454,183 (±0.21%)	1,522,897,178 (±0.2%)	0.02% / 3.93%	0.40 (±0.25%)
tcmalloc	501,271,115 (±0.17%)	792,931,505 (±0.07%)	0.07% / 3.99%	0.35 (±0.07%)
TLSF	15,817,495,323 (±0.22%)	6,295,724,312 (±0.30%)	3.07% / 21.00%	2.70 (±0.22%)

Table A.12: The states of event counters for *shbench* in test set 4

In summary, *nMART* shows the best performance throughout cache behaviour benchmarks regarding the false sharing problem. Also, *Hoard* avoids inducing false sharing so its cache miss rate is lower than others, but it consumes more memory space (as seen in Appendix A.1.2). Others induced some false sharing both actively and passively. In the *shbench* case, *nMART*, *Hoard* and *tcmalloc* show a lower rate of cache misses, except in the L1 case of *Hoard*. In larson, *Hoard* shows the worst cache misses for the L1 cache, while it shows a lower cache miss rate than others for the L3 cache.

Appendix B

Additional Evaluation of the Revised Node Distance Tables

B.1 Implementation of Node Distance tables in Linux

This appendix evaluates the implementation of the new metric for discovering node distances, discussed in Chapter 3.3.1, and discusses how much the new metric on a modified Linux kernel can improve the performance compared with the original kernel.

In the source of the 3.0.4 version of Linux [Linus Torvalds, 2011], the initialization of the node distance table (called the *numa_distance* array in the kernel) occurs at boot time by *numa_alloc_distance()* and *numa_set_distance()*. The *node_distance()* macro returns one of the element's values, the cost between the start node and the end node, which is received as its parameters, to the table. The actual work is delegated to the *__node_distance()* function.

Reading the node distance table

node_distance() is invoked by *find_next_best_node()*, *find_near_online_node()* and *node_read_distance()*. These caller functions pick up the next closest node or value of the cost when the kernel initialize an array of *zonelist* elements in *pg_data_t*.

Source 25 arch/x86/mm/numa.c

```
458 int __node_distance(int from, int to)
459 {
460   if (from >= numa_distance_cnt || to >= numa_distance_cnt)
461     return from == to ? LOCAL_DISTANCE : REMOTE_DISTANCE;
462   return numa_distance[from * numa_distance_cnt + to];
463 }
```

Building fallback lists

In the *build_zonelists()* function (shown below), there is a large external loop from line 3025 to line 3049 which works through all node zones. In the process, it attempts to find the next closest node in terms of the information from the *numa_distance* array, appending to the *zonelist* as a fallback list.

Source 26 linux/mm/page_alloc.c

```
3000 static void build_zonelists(pg_data_t *pgdat)
3001 {

3025   while ((node = find_next_best_node(local_node, &used_mask)) >= 0) {
3026     int distance = node_distance(local_node, node);

3045     if (order == ZONELIST_ORDER_NODE)
3046       build_zonelists_in_node_order(pgdat, node);
3047     else
3048       node_order[j++] = node; /* remember order */
3049   }
3050

3057 }
```

The *fallback* list is used to choose an alternative memory region when it cannot find a free area within any of the three local zones: ZONE_HIGHMEM, ZONE_DMA32, and ZONE_DMA. The fallback list entries are ordered by means of the type of memory region. The actual work is delegated to the *build_zonelists_node()* function invoked by *build_zonelists_in_node_order()*, as shown in the following code. It builds allocation fallback zone lists, adding all populated zones of a node to the *zonelist*, finally.

Source 27 linux/mm/page_alloc.c

```
2673 static int build_zonelists_node(pg_data_t *pgdat, struct zonelist *zonelist,
2674                                 int nr_zones, enum zone_type zone_type)
2675 {
2676   struct zone *zone;
2677
2678   BUG_ON(zone_type >= MAX_NR_ZONES);
2679   zone_type++;
2680
2681   do {
2682     zone_type--;
2683     zone = pgdat->node_zones + zone_type;
2684     if (populated_zone(zone)) {
2685             zoneref_set_zone(zone,
2686                     &zonelist->_zonerefs[nr_zones++]);
2687             check_highest_zone(zone_type);
2688     }
2689
2690   } while (zone_type);
2691   return nr_zones;
2692 }
```

Allocation page frames

Each memory zone contains the information of page frames that are held in a centralized array managed by the Buddy algorithm, which allocates memory only in the size of powers of 2. These page frames will be picked up by the kernel page frame allocator called zone allocator. A set of page frames is handled by the *alloc_pages()* macro, which invokes *alloc_pages_current()*. The *alloc_pages_current()* function allocates a page from the kernel page pool. The actual work is delegated to *__alloc_pages_nodemask()*, which is the main function of the buddy system because it deals with the core features of allocation.

Source 28 linux/mm/page_alloc.c

```
2234 struct page *
2235 __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
2236                         struct zonelist *zonelist, nodemask_t *nodemask)
2237 {


2251   /*
2252    * Check the zones suitable for the gfp_mask contain at least one
2253    * valid zone. It's possible to have an empty zonelist as a result
2254    * of GFP_THISNODE and a memoryless node
2255    */
2256   if (unlikely(!zonelist->_zonerefs->zone))
2257     return NULL;
```

```
2269   /* First allocation attempt */
2270   page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL, nodemask, order,
2271     zonelist, high_zoneidx, ALLOC_WMARK_LOW|ALLOC_CPUSET,
2272     preferred_zone, migratetype);
2273   if (unlikely(!page))
2274     page = __alloc_pages_slowpath(gfp_mask, order,
2275       zonelist, high_zoneidx, nodemask,
2276       preferred_zone, migratetype);

2280   return page;
2281 }
```

Simply put, the *get_page_from_freelist()* function returns the number of pages to satisfy the allocation request of the fresh memory area by going through the zonelist. It is looking for a zone with enough free memory space. However, if too little memory is available, the kernel loops over all zones in the fallback list and wakes up the swapping daemon called *kswapd* in the *__alloc_pages_slowpath()* function.

These values are firstly collected at the system boot by *build_all_zonelists()*, which splits up the memory according to the zone types: ZONE_DMA, ZONE_NORMAL, and ZONE_HIGHMEM. The *build_all_zonelists()* function invokes *__build_all_zonelist()*, which fulfils data into the *zonelists* data structure, and the *build_zonelists()* function is called the *__build_all_zonelist()* function inside. *build_zonelists()* ensures linking all memory regions, which will be taken for memory allocation in order.

As seen from the source code of the kernel, the *numa_distance* array stores the values of the node distance from the SLIT, being used when page frames are served. Therefore, we are simply able to achieve our objectives by modifying the array.

B.2 Measuring Node Distances

We have reported on two experiments to measure the actual time taken to access measures from various nodes in two distinguishing ccNUMA platforms described in Chapter 4.1, and established new node distance tables based on the experiments.

A four nodes-based ccNUMA architecture system

To measure real node distance, we have created an application to measure the node distance as many as 100 times under the same condition. The application

has performed some memory allocation operations and *memcpy()* on all processors in order, and computed the time takes for these operations to complete. At start-up, the program sets memory affinity to allow a certain range of virtual memory addresses to be mapped to the physical memory in a particular ccNUMA node, and then it runs on the processor 0 to the last processor in order. After all operations have finished, the program sets its memory affinity from the current node to the next node, and performs the same operations repeatedly until it traverses all nodes. In the experiment, the program measures and records the time to complete all operations.

For instance, as seen in Appendix D.1.1, the application sets allowing the memory allocation from the physical memory belonging to *node 0*, and requests allocating the given size of 256M bytes memory and memory copy operations, which writes some meaningless data into the memory served to ensure mapping the virtual memory onto the physical memory, on *processor 0* until finishing the operations. After this, it will change processor affinity from *processor 0* to *processor 1* to run on another processor in order. After traversal of all of the processors with *node 0* memory affinity, it will change memory affinity from *node 0* to *node 1*, and run itself on from *processor 0* to *processor 15* repeatedly.

The average value for both the local and remote accesses are shown in table B.5. As expected, the average execution time in the remote accesses is longer than the values of local accesses. For instance, at the first line, the average execution time in the local access, the application running on node 0 with memory affinity on node 0, is 2.321s compared to 3.33s for the remote accesses at line 4. For all cases of minimum and maximum execution times, the values for the remote accesses are even longer than the local accesses.

The experiment results enable us to establish a new node distance table shown in B.1. In this table the measured value for the access of the local node is normalized to 10, and the remote values can be defined as the measured value of the remote node multiplied by 10 over that for the measured value of the local node. As can be seen from the results, the measured distances are different from those given by the SLIT in the BIOS (and shown in Table 3.1). Note that the numbers in brackets represent the original values of the node distance table.

	node 0	node 1	node 2	node 3
node 0	10 (10)	13 (20)	12 (20)	14 (20)
node 1	13 (20)	10 (10)	14 (20)	12 (20)
node 2	12 (20)	14 (20)	10 (10)	13 (20)
node 3	14 (20)	12 (20)	13 (20)	10 (10)

Table B.1: Measured Node Distances

In the results, the measured consumption time of 2-hop remote access is longer than 1-hop remote access or 0-hop local access, which means that the 2-hop remote access should be avoided as much as possible to improve the performance of memory operations. For this reason, we have changed the values of the node distance table in the OS to use the new measured values.

An eight nodes-based ccNUMA architecture system

Table B.2 shows the original SLIT of the system. As seen in the table, it contains slightly different values compared to table 3.1. This is because the processor consists of sibling nodes, so it needs to represent them in terms of different values. The value of 16 means 1-hop remote access including sibling nodes, and the value of 22 means that it needs 2-hop access to remote memory.

	node 0	node 1	node 2	node 3	node 4	node 5	node 6	node 7
node 0	10	16	16	22	16	22	16	22
node 1	16	10	22	16	22	16	22	16
node 2	16	22	10	16	16	22	16	22
node 3	22	16	16	10	22	16	22	16
node 4	16	22	16	22	10	16	16	22
node 5	22	16	22	16	16	10	22	16
node 6	16	22	16	22	16	22	10	16
node 7	22	16	22	16	22	16	16	10

Table B.2: The SLIT of more complex architecture system

As we have described, the system comprises four processors physically as well as eight nodes logically, as seen in figure 4.2. In particular, nodes 0-1, 2-3, 4-5 and 6-7

belong to each processor, respectively; they are in a sibling relationship with each other, so their relationship should be closer than other nodes. For instance, node 0 and node 1 (cores 0 to 7) belong to the same processor physically, so the value of 0-hop is 10, and the value of 1-hop between node 0 and node 1 is 16 in the table.

However, the table does not represent the systems' architecture very accurately. In the table, the values of node distances from node 0 to nodes 1, 2, 4 or 6 are even just 16, although their relationships are not entirely sibling nodes. According to the architecture design, the value of node distance from node 0 to node 1 should be less than others. Other similar cases can be found in the table.

In order to describe the systems' topology accurately, the experiment to evaluate the node distance is executed using the same application, which requests allocating the given size of memory and memory copy operations, which writes some meaningless data. Table B.3 shows the different effects of local and remote accesses on performance.

Line	Processor Affinity		Memory Affinity	Average	STD
	Node ID	Core ID			
1	0	00-03	node 0	0.185	0.007
2	1	04-07	node 0	0.279	0.008
3	2	08-11	node 0	0.468	0.013
4	3	12-15	node 0	0.506	0.016
5	4	16-19	node 0	0.395	0.023
6	5	20-23	node 0	0.44	0.022
7	6	24-27	node 0	0.483	0.013
8	7	28-31	node 0	0.5	0.013
9	0	00-03	node 1	0.268	0.007
10	1	04-07	node 1	0.18	0.007
11	2	08-11	node 1	0.483	0.013
12	3	12-15	node 1	0.443	0.012
13	4	16-19	node 1	0.362	0.01
14	5	20-23	node 1	0.299	0.009
15	6	24-27	node 1	0.494	0.013
16	7	28-31	node 1	0.464	0.013
17	0	00-03	node 2	0.381	0.014

18	1	04-07	node 2	0.467	0.013
19	2	08-11	node 2	0.174	0.008
20	3	12-15	node 2	0.26	0.007
21	4	16-19	node 2	0.431	0.01
22	5	20-23	node 2	0.457	0.01
23	6	24-27	node 2	0.433	0.01
24	7	28-31	node 2	0.407	0.012
25	0	00-03	node 3	0.426	0.017
26	1	04-07	node 3	0.437	0.012
27	2	08-11	node 3	0.264	0.007
28	3	12-15	node 3	0.179	0.008
29	4	16-19	node 3	0.469	0.012
30	5	20-23	node 3	0.434	0.011
31	6	24-27	node 3	0.471	0.012
32	7	28-31	node 3	0.384	0.014
33	0	00-03	node 4	0.43	0.011
34	1	04-07	node 4	0.356	0.008
35	2	08-11	node 4	0.362	0.009
36	3	12-15	node 4	0.462	0.01
37	4	16-19	node 4	0.176	0.004
38	5	20-23	node 4	0.261	0.005
39	6	24-27	node 4	0.429	0.01
40	7	28-31	node 4	0.462	0.011
41	0	00-03	node 5	0.473	0.012
42	1	04-07	node 5	0.295	0.007
43	2	08-11	node 5	0.407	0.012
44	3	12-15	node 5	0.428	0.01
45	4	16-19	node 5	0.264	0.005
46	5	20-23	node 5	0.178	0.005
47	6	24-27	node 5	0.467	0.011
48	7	28-31	node 5	0.431	0.009
49	0	00-03	node 6	0.391	0.016
50	1	04-07	node 6	0.469	0.015
51	2	08-11	node 6	0.425	0.013
52	3	12-15	node 6	0.464	0.016

53	4	16-19	node 6	0.431	0.014
54	5	20-23	node 6	0.406	0.016
55	6	24-27	node 6	0.169	0.009
56	7	28-31	node 6	0.255	0.009
57	0	00-03	node 7	0.416	0.013
58	1	04-07	node 7	0.452	0.012
59	2	08-11	node 7	0.461	0.011
60	3	12-15	node 7	0.443	0.011
61	4	16-19	node 7	0.471	0.013
62	5	20-23	node 7	0.368	0.01
63	6	24-27	node 7	0.264	0.005
64	7	28-31	node 7	0.178	0.005

Table B.3: The measured results for more complex system

The average value for both the local and remote accesses is shown in table B.3. The average execution time in the remote accesses is longer than the values of local accesses. Furthermore, the values of sibling node access at line 2 are less than the other explicit remote accesses. At the first line, the average execution time in the local access, the application running on *node 0* with memory affinity on *node 0*, is 0.185s compared to 0.506s for the remote accesses at line 4. At line 2, since the sibling relationship with cores 0-3, the value of average execution time is significantly shorter, by almost half, than other remote accesses. For all cases of minimum and maximum execution times, the values for the remote accesses are even longer than the local accesses.

After evaluating the real node distance with the application, we are able to describe the architecture design, especially the sibling relationship, and establish the new node distance in table B.4 based on table B.3. The values in the table are significantly different compared to the original node distance in table B.2. The main reason why they are different is that the original model depends on the simple metric to obtain information to describe the system architecture design.

The new table is able to describe the system architecture more accurately compared to the original table. Only the sibling nodes are able to be represented by the value of 15, and other values of remote accesses are longer than them.

This experiment to measure the real node distances highlights the importance of describing the ccNUMA architecture design, showing that the new node distance table is better than the original table in representing the underlying architecture. The result will be used to sort the node order, which indicates how the physical memory will be placed as closely as possible to a specified node.

	node 0	node 1	node 2	node 3	node 4	node 5	node 6	node 7
node 0	10 (10)	15 (16)	25 (16)	27 (22)	21 (16)	24 (22)	26 (16)	27 (22)
node 1	15 (16)	10 (10)	27 (22)	25 (16)	20 (22)	17 (16)	27 (22)	26 (16)
node 2	22 (16)	27 (22)	10 (10)	15 (16)	25 (16)	26 (22)	25 (16)	23 (22)
node 3	24 (22)	24 (16)	15 (16)	10 (10)	26 (22)	24 (16)	26 (22)	21 (16)
node 4	24 (16)	20 (22)	21 (16)	26 (22)	10 (10)	15 (16)	24 (16)	26 (22)
node 5	27 (22)	17 (16)	23 (22)	24 (16)	15 (16)	10 (10)	26 (22)	24 (16)
node 6	23 (16)	28 (22)	25 (16)	27 (22)	26 (16)	24 (22)	10 (10)	15 (16)
node 7	23 (22)	25 (16)	26 (22)	25 (16)	27 (22)	21 (16)	15 (16)	10 (10)

Table B.4: The new node distance of more complex architecture system

B.3 Evaluation On The Real Node Distance

This section describes the benefits from discovering real node distance on the underlying system. The evaluation measures the allocation time for some given sizes of memory blocks in the remote memory area on the experimental machine described in figure 4.1. For the allocation of local memory, it is not appropriate to compare the average execution time between the original approach and our approach. This is because our model has attempted to reduce the average execution time of the remote memory accesses, and the average time of local accesses on both models should be approximately equal. The results are presented and analysed below. In order to represent memory affinity and processor affinity simply, we have used the following representation method.

We have used bits to represent four nodes in the system. Each individual bit indicates a specific node according to its position. A bit set by 1 means that the application runs on the specific node where the bit indicates. Assuming that an application runs on node 0 only, it represents 0001; otherwise, it represents 1000 if an application runs on node 3. Similarly, it can be used to represent the memory affinity. An application requests allocating memory to node 0 only is represented by 0001, and by 1000 if only node 3 is used. Therefore, the pair 0001 1110 (processor affinity, memory affinity) indicates that a task runs on node 0 in which it allocates page frames from nodes 1 to 3.

The application iterates requesting memory allocations as many as 100 times per core so, in its entirety, it executes 400 times per node. It allocates sizes of blocks that are a power of two apart from 32M bytes to 512M bytes. After that, it writes some meaningless data into the memory served to ensure mapping the virtual memory into the physical memory. In the process, in order to exclude local memory for the allocation, we have exploited a collection of application-level functions supported by the NUMA library called *libnuma*, with the application measing the time to request memory and to complete write operations.

Allocation time test based on node 0

The execution time is measured for the allocation of memory blocks which are of the sizes 32M, 64M, 128M, 256M and 512M bytes with the 0001 1110 processor and memory affinity. The measured execution time is noted for both the original model and the new model. Table B.6 shows the statistics analysis of the timing.

The following Table B.5 shows average values for both the local and remote accesses on our experimental machine, an four nodes-based ccNUMA architecture system.

Line	Processor Affinity		Memory Affinity	Average	STD
	Node ID	Core ID			
1	0	00-03	node 0	2.321	0.052
2	1	04-07	node 0	3.065	0.016
3	2	08-11	node 0	2.915	0.031
4	3	12-15	node 0	3.33	0.026
1	0	00-03	node 1	3.044	0.011
2	1	04-07	node 1	2.268	0.032
3	2	08-11	node 1	3.28	0.017
4	3	12-15	node 1	2.847	0.02
1	0	00-03	node 2	2.877	0.022
2	1	04-07	node 2	3.275	0.019
3	2	08-11	node 2	2.266	0.036
4	3	12-15	node 2	3.07	0.011
1	0	00-03	node 3	3.287	0.027
2	1	04-07	node 3	2.845	0.033
3	2	08-11	node 3	3.083	0.015
4	3	12-15	node 3	2.271	0.053

Table B.5: The measured actual time taken on a four nodes-based ccNUMA system

The values of average execution time prove that the allocation time with our new approach is less than the original approach. For the allocation memory on node 0, the difference is little but the new approach is quicker to allocate memory. As seen in figure B.1, the performance improvement varies from 5.1% to 7.1% approximately.

Allocation time test based on node 1

Model	Size	Affinity	Average	STD
Original	32MB	0001 1110	0.377	0.002
Original	64MB	0001 1110	0.752	0.004
Original	128MB	0001 1110	1.513	0.011
Original	256MB	0001 1110	3.043	0.036
Original	512MB	0001 1110	6.253	0.071
New	32MB	0001 1110	0.352	0.002
New	64MB	0001 1110	0.715	0.006
New	128MB	0001 1110	1.426	0.01
New	256MB	0001 1110	2.894	0.057
New	512MB	0001 1110	5.881	0.134

Table B.6: Allocation timing statistics based on node 0

The execution time is measured for the allocation of memory blocks that are of the sizes 32M, 64M, 128M, 256M and 512M bytes with the 0010 1101 processor and memory affinity. The measured execution time is noted for both the original model and the new model. Table B.7 shows the statistics analysis of the timing.

The values of average execution time prove that the allocation time with our new approach is less time the original approach. For the allocation memory on node 1, the difference is relatively big compared to the results of node 0. As seen in figure B.1, the performance improvement varies from 13.8% to 16% approximately.

Model	Size	Affinity	Average	STD
Original	32MB	0010 1101	0.404	0.004
Original	64MB	0010 1101	0.809	0.004
Original	128MB	0010 1101	1.627	0.008
Original	256MB	0010 1101	3.251	0.007
Original	512MB	0010 1101	6.615	0.104
New	32MB	0010 1101	0.348	0.001
New	64MB	0010 1101	0.703	0.005
New	128MB	0010 1101	1.413	0.009
New	256MB	0010 1101	2.853	0.054
New	512MB	0010 1101	5.811	0.149

Table B.7: Allocation timing statistics based on node 1

Allocation time test based on node 2

The execution time is measured for the allocation of memory blocks that are of the sizes 32M, 64M, 128M, 256M and 512M bytes with the 0100 1011 processor and memory affinity. The measured execution time has been noted for both the original model and the new model. Table B.7 shows the statistics analysis of the timing.

The values of average execution time prove that the allocation time with our new approach is less than the original approach. For the allocation memory on node 2, the difference is relatively little compared to the results of node 1. As seen in figure B.1, the performance improvement varies from 5.7% to 7.54% approximately.

Model	Size	Affinity	Average	STD
Original	32MB	0100 1011	0.379	0.001
Original	64MB	0100 1011	0.765	0.002
Original	128MB	0100 1011	1.531	0.003
Original	256MB	0100 1011	3.097	0.035
Original	512MB	0100 1011	6.211	0.075
New	32MB	0100 1011	0.355	0.002
New	64MB	0100 1011	0.716	0.002
New	128MB	0100 1011	1.448	0.018
New	256MB	0100 1011	2.92	0.045
New	512MB	0100 1011	5.775	0.055

Table B.8: Allocation timing statistics based on node 2

Allocation time test based on node 3

The execution time is measured for the allocation of memory blocks that are of the sizes 32M, 64M, 128M, 256M and 512M bytes with the 1000 0111 processor and memory affinity. The measured execution time has been noted for both the original model and the new model. Table B.7 shows the statistics analysis of the timing.

The values of average execution time prove that the allocation time with our new approach is less than the original approach. For the allocation memory on node 3, the difference is relatively big compared to the results of node 0. As seen in figure B.1, the performance improvement varies from 15.7% to 17.4% approximately.

Overall, as we can see in figure B.1, our model shows significantly improved performance compared to the original model. For instance, when allocating memory of size 512M bytes on 1000 0111 affinity, the average execution time is 5.8 seconds with our model; withe the original model it

Model	Size	Affinity	Average	STD
Original	32MB	1000 0111	0.41	0.004
Original	64MB	1000 0111	0.823	0.005
Original	128MB	1000 0111	1.651	0.009
Original	256MB	1000 0111	3.341	0.044
Original	512MB	1000 0111	6.755	0.126
New	32MB	1000 0111	0.349	0.002
New	64MB	1000 0111	0.711	0.008
New	128MB	1000 0111	1.425	0.023
New	256MB	1000 0111	2.88	0.066
New	512MB	1000 0111	5.799	0.146

Table B.9: Allocation timing statistics based on node 3

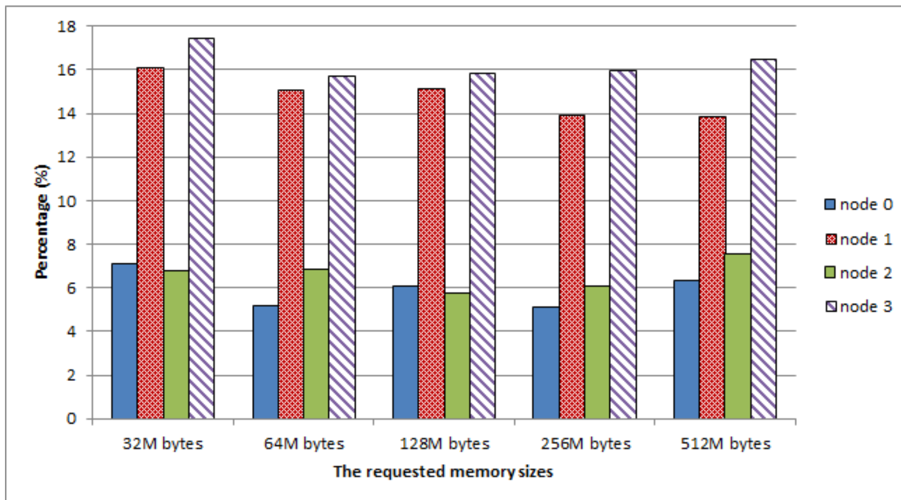

Figure B.1: The ratio of performance improvements

is 6.7 seconds. In this case, the performance of average execution time improves by approximately 16.4%.

Appendix C

Further Evaluation of the Synthetic Models

In this appendix we present the statistics that that were collected during the evaluation using synthetic models reported in Chapter 4.

The following tables represent statistics that were collected during each application execution.

cfrac						
	B_S	Std.B_S	B_{IT}	Std.B_{IT}	B_{HT}	Std.B_{HT}
Test Set 1	19.37	22.28	90492	287654	33277	19264.10
Test Set 2	20.64	23.54	257119	1596396	345504	189851.00
Test Set 3	21.64	26.27	542505	3155992	1196417	680984.17

espresso						
	B_S	Std.B_S	B_{IT}	Std.B_{IT}	B_{HT}	Std.B_{HT}
Test Set 1	71.21	159.06	2882	24836	141	1759.00
Test Set 2	89.51	230.65	2829	19962	142	2976.15
Test Set 3	111.71	754.18	5443	246331	1760	34169.08

gawk						
	B_S	Std.B_S	B_{IT}	Std.B_{IT}	B_{HT}	Std.B_{HT}
Test Set 1	83.82	649.21	6843	37772	2287	3961.06

Test Set 2	49.28	579.32	2666	37817	17782	40998.00
Test Set 3	49.29	647.36	2631	48901	48037	112528.41

p2c						
	B_S	Std.B_S	B_{IT}	Std.B_{IT}	B_{HT}	Std.B_{HT}
Test Set 1	62.09	80.10	10247	132056	2504	1902.56
Test Set 2	49.03	57.62	9429	175924	8618	7819.48
Test Set 3	34.07	28.70	30969	1070255	105422	218719.33

Table C.1: The mean of all test sets of applications

Test Set 1						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	319	20.88%	0	0.00%	2	0.13%
16	524	55.17%	0	0.00%	0	0.13%
32	354	78.34%	0	0.00%	0	0.13%
64	322	99.41%	0	0.00%	1	0.20%
128	0	99.41%	0	0.00%	0	0.20%
256	1	99.48%	0	0.00%	0	0.20%
512	8	100.00%	0	0.00%	7	0.65%
1k	0	100.00%	124	8.12%	7	1.11%
2k	0	100.00%	282	26.59%	12	1.90%
4k	0	100.00%	61	30.58%	12	2.68%
8k	0	100.00%	192	43.16%	102	9.36%
16k	0	100.00%	124	51.28%	217	23.56%
32k	0	100.00%	291	70.33%	451	53.08%
64k	0	100.00%	87	76.03%	632	94.44%
128k	0	100.00%	113	83.43%	85	100.00%
256k	0	100.00%	99	89.91%	0	100.00%
512k	0	100.00%	89	95.74%	0	100.00%
1m	0	100.00%	57	99.48%	0	100.00%
2m	0	100.00%	6	99.87%	0	100.00%
4m	0	100.00%	1	99.93%	0	100.00%
8m	0	100.00%	1	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.2: The cumulative percentage and frequency of *cfrac* test set 1

Test Set 2						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	0	0.00%	0	0.00%	2	0.04%
16	3013	57.10%	0	0.00%	0	0.04%
32	1160	79.08%	0	0.00%	0	0.04%
64	1094	99.81%	0	0.00%	0	0.04%
128	1	99.83%	0	0.00%	1	0.06%
256	0	99.83%	0	0.00%	0	0.06%
512	4	99.91%	0	0.00%	0	0.06%
1k	5	100.00%	763	14.46%	2	0.09%
2k	0	100.00%	1034	34.06%	4	0.17%
4k	0	100.00%	437	42.34%	14	0.44%
8k	0	100.00%	390	49.73%	8	0.59%
16k	0	100.00%	410	57.51%	33	1.21%
32k	0	100.00%	881	74.20%	63	2.41%
64k	0	100.00%	126	76.59%	208	6.35%
128k	0	100.00%	108	78.64%	567	17.09%
256k	0	100.00%	173	81.92%	1083	37.62%
512k	0	100.00%	245	86.56%	2046	76.39%
1m	0	100.00%	300	92.25%	1246	100.00%
2m	0	100.00%	266	97.29%	0	100.00%
4m	0	100.00%	118	99.53%	0	100.00%
8m	0	100.00%	24	99.98%	0	100.00%
16m	0	100.00%	0	99.98%	0	100.00%
32m	0	100.00%	0	99.98%	0	100.00%
64m	0	100.00%	0	99.98%	0	100.00%
128m	0	100.00%	1	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.3: The cumulative percentage and frequency of of *cfrac* test set 2

Test Set 3						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	0	0.00%	0	0.00%	2	0.02%
16	5127	57.88%	0	0.00%	0	0.02%
32	1905	79.39%	0	0.00%	0	0.02%
64	1816	99.89%	0	0.00%	0	0.02%
128	1	99.90%	0	0.00%	1	0.03%
256	0	99.90%	0	0.00%	0	0.03%
512	4	99.94%	0	0.00%	0	0.03%
1k	4	99.99%	1433	16.18%	0	0.03%
2k	1	100.00%	1210	29.84%	4	0.08%
4k	0	100.00%	1379	45.41%	2	0.10%
8k	0	100.00%	140	46.99%	7	0.18%
16k	0	100.00%	450	52.07%	20	0.41%
32k	0	100.00%	1169	65.27%	37	0.82%
64k	0	100.00%	154	67.01%	84	1.77%
128k	0	100.00%	184	69.09%	160	3.58%
256k	0	100.00%	271	72.15%	570	10.01%
512k	0	100.00%	460	77.34%	1010	21.42%
1m	0	100.00%	662	84.81%	1940	43.32%
2m	0	100.00%	674	92.42%	3907	87.42%
4m	0	100.00%	456	97.57%	1114	100.00%
8m	0	100.00%	182	99.63%	0	100.00%
16m	0	100.00%	30	99.97%	0	100.00%
32m	0	100.00%	2	99.99%	0	100.00%
64m	0	100.00%	0	99.99%	0	100.00%
128m	0	100.00%	0	99.99%	0	100.00%
256m	0	100.00%	0	99.99%	0	100.00%
512m	0	100.00%	1	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.4: The cumulative percentage and frequency of *cfrac* test set 3

Test Set 1						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	9435	38.26%	0	0.00%	13898	56.36%
16	835	41.65%	0	0.00%	2888	68.08%
32	648	44.28%	0	0.00%	2014	76.24%
64	9460	82.64%	0	0.00%	1823	83.64%
128	2062	91.00%	0	0.00%	1283	88.84%
256	961	94.90%	0	0.00%	1447	94.71%
512	179	95.63%	2776	11.26%	418	96.40%
1k	933	99.41%	13765	67.08%	778	99.56%
2k	137	99.97%	3808	82.53%	31	99.68%
4k	0	99.97%	2290	91.82%	13	99.74%
8k	8	100.00%	1197	96.67%	7	99.76%
16k	0	100.00%	297	97.87%	3	99.78%
32k	0	100.00%	337	99.24%	4	99.79%
64k	0	100.00%	72	99.53%	51	100.00%
128k	0	100.00%	53	99.75%	0	100.00%
256k	0	100.00%	35	99.89%	0	100.00%
512k	0	100.00%	16	99.96%	0	100.00%
1m	0	100.00%	7	99.98%	0	100.00%
2m	0	100.00%	4	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.5: The cumulative percentage and frequency of *espresso* test set 1

Test Set 2						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	17505	28.54%	0	0.00%	28813	46.97%
16	1635	31.20%	0	0.00%	6981	58.36%
32	908	32.69%	0	0.00%	6213	68.49%
64	31987	84.83%	0	0.00%	8934	83.05%
128	4576	92.30%	0	0.00%	4001	89.57%
256	1588	94.88%	0	0.00%	4185	96.40%
512	916	96.38%	7262	11.84%	1375	98.64%
1k	129	96.59%	37688	73.28%	691	99.77%
2k	1976	99.81%	7097	84.86%	34	99.82%
4k	109	99.99%	4212	91.72%	13	99.84%
8k	8	100.00%	3058	96.71%	29	99.89%
16k	0	100.00%	536	97.58%	3	99.89%
32k	0	100.00%	762	98.82%	6	99.90%
64k	0	100.00%	449	99.56%	2	99.91%
128k	0	100.00%	111	99.74%	57	100.00%
256k	0	100.00%	88	99.88%	0	100.00%
512k	0	100.00%	51	99.96%	0	100.00%
1m	0	100.00%	19	100.00%	0	100.00%
2m	0	100.00%	3	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.6: The cumulative percentage and frequency of *espresso* test set 2

Test Set 3						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	16336	0.98%	0	0.00%	524782	31.45%
16	533275	32.94%	0	0.00%	138481	39.75%
32	10730	33.59%	0	0.00%	138746	48.07%
64	875171	86.04%	0	0.00%	129981	55.86%
128	116984	93.05%	0	0.00%	119147	63.00%
256	51834	96.16%	0	0.00%	108343	69.50%
512	35544	98.29%	97750	5.86%	103608	75.71%
1k	10500	98.92%	1171033	76.05%	91362	81.18%
2k	4688	99.20%	119037	83.18%	64993	85.08%
4k	4918	99.50%	142397	91.72%	78281	89.77%
8k	6160	99.87%	63465	95.52%	75479	94.30%
16k	198	99.88%	35123	97.63%	51100	97.36%
32k	2041	100.00%	26194	99.20%	39655	99.73%
64k	5	100.00%	5870	99.55%	4288	99.99%
128k	0	100.00%	4124	99.80%	16	99.99%
256k	0	100.00%	1568	99.89%	16	99.99%
512k	0	100.00%	840	99.94%	20	99.99%
1m	0	100.00%	403	99.97%	6	100.00%
2m	0	100.00%	173	99.98%	7	100.00%
4m	0	100.00%	98	99.98%	3	100.00%
8m	0	100.00%	142	99.99%	70	100.00%
16m	0	100.00%	94	100.00%	0	100.00%
32m	0	100.00%	45	100.00%	0	100.00%
64m	0	100.00%	26	100.00%	0	100.00%
128m	0	100.00%	1	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.7: The cumulative percentage and frequency of *espresso* test set 3

Test Set 1						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	2050	44.59%	0	0.00%	2457	53.45%
16	198	48.90%	0	0.00%	276	59.45%
32	987	70.37%	0	0.00%	271	65.35%
64	1064	93.52%	0	0.00%	44	66.30%
128	193	97.72%	0	0.00%	17	66.67%
256	56	98.93%	0	0.00%	37	67.48%
512	7	99.09%	8	0.17%	26	68.04%
1k	4	99.17%	880	19.32%	81	69.81%
2k	2	99.22%	786	36.42%	99	71.96%
4k	0	99.22%	1259	63.82%	203	76.38%
8k	32	99.91%	753	80.20%	466	86.51%
16k	4	100.00%	599	93.23%	620	100.00%
32k	0	100.00%	266	99.02%	0	100.00%
64k	0	100.00%	25	99.56%	0	100.00%
128k	0	100.00%	7	99.72%	0	100.00%
256k	0	100.00%	4	99.80%	0	100.00%
512k	0	100.00%	4	99.89%	0	100.00%
1m	0	100.00%	4	99.98%	0	100.00%
2m	0	100.00%	1	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.8: The cumulative percentage and frequency of *gawk* test set 1

Test Set 2						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	35709	28.14%	0	0.00%	99306	78.24%
16	36871	57.19%	0	0.00%	1005	79.04%
32	28675	79.78%	0	0.00%	713	79.60%
64	25107	99.56%	0	0.00%	82	79.66%
128	18	99.58%	0	0.00%	53	79.70%
256	15	99.59%	0	0.00%	86	79.77%
512	7	99.59%	8	0.01%	62	79.82%
1k	4	99.60%	32363	25.51%	91	79.89%
2k	2	99.60%	34683	52.83%	213	80.06%
4k	0	99.60%	52327	94.06%	135	80.17%
8k	505	100.00%	5721	98.57%	1	80.17%
16k	4	100.00%	677	99.10%	1046	80.99%
32k	0	100.00%	924	99.83%	2449	82.92%
64k	1	100.00%	106	99.91%	5117	86.95%
128k	1	100.00%	42	99.95%	10644	95.34%
256k	0	100.00%	22	99.96%	5916	100.00%
512k	0	100.00%	33	99.99%	0	100.00%
1m	0	100.00%	9	100.00%	0	100.00%
2m	0	100.00%	1	100.00%	0	100.00%
4m	0	100.00%	1	100.00%	0	100.00%
8m	0	100.00%	0	100.00%	0	100.00%
16m	0	100.00%	1	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.9: The cumulative percentage and frequency of *gawk* test set 2

Test Set 3						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	88342	25.13%	0	0.00%	274761	78.16%
16	101810	54.09%	0	0.00%	3464	79.14%
32	90035	79.70%	0	0.00%	1787	79.65%
64	69922	99.59%	0	0.00%	174	79.70%
128	18	99.59%	0	0.00%	104	79.73%
256	15	99.60%	0	0.00%	191	79.78%
512	7	99.60%	11	0.00%	99	79.81%
1k	4	99.60%	102285	29.10%	86	79.84%
2k	2	99.60%	66203	47.93%	174	79.88%
4k	0	99.60%	159412	93.27%	281	79.96%
8k	1395	100.00%	18692	98.59%	618	80.14%
16k	4	100.00%	1882	99.13%	1238	80.49%
32k	0	100.00%	2618	99.87%	2182	81.11%
64k	1	100.00%	177	99.92%	4671	82.44%
128k	1	100.00%	113	99.95%	10305	85.37%
256k	1	100.00%	43	99.97%	19100	90.81%
512k	0	100.00%	87	99.99%	32322	100.00%
1m	0	100.00%	29	100.00%	0	100.00%
2m	0	100.00%	1	100.00%	0	100.00%
4m	0	100.00%	0	100.00%	0	100.00%
8m	0	100.00%	1	100.00%	0	100.00%
16m	0	100.00%	1	100.00%	0	100.00%
32m	0	100.00%	1	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.10: The cumulative percentage and frequency of *gawk* test set 3

Test Set 1						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	620	23.14%	0	0.00%	285	10.64%
16	136	28.22%	0	0.00%	72	13.33%
32	350	41.28%	0	0.00%	55	15.38%
64	813	71.63%	0	0.00%	37	16.76%
128	286	82.31%	0	0.00%	28	17.81%
256	457	99.37%	0	0.00%	48	19.60%
512	0	99.37%	7	0.26%	4	19.75%
1k	17	100.00%	631	23.82%	70	22.36%
2k	0	100.00%	1088	64.45%	599	44.72%
4k	0	100.00%	578	86.03%	768	73.39%
8k	0	100.00%	159	91.97%	675	98.58%
16k	0	100.00%	93	95.44%	38	100.00%
32k	0	100.00%	85	98.62%	0	100.00%
64k	0	100.00%	14	99.14%	0	100.00%
128k	0	100.00%	5	99.33%	0	100.00%
256k	0	100.00%	3	99.44%	0	100.00%
512k	0	100.00%	7	99.70%	0	100.00%
1m	0	100.00%	4	99.85%	0	100.00%
2m	0	100.00%	1	99.89%	0	100.00%
4m	0	100.00%	2	99.96%	0	100.00%
8m	0	100.00%	1	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.11: The cumulative percentage and frequency of *p2c* test set 1

Test Set 2						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	1007	16.49%	0	0.00%	1243	20.35%
16	167	19.22%	0	0.00%	272	24.81%
32	1994	51.87%	0	0.00%	218	28.38%
64	1828	81.81%	0	0.00%	89	29.83%
128	574	91.21%	0	0.00%	26	30.26%
256	519	99.71%	0	0.00%	33	30.80%
512	1	99.72%	75	1.23%	116	32.70%
1k	17	100.00%	1496	25.73%	201	35.99%
2k	0	100.00%	2331	63.90%	128	38.09%
4k	0	100.00%	1098	81.89%	42	38.78%
8k	0	100.00%	571	91.24%	779	51.53%
16k	0	100.00%	258	95.46%	1290	72.65%
32k	0	100.00%	183	98.46%	1670	100.00%
64k	0	100.00%	42	99.15%	0	100.00%
128k	0	100.00%	17	99.43%	0	100.00%
256k	0	100.00%	12	99.62%	0	100.00%
512k	0	100.00%	11	99.80%	0	100.00%
1m	0	100.00%	5	99.89%	0	100.00%
2m	0	100.00%	3	99.93%	0	100.00%
4m	0	100.00%	2	99.97%	0	100.00%
8m	0	100.00%	1	99.98%	0	100.00%
16m	0	100.00%	1	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.12: The cumulative percentage and frequency of *p2c* test set 2

Test Set 3						
	B_S		B_{IT}		B_{HT}	
Bin	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*	*Freq*	*Cumulative %*
8	2382	5.42%	0	0.00%	21895	49.83%
16	306	6.12%	0	0.00%	3138	56.97%
32	30334	75.15%	0	0.00%	1622	60.66%
64	7859	93.03%	0	0.00%	1226	63.45%
128	2107	97.83%	0	0.00%	817	65.31%
256	935	99.96%	0	0.00%	512	66.47%
512	2	99.96%	515	1.17%	505	67.62%
1k	17	100.00%	6662	16.33%	458	68.67%
2k	0	100.00%	9852	38.75%	230	69.19%
4k	0	100.00%	6931	54.53%	397	70.09%
8k	0	100.00%	10188	77.71%	830	71.98%
16k	0	100.00%	7521	94.83%	909	74.05%
32k	0	100.00%	1682	98.66%	1611	77.72%
64k	0	100.00%	258	99.24%	1068	80.15%
128k	0	100.00%	131	99.54%	14	80.18%
256k	0	100.00%	47	99.65%	1044	82.55%
512k	0	100.00%	65	99.80%	2341	87.88%
1m	0	100.00%	31	99.87%	5325	100.00%
2m	0	100.00%	11	99.89%	0	100.00%
4m	0	100.00%	10	99.92%	0	100.00%
8m	0	100.00%	8	99.93%	0	100.00%
16m	0	100.00%	6	99.95%	0	100.00%
32m	0	100.00%	13	99.98%	0	100.00%
64m	0	100.00%	8	100.00%	0	100.00%
128m	0	100.00%	2	100.00%	0	100.00%
More	0	100.00%	0	100.00%	0	100.00%

Table C.13: The cumulative percentage and frequency of *p2c* test set 3

Appendix D

Supporting Implementations

In this appendix we provide details of the various algorithms that have been written in support of the research.

D.1 Supplemental Applications

D.1.1 Memory Access Latencies on ccNUMA architecture systems

This application is derived from the TAU benchmark[1], which measures the time elapsed by *memcpy()* on the underlying ccNUMA architecture system. The original code has been changed to use APIs provided by the Linux instead of the TAU library.

Source 29

```
1  #define _GNU_SOURCE
2  #include <sched.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/time.h>
6  #include <string.h>
7  #include <numaif.h>
8  #include <numa.h>
9
10 #define MEM_MB 5
11 #define MEM_SIZE MEM_MB*1024L*1024L
12 #define ITER 40
13
14 double getTime() {
15   struct timeval tp;
16   static double last_timestamp = 0.0;
```

[1] http://www.nic.uoregon.edu/tau-wiki/Guide:Opteron_NUMA_Analysis

```
17    double timestamp;
18    gettimeofday (&tp, 0);
19    timestamp = (double) tp.tv_sec * 1e6 + tp.tv_usec;
20    return timestamp;
21  }
22
23  int getNumCPU() {
24    cpu_set_t mask;
25    if (sched_getaffinity(0,sizeof(cpu_set_t),&mask)) {
26      fprintf (stderr, "Unable to retrieve affinity\n");
27      exit(1);
28    }
29    int nproc = 0;
30    for(int i=0; i<CPU_SETSIZE; i++) {
31      if( CPU_ISSET(i,&mask) ) {
32        nproc++;
33      }
34    }
35    return nproc;
36  }
37
38  void memtest(char *ptr) {
39    for (int i=0; i<ITER; i++) {
40      memcpy(ptr, ptr+(MEM_SIZE/2),MEM_SIZE/2);
41    }
42  }
43
44  void setCPU(int cpu) {
45    cpu_set_t mask;
46    CPU_ZERO(&mask);
47    CPU_SET(cpu, &mask);
48    sched_setaffinity(0, sizeof(cpu_set_t), &mask);
49  }
50
51  void test(int cpu, int nproc) {
52    setCPU(cpu);
53    char *ptr = (char*) malloc (MEM_SIZE);
54    if (!ptr) {
55      fprintf (stderr, "failed to malloc\n");
56      exit(1);
57    }
58    // make sure it all gets paged in
59    for (long j=0; j<MEM_SIZE; j++) {
60      ptr[j] = j;
61    }
62    for (int i = 0; i < nproc; i++) {
63      setCPU(i);
64      double start = getTime();
65      memtest(ptr);
66      double end = getTime();
67      printf ("%d: time = %G seconds\n", i, (end - start) / (1000*1000));
68    }
69    free (ptr);
70  }
71
72  int main (int argc, char **argv) {
```

```
73      int nproc = getNumCPU();
74    for (int i = 0; i < nproc; i++) {
75      test(i,nproc);
76    }
77    return 0;
78  }
```

D.1.2 Header Generator

The application reads a number of raw data files generated by the application D.1.1, and generates a header file, which is used at compile-time of the Linux kernel. The header file contains the information of real node distances.

Source 30

```
1   #include <stdio.h>
2   #include <time.h>
3   #include <stdlib.h>
4   #include <sys/types.h>
5   #include <dirent.h>
6   #include <unistd.h>
7   #include <numa.h>
8
9   #define MAX_DATAFILE 4
10  #define BIT_IN_BYTE 8
11  #define DEFAULT_FILE "numa_distance_table.h"
12  int CntCPU=0;
13  int CntNode=0;
14
15  double *TotalTimePerCPU;
16  double *TotalTimePerNode;
17  int *TotalDataCntPerCPU;
18  int *TotalCPUPerNode;
19  int *distanceTable;
20
21  void getTotalCPUPerNode()
22  {
23    struct bitmask *cpuMask;
24    cpuMask = (struct bitmask*)numa_allocate_cpumask();
25    if ( !cpuMask )
26    {
27      fprintf(stderr, "fail to numa_bitmask_alloc() for cpu\n");
28      exit(1);
29    }
30
31    int cntNode = numa_max_node()+1;
32    int i=0;
33
34    //scan all node
35    for(i=0; i<cntNode; i++)
36    {
37      numa_bitmask_clearall(cpuMask);
38
39      if ( numa_node_to_cpus(i, cpuMask) != 0 )
40      {
```

```
41        fprintf(stderr, "fail to numa_node_to_cpus()\n");
42        exit(1);
43      }
44
45      TotalCPUPerNode[i] = getCntCPUsonNode(cpuMask);
46    }
47    numa_free_cpumask(cpuMask);
48  }
49
50  int getNodeID(int cpuID)
51  {
52    struct bitmask *cpuMask;
53    cpuMask = (struct bitmask*)numa_allocate_cpumask();
54    if ( !cpuMask )
55    {
56      fprintf(stderr, "fail to numa_bitmask_alloc() for cpu\n");
57      exit(1);
58    }
59
60    int cntNode = numa_max_node()+1;
61    int i=0;
62
63    //scan all node
64    for(i=0; i<cntNode; i++)
65    {
66      numa_bitmask_clearall(cpuMask);
67
68      if ( numa_node_to_cpus(i, cpuMask) != 0 )
69      {
70        fprintf(stderr, "fail to numa_node_to_cpus()\n");
71        exit(1);
72      }
73
74      int cntCPUonNode = getCntCPUsonNode(cpuMask);
75      int idx=0,j=0;
76      int resID=0;
77      for ( j=0, idx=0; j<cntCPUonNode; j++)
78      {
79        if ( ( resID = getCPUID(&idx, cpuMask)) != -1 )
80        {
81          if ( resID == cpuID )
82          {
83            numa_free_cpumask(cpuMask);
84            return i;
85          }
86        }
87      }
88
89    }
90    numa_free_cpumask(cpuMask);
91    return -1;
92  }
93
94  int getConfiguredCPUs()
95  {
96    int             filecount=0;
```

```
97    char            *dirnamep = "/sys/devices/system/cpu";
98    struct dirent   *dirent;
99    DIR             *dir;
100   int   max, n;
101   dir = opendir(dirnamep);
102
103   if (dir == NULL)
104   {
105     /* fall back to using the online cpu count */
106     return sysconf(_SC_NPROCESSORS_CONF) - 1;
107   }
108   while ((dirent = readdir(dir)) != 0)
109   {
110     if (sscanf(dirent->d_name, "cpu%d", &n) == 1 && n > max )
111     {
112       max = n;
113     }
114   }
115
116   closedir(dir);
117   return max+1;
118 }
119
120 int getMin(double *val, int total)
121 {
122   int i=0, idx=0;
123   double min=val[0];
124   for(i=0,idx=0; i<total; i++)
125   {
126     if ( min > val[i] )
127     {
128       min = (val[i]);
129       idx = i;
130     }
131   }
132   return idx;
133 }
134 int getMax(double *val, int total)
135 {
136   int i=0, idx=0;
137   double max=val[0];
138   for(i=0; i<total; i++)
139   {
140     if ( max < val[i] )
141     {
142       max = (val[i]);
143       idx=i;
144     }
145   }
146   return idx;
147 }
148
149 int getCPUID(int* idx, struct bitmask* mask)
150 {
151   int len = sizeof(*(mask->maskp)) * BIT_IN_BYTE;
152   int i=0, cnt=0, bit=0;
```

```
153
154   for (i=*idx; i<len; i++)
155   {
156     (*idx)++;
157     bit = ((*(mask->maskp)>>i) & 0x01) * (i+1);
158     if ( bit > 0 )
159       return (bit-1);
160   }
161   return -1;
162 }
163
164 int getCntCPUsonNode(struct bitmask* mask)
165 {
166   int len = sizeof(*(mask->maskp)) * BIT_IN_BYTE;
167   int i=0, cnt=0;
168
169   for (i=0; i<len; i++)
170   {
171     if ( ((*(mask->maskp) >> i) & 0x01 ) == 1 )
172       cnt++;
173   }
174   return cnt;
175 }
176
177 static void saveDistanceTable()
178 {
179   FILE *fp;
180
181   if ( NULL == (fp = fopen(DEFAULT_FILE, "w")) )
182   {
183     fprintf(stderr, "fail to fopen(%s)\n", DEFAULT_FILE);
184     exit(1);
185   }
186
187   fprintf(fp, "#ifndef __X86_MM_NUMA_DISTANCE_TABLE_H\n");
188   fprintf(fp, "#define __X86_MM_NUMA_DISTANCE_TABLE_H\n\n");
189   fprintf(fp, "#define __NUMA_DISTANCE_TABLE_ROWS__\t%d\n", CntNode);
190   fprintf(fp, "#define __NUMA_DISTANCE_TABLE_COLS__\t%d\n\n", CntNode);
191   fprintf(fp, "u16 __numa_distance_table[%d][%d]={\n", CntNode, CntNode);
192   int i=0;
193   for(i=0; i<CntNode; i++)
194   {
195     fprintf(fp, "\t{");
196     int j=0;
197     for(j=0; j<CntNode; j++)
198     {
199       fprintf(fp, "%d", distanceTable[i*CntNode+j]);
200       if ( j != CntNode-1 )
201         fprintf(fp, ",");
202     }
203     if ( i != CntNode-1 )
204       fprintf(fp, "},\n");
205     else
206       fprintf(fp, "}\n");
207
208   }
```

```
209   fprintf(fp, "};\n");
210   fprintf(fp, "#endif //__X86_MM_NUMA_DISTANCE_TABLE_H\n");
211 
212   fclose(fp);
213 }
214 
215 static void generate(char** filename)
216 {
217   int nodeID=0, cnt=0;
218   time_t time;
219   int cpuID;
220   double diff;
221 
222   //loop for all data files
223   for( nodeID=0; nodeID<MAX_DATAFILE; nodeID++)
224   {
225 #ifdef __DEBUG__
226     printf("data filename[%d] = %s\n", nodeID, filename[nodeID]);
227 #endif
228     FILE *fp;
229 
230     if ( NULL == (fp = fopen(filename[nodeID], "r")) )
231     {
232       fprintf(stderr, "fail to fopen(%s)\n", filename[nodeID]);
233       exit(1);
234     }
235 
236     //read file
237     cnt=0;
238     while( fscanf(fp, "%lu\t%d\t%lf\n", &time, &cpuID, &diff) > 0 )
239     {
240       if ( cpuID > CntCPU )
241       {
242         fprintf(stderr, "check datafile & the number of "
243                 "CPU[%d:%d] in the system\n", cpuID, CntCPU);
244         exit(1);
245       }
246 
247       (TotalTimePerCPU[nodeID*CntCPU + cpuID]) += diff;
248       (TotalDataCntPerCPU[nodeID*CntCPU + cpuID])++;
249       cnt++;
250     }
251     fclose(fp);
252 #ifdef __DEBUG__
253     printf("total cnt =%d\n", cnt);
254 #endif
255   }
256 
257   int j=0;
258 #ifdef __DEBUG__
259   for(nodeID=0; nodeID<CntNode; nodeID++)
260   {
261     for(j=0; j<CntCPU-1; j++)
262     {
263       printf("Node[%d:CPU%02d]: cnt=%d\t total=%lf\t"
264           " avg=%lf\n", nodeID, j,
```

```
265                 (TotalDataCntPerCPU[nodeID*CntCPU + j]),
266                 (TotalTimePerCPU[nodeID*CntCPU + j]),
267               (double)( (TotalTimePerCPU[nodeID*CntCPU +j]) /
268                ((TotalDataCntPerCPU[nodeID*CntCPU+j])*1.0f)) );
269     }
270   }
271 #endif
272
273   int idx_node=0;
274   for(nodeID=0, idx_node=0; nodeID<CntNode; nodeID++)
275   {
276     for(j=0; j<CntCPU-1; j++)
277     {
278       //get nodeID to which the cpu belongs
279       idx_node = getNodeID(j);
280 #ifdef __DEBUG__
281       printf("CPU %d on Node %d\n", j, idx_node);
282 #endif
283       if ( idx_node !=-1)
284       {
285         TotalTimePerNode[nodeID*CntNode+idx_node] +=
286           TotalTimePerCPU[nodeID*CntCPU + j] /
287            ((TotalDataCntPerCPU[nodeID*CntCPU+j])*1.0f);
288       }
289     }
290   } // for(nodeID=0; nodeID<CntNode; nodeID++)
291
292   for(nodeID=0, idx_node=0; nodeID<CntNode; nodeID++)
293   {
294     for(idx_node=0; idx_node<CntNode; idx_node++)
295     {
296       TotalTimePerNode[nodeID*CntNode+idx_node] /=
297        (TotalCPUPerNode[idx_node]*1.0f);
298 #ifdef __DEBUG__
299       printf("TotalTimePerNode[%d][%d] = %lf\n", nodeID, idx_node,
300         TotalTimePerNode[nodeID*CntNode+idx_node]  );
301 #endif
302     }
303   }
304
305   int i=0;
306   for(i=0; i<CntNode; i++)
307   {
308     int idx_min=0, idx_max=0;
309     idx_min = getMin(&TotalTimePerNode[i*CntNode], CntNode);
310     idx_max = getMax(&TotalTimePerNode[i*CntNode], CntNode);
311
312     double min=.0f, max=.0f;
313     for(j=0; j<CntNode; j++)
314     {
315       distanceTable[i*CntNode+j] =
316        (int)((TotalTimePerNode[i*CntNode+j]*10.0f/
317         TotalTimePerNode[i*CntNode+idx_min]) + 0.5f);
318 #ifdef __DEBUG__
319       printf("TotalTimePerNode[%d][%d] = %lf, adjusted val=%lf,
320        distance=%u\n", i, j, TotalTimePerNode[i*CntNode+j],
```

```
321         ( TotalTimePerNode[i*CntNode+j]*10.0f/
322           TotalTimePerNode[i*CntNode+idx_min] ),
323         (int)(distanceTable[i*CntNode+j])  );
324 #endif
325     }
326   }
327 }
328 
329 int main(int argc, char** argv)
330 {
331   char* filename[MAX_DATAFILE];
332 
333   if ( argc != MAX_DATAFILE+1 )
334   {
335     fprintf(stderr, "./generateHeader datafile_0 datafile_1 "
336             "datafile_2 datafile_3\n");
337     exit(1);
338   }
339 
340   CntCPU = getConfiguredCPUs()+1;
341   CntNode = numa_max_node()+1;
342 
343   TotalTimePerCPU = (double*)malloc(sizeof(double) * CntCPU * CntNode);
344   TotalDataCntPerCPU = (int*)malloc(sizeof(int) * CntCPU * CntNode);
345   TotalCPUPerNode = (int*)malloc(sizeof(int)*CntNode);
346 
347   TotalTimePerNode = (double*)malloc(sizeof(double) * CntNode * CntNode);
348   distanceTable = (int*)malloc(sizeof(int)*CntNode*CntNode);
349 
350   if ( NULL == TotalTimePerCPU || NULL == TotalTimePerNode ||
351     NULL == TotalDataCntPerCPU )
352   {
353     fprintf(stderr, "fail to malloc\n");
354     exit(1);
355   }
356   memset(TotalTimePerCPU, '\0', sizeof(double) * CntCPU * CntNode);
357   memset(TotalDataCntPerCPU, '\0', sizeof(unsigned int) * CntCPU * CntNode);
358   memset(TotalCPUPerNode, '\0', sizeof(int)*CntNode);
359 
360   memset(TotalTimePerNode, '\0', sizeof(double) * CntNode * CntNode);
361   memset(distanceTable, '\0', sizeof(int) * CntNode * CntNode);
362 
363   int i=0;
364   for(i=0; i<MAX_DATAFILE; i++)
365   {
366     filename[i] = argv[i+1];
367   }
368   getTotalCPUPerNode();
369 
370   generate(filename);
371 
372   saveDistanceTable();
373 
374 printf("done\n");
375   return 0;
376 }
```

377

D.1.3 Determining the total virtual memory usage

This application is derived from source code[2] on the Internet. It has been changed to use a number of static arrays instead of calling the *malloc()* function. Using */proc/pid/status* (*pid* is the process ID), we can collect information regarding memory, especially peak virtual memory and data area usage, and current virtual memory usage, and so on.

Source 31

```
1  #define _GNU_SOURCE
2  #include <sys/types.h>
3  #include <sys/wait.h>
4  #include <signal.h>
5  #include <stdio.h>
6  #include <unistd.h>
7  #include <stdlib.h>
8  #include <string.h>
9
10 #define PATH_MAX 2048
11
12 int child_pid;
13
14 static int main_loop(char *pidstatus)
15 {
16   char linetemp[128];
17   char *line = linetemp;
18   char vmsize[128];
19   char vmpeak[128];
20   char vmrss[128];
21   char vmhwm[128];
22   char vmdata[128];
23   char cpuallowed[128];
24   char memallowed[128];
25
26   size_t len;
27
28   FILE *f;
29
30   memset(linetemp, '\0', 128);
31   memset(vmsize, '\0', 128);
32   memset(vmpeak, '\0', 128);
33   memset(vmrss, '\0', 128);
34   memset(vmhwm, '\0', 128);
35   memset(vmdata, '\0', 128);
36   memset(cpuallowed, '\0', 128);
37   memset(memallowed, '\0', 128);
38   len = 128;
39
40   f = fopen(pidstatus, "r");
```

[2] http://locklessinc.com/articles/memory_usage/

```
41    if (!f) return 1;
42    int idx=0;
43
44    /* Read memory size data from /proc/pid/status */
45    while (idx<7)
46    {
47      if (getline(&line, &len, f) == -1)
48      {
49        /* Some of the information isn't there, die */
50        return 1;
51      }
52
53      /* Find VmPeak */
54      if (!strncmp(line, "VmPeak:", 7))
55      {
56        strncpy(vmpeak, &line[7], strlen(&line[7]));
57        idx++;
58      }
59
60      /* Find VmSize */
61      else if (!strncmp(line, "VmSize:", 7))
62      {
63        strncpy(vmsize, &line[7], strlen(&line[7]));
64        idx++;
65      }
66
67      /* Find VmRSS */
68      else if (!strncmp(line, "VmRSS:", 6))
69      {
70        strncpy(vmrss, &line[6], strlen(&line[6]));
71        idx++;
72      }
73
74      /* Find VmHWM */
75      else if (!strncmp(line, "VmHWM:", 6))
76      {
77        strncpy(vmhwm, &line[6], strlen(&line[6]));
78        idx++;
79      }
80      /* Find VmData */
81      else if (!strncmp(line, "VmData:", 7))
82      {
83        strncpy(vmdata, &line[7], strlen(&line[7]));
84        idx++;
85      }
86      else if (!strncmp(line, "Cpus_allowed_list:", 18))
87      {
88        strncpy(cpuallowed, &line[18], strlen(&line[18]));
89        idx++;
90      }
91      else if (!strncmp(line, "Mems_allowed_list:", 18))
92      {
93        strncpy(memallowed, &line[strlen("Mems_allowed_list:")],
94         strlen(&line[18]));
95        idx++;
96      }
```

```
97    }
98    fclose(f);
99
100   /* Get rid of " kB\n"*/
101   len = strlen(vmsize);
102   vmsize[len - 4] = 0;
103   len = strlen(vmpeak);
104   vmpeak[len - 4] = 0;
105   len = strlen(vmrss);
106   vmrss[len - 4] = 0;
107   len = strlen(vmhwm);
108   vmhwm[len - 4] = 0;
109   len = strlen(vmdata);
110   vmdata[len - 4] = 0;
111
112   len = strlen(cpuallowed);
113   cpuallowed[len - 1] = 0;
114   len = strlen(memallowed);
115   memallowed[len - 1] = 0;
116
117   /* Output results to stderr */
118   fprintf(stderr, "%s\t%s\t%s\t%s\t%s\n",
119     vmsize, vmpeak, vmrss, vmhwm, vmdata/*, cpuallowed, memallowed*/);
120
121   /* Success */
122   return 0;
123 }
124
125
126 int main(int argc, char **argv)
127 {
128   char buf[1024];
129   child_pid = fork();
130
131   if (0 == child_pid)
132     execvp(argv[1], &argv[1]);
133   else
134   {
135     snprintf(buf, PATH_MAX, "/proc/%d/status", child_pid);
136
137     /* Continual scan of proc */
138     while (waitpid(child_pid, NULL, WNOHANG) == 0)
139     {
140       if ( main_loop(buf) )  break;
141       usleep(100000);
142     }
143   }
144
145   return 0;
146 }
```

D.2 Examples Of Conventional Allocators

We needed to implement some conventional memory allocation algorithms because it is hard to find the source codes of these algorithms, that can be used on our target machine and operating system. Here we provide a core function of these allocators.

D.2.1 Best-fit

The source code below is the core function of the implementation of *Best-fit*.

Source 32 BestFit()

```
1  void* BestFit(size_t _size)
2  {
3    chunk_t* chunk = FreeList;
4    size_t smallest = ~(unsigned long long)0;
5    chunk_t* BestOne = NULL;
6
7    while( chunk != NULL )
8    {
9      if ( chunk->size >= _size && smallest > chunk->size )
10     {
11       smallest = chunk->size;
12       BestOne = chunk;
13     }
14     chunk = GET_NEXT(chunk);
15   }
16
17   if ( NULL != BestOne )
18     chunk = BestOne;
19   else
20     return IncreaseHeap(_size);
21
22   //Split
23   if ( chunk->size >=
24       (_size + (2*HEADER_OVERHEAD) + MINIMUM_CHUNK_SIZE) )
25   {
26     chunk->size -= _size;
27     SET_TAILER(chunk, chunk->size);
28
29     chunk_t* ret = (chunk_t*)GET_P_NEXT(chunk);
30     InitChunk(ret, _size);
31
32     chunk = ret;
33   }
34   else
35   {
36     SET_USE_CHUNK(chunk);
37     ExtractChunk(chunk);
38   }
39   return chunk;
40 }
```

D.2.2 First-fiT

The source code below is a core function of the implementation of *First-fit*.

Source 33 FirstFit()

```
void* FirstFit(size_t _size)
{
  chunk_t* chunk = FreeList;

  while( chunk != NULL && chunk->size < _size )
    chunk = GET_NEXT(chunk);

  if ( NULL == chunk ) return IncreaseHeap(_size);

  //Split
  if ( chunk->size >=
      (_size + (2*HEADER_OVERHEAD) + MINIMUM_CHUNK_SIZE) )
  {
    chunk->size -= _size;
    SET_TAILER(chunk, chunk->size);

    chunk_t* ret = (chunk_t*)GET_P_NEXT(chunk);
    InitChunk(ret, _size);

    chunk = ret;
  }
  else
  {
    SET_USE_CHUNK(chunk);
    ExtractChunk(chunk);
  }
  return chunk;
}
```

D.2.3 Half-fit

The source code below is a core function of the implementation of *Half-fit*.

Source 34 HalfFit()

```
static __inline__ void* HalfFit(size_t _size)
{
  chunk_t* chunk = NULL;
  int fl=0;

  chunk = FindSuitableBlock(_size, &fl);
  if ( NULL == chunk ) return IncreaseHeap(_size);

  //Split
  if ( chunk->size - _size >=  MINIMUM_CHUNK_SIZE )
  {
    chunk->size -= _size;
    SET_TAILER(chunk, chunk->size);

    chunk_t* ret = (chunk_t*)GET_P_NEXT(chunk);
```

```
16        InitChunk(ret, _size);
17
18        ExtractChunk(chunk, fl);
19        FindIndex(chunk->size, &fl);
20        InsertChunk(chunk, fl);
21
22        chunk = ret;
23      }
24      else
25      {
26        SET_USE_CHUNK(chunk);
27        ExtractChunk(chunk, fl);
28      }
29      return chunk;
30    }
```

Abbreviations

API	Application Programming Interface
CAS	Compare and Swap
ccNUMA	Cache Coherent Non-Uniform Memory Access
DSA	Dynamic storage allocation
EDF	Earliest deadline first
FP	Fixed-task priority
MMU	Memory Management Unit
NASA	ccNUMA-Aware dynamic Storage Allocation algorithm
nMART	Node-oriented dynamic Memory Allocation algorithm for Real-Time systems on ccNUMA architectures
NUMA	Non-Uniform Memory Access
RTS	Real-Time System
SLIT	System Locality Information Table
SRAT	System Resource Affinity Table
UMA	Uniform Memory Access
WCET	Worst-Case Execution Time

References

[Apache Software Foundation, 2013] Apache Software Foundation (2013). Memory management with pools. "http://www.fmc-modeling.org/category/projects/apache/amp/3_3Extending_Apache.html".

[Banús et al., 2002] Banús, J. M., Arenas, A., and Labarta, J. (2002). An efficient scheme to allocate soft-aperiodic tasks in multiprocessor hard real-time systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 2*, PDPTA '02, pages 809–815. CSREA Press.

[Barrett and Zorn, 1993] Barrett, D. A. and Zorn, B. G. (1993). Using lifetime predictors to improve memory allocation performance. *SIGPLAN Not.*, 28(6):187–196.

[Bays, 1977] Bays, C. (1977). A comparison of next-fit, first-fit, and best-fit. *Commun. ACM*, 20(3):191–192.

[Berger et al., 2000] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. (2000). Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128.

[Bohra and Gabber, 2001] Bohra, A. and Gabber, E. (2001). Are mallocs free of fragmentation? In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 105–117, Berkeley, CA, USA. USENIX Association.

[Bolosky and Scott, 1993] Bolosky, W. J. and Scott, M. L. (1993). False sharing and its effect on shared memory performance. In *In Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 57–71.

[Brent, 1989] Brent, R. P. (1989). Efficient implementation of the first-fit strategy for dynamic storage allocation. *ACM Trans. Program. Lang. Syst.*, 11(3):388–403.

[Broquedis et al., 2010] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc: A generic framework for managing hardware affinities in hpc applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180 –186.

[Burns and Wellings, 2001] Burns, A. and Wellings, A. J. (2001). *Real-Time Systems and Programming Languages*. Addison Wesley, 3rd edition.

[Chen et al., 2005] Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., and Iyer, R. (2005). Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 378 – 387, Washington, DC, USA. IEEE Computer Society.

[Chowdhury and Srimani, 1987] Chowdhury, S. K. and Srimani, P. K. (1987). Worst case performance of weighted buddy systems. *Acta Informatica*, 24:555–564. 10.1007/BF00263294.

[Confessore et al., 2001] Confessore, G., Dell'Olmo, P., and Giordani, S. (2001). An approximation result for a periodic allocation problem. *Discrete Applied Mathematics*, 112(1–3):53–72.

[Cowan et al., 2000] Cowan, C., Wagle, F., Pu, C., Beattie, S., and Walpole, J. (2000). Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 119–129.

[Crespo et al., 2006] Crespo, A., Ripoll, I., and Masmano, M. (2006). Dynamic memory management for embedded real-time systems. 225:195–204.

[Daniel P. Bovet, 2005] Daniel P. Bovet, M. C. (2005). *Understanding the Linux Kernel*, chapter Chapter 2. O'Reilly, 3rd edition.

[Douglas, 2011] Douglas, N. (2011). nedmalloc. http://www.nedprod.com/programs/portable/nedmalloc.

[Edge, 2009] Edge, J. (2009). Perfcounters added to the mainline. http://lwn.net/Articles/336542/.

[Ferreira et al., 2011] Ferreira, T., Matias, R., Macedo, A., and Araujo, L. (2011). An experimental study on memory allocators in multicore and multithreaded applications. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on*, pages 92 –98.

[Fredkin, 1960] Fredkin, E. (1960). Trie memory. *Commun. ACM*, 3(9):490–499.

[FSF, 2012a] FSF, F. s. f. (2012a). Glibc, the gnu c library. "http://www.gnu.org/software/libc/libc.html".

[FSF, 2012b] FSF, F. s. f. (2012b). The gnu c++ library manual. "http://gcc.gnu.org/onlinedocs/libstdc++".

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., San Francisco, CA.

[Gergov, 1996] Gergov, J. (1996). Approximation algorithms for dynamic storage allocation. In *Algorithms — ESA '96*, volume 1136, pages 52–61. Springer Berlin / Heidelberg.

[Gergov, 1999] Gergov, J. (1999). Algorithms for compile-time memory optimization. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '99, pages 907–908, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Gloger, 2001] Gloger, W. (2001). Dynamic memory allocator implementations in linux system libraries. "http://www.dent.med.uni-muenchen.de/~wmglo/malloc\discretionary{-}{}{}slides.html".

[Gloger, 2006] Gloger, W. (2006). ptmalloc2. "http://www.malloc.de/en/".

[Grunwald et al., 1993] Grunwald, D., Zorn, B., and Henderson, R. (1993). Improving the cache locality of memory allocation. *SIGPLAN Not.*, 28(6):177–186.

[Hasan and Chang, 2005] Hasan, Y. and Chang, M. (2005). A study of best-fit memory allocators. *Computer Languages, Systems & Structures*, 31(1):35 – 48.

[Hasan et al., 2010] Hasan, Y., Chen, W.-M., Chang, J. M., and Gharaibeh, B. M. (2010). Upper bounds for dynamic memory allocation. *IEEE Trans. Comput.*, 59(4):468–477.

[Hewlett-Packard Corporation, 2012] Hewlett-Packard Corporation (2012). HP ProLiant DL980 G7 server with HP PREMA Architecture PREMA Architecture. http://h20195.www2.hp.com/v2/GetPDF.aspx/4AA3-0643ENW.pdf. Technical Whitepaper.

[Hewlett-Packard Corporation et al., 2011] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation (2011). Advanced configuration and power interface specification. http://acpi.info/DOWNLOADS/ACPIspec40a.pdf.

[Hirschberg, 1973] Hirschberg, D. S. (1973). A class of dynamic memory allocation algorithms. *Commun. ACM*, 16(10):615–618.

[Hyde and Fleisch, 1996] Hyde, R. L. and Fleisch, B. D. (1996). An analysis of degenerate sharing and false coherence. *J. Parallel Distrib. Comput.*, 34(2):183–195.

[Jeremiassen and Eggers, 1995] Jeremiassen, T. E. and Eggers, S. J. (1995). Reducing false sharing on shared memory multiprocessors through compile time data transformations. *SIGPLAN Not.*, 30(8):179–188.

[Johnstone and Wilson, 1998] Johnstone, M. S. and Wilson, P. R. (1998). The memory fragmentation problem: solved? *SIGPLAN Not.*, 34(3):26–36.

[Kaminski, 2009] Kaminski, P. (2009). Numa aware heap memory manager. "http://amddevcentral.com/Assets/NUMA_aware_heap_memory_manager_article_final.pdf".

[Kingsley, 1982] Kingsley, C. (1982). Description of a very fast storage allocator.

[Kleen, 2005] Kleen, A. (2005). *A NUMA API for Linux*. Novel Inc. accessed on September 2011.

[Knuth, 1997] Knuth, D. (1997). *The art of computer programming: Fundamental Algorithms*, volume 1. addison-Wesley, 2 edition.

[Larson and Krishnan, 1998] Larson, P.-k. and Krishnan, M. (1998). Memory allocation for long-running server applications. *SIGPLAN Not.*, 34(3):176–185.

[Lea, 1996] Lea, D. (1996). A memory allocator. "http://g.oswego.edu/dl/html/malloc.html". Unix/Mail December, 1996.

[Linus Torvalds, 2011] Linus Torvalds, e. (2011). Source codes of linux kernel v3.0.4. "http://lxr.linux.no/linux+v3.0.4/".

[Liu and Berger, 2011] Liu, T. and Berger, E. D. (2011). Sheriff: precise detection and automatic mitigation of false sharing. *SIGPLAN Not.*, 46(10):3–18.

[Luby et al., 1994] Luby, M. G., Naor, J. S., and Orda, A. (1994). Tight bounds for dynamic storage allocation. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, SODA '94, pages 724–732, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Majo and Gross, 2011] Majo, Z. and Gross, T. R. (2011). Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 1–10, New York, NY, USA. ACM.

[Marathe and Mueller, 2006] Marathe, J. and Mueller, F. (2006). Hardware profile-guided automatic page placement for ccnuma systems. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 90–99, New York, NY, USA. ACM.

[Marchand et al., 2007] Marchand, A., Balbastre, P., Ripoll, I., Masmano, M., and Crespo, A. (2007). Memory resource management for real-time systems. In *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 201 –210.

[Masmano, 2012] Masmano (2012). The lastest version of TLSF source. http://wks.gii.upv.es/tlsf/files/src/TLSF-2.4.6.tbz2.

[Masmano et al., 2008a] Masmano, M., Ripoll, I., Balbastre, P., and Crespo, A. (2008a). A constant-time dynamic storage allocator for real-time systems. *Real-Time Systems*, 40(2):149–179.

[Masmano et al., 2003] Masmano, M., Ripoll, I., and Crespo, A. (2003). Dynamic storage allocation for real-time embedded systems. *Proc. of Real-Time System Simposium WIP*.

[Masmano et al., 2008b] Masmano, M., Ripoll, I., Real, J., Crespo, A., and Wellings, A. (2008b). Implementation of a constant-time dynamic storage allocator. *Software: Practice and Experience*, 38(10):995–1026.

[McCamant and Ernst, 2007] McCamant, S. and Ernst, M. D. (2007). A simulation-based proof technique for dynamic information flow. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, PLAS '07, pages 41–46, New York, NY, USA. ACM.

[McCurdy and Vetter, 2010] McCurdy, C. and Vetter, J. (2010). Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 87–96.

[Mehta et al., 1997] Mehta, H., Owens, R., Irwin, M., Chen, R., and Ghosh, D. (1997). Techniques for low energy software. In *Low Power Electronics and Design, 1997. Proceedings., 1997 International Symposium on*, pages 72–75.

[MicroQuill, 2012] MicroQuill (2012). shbench benchmark tool. http://www.microquill.com.

[Mochel, 2005] Mochel, P. (2005). The sysfs filesystem. In *Linux Symposium*, pages 313–326, Ottawa, Ontario, Canada.

[Molnar, 2009] Molnar, I. (2009). Performance counters for linux, v8. http://lwn.net/Articles/336542/.

[Nethercote and Mycroft, 2002] Nethercote, N. and Mycroft, A. (2002). The cache behaviour of large lazy functional programs on stock hardware. *SIGPLAN Not.*, 38(2 supplement):44–55.

[Nilsen and Gao, 1995] Nilsen, K. and Gao, H. (1995). The real-time behavior of dynamic memory management in c++. In *Real-Time Technology and Applications Symposium, 1995. Proceedings*, pages 142–153.

[Ogasawara, 1995] Ogasawara, T. (1995). An algorithm with constant execution time for dynamic storage allocation. In *RTCSA '95: Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, pages 21–25, Washington, DC, USA. IEEE Computer Society.

[Ogasawara, 2009] Ogasawara, T. (2009). Numa-aware memory manager with dominant-thread-based copying gc. *SIGPLAN Not.*, 44(10):377–390.

[Oracle Inc., 2013] Oracle Inc. (2013). Memory architecture. "http://docs.oracle.com/cd/E14072_01/server.112/e10713/memory.htm".

[Ortiz and Santiago, 2008] Ortiz, D. and Santiago, N. (2008). Impact of source code optimizations on power consumption of embedded systems. In *Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on*, pages 133–136.

[Page and Hagins, 1986] Page, I. and Hagins, J. (1986). Improving the performance of buddy systems. *Computers, IEEE Transactions on*, C-35(5):441 –447.

[Polishchuk et al., 2007] Polishchuk, M., Liblit, B., and Schulze, C. W. (2007). Dynamic heap type inference for program understanding and debugging. *SIGPLAN Not.*, 42(1):39–46.

[Puaut, 2002] Puaut, I. (2002). Real-Time Performance of Dynamic Memory Allocation Algorithms. In *ECRTS '02: Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 41–49, Washington, DC, USA. IEEE Computer Society.

[Puaut and Hardy, 2007] Puaut, I. and Hardy, D. (2007). Predictable paging in real-time systems: A compiler approach. In *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 169 –178.

[Randell, 1969] Randell, B. (1969). A note on storage fragmentation and program segmentation. *Commun. ACM*, 12(7):365–ff.

[Robertson et al., 2003] Robertson, W., Kruegel, C., Mutz, D., and Valeur, F. (2003). Run-time detection of heap-based overflows. In *Proceedings of the 17th USENIX conference on System administration*, LISA '03, pages 51–60, Berkeley, CA, USA. USENIX Association.

[Robson, 1980] Robson, J. (1980). Storage allocation is np-hard. *Information Processing Letters*, 11(3):119–125.

[Robson, 1971] Robson, J. M. (1971). An estimate of the store size necessary for dynamic storage allocation. *J. ACM*, 18(3):416–423.

[Robson, 1977] Robson, J. M. (1977). Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244.

[Sanjay Ghemawat, 2010] Sanjay Ghemawat, P. M. (2010). Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[SGI, 2004] SGI (2004). Standard template library programmer's guide: Allocators. "http://www.sgi.com/tech/stl/Allocators.html".

[Shen and Peterson, 1974] Shen, K. K. and Peterson, J. L. (1974). A weighted buddy method for dynamic storage allocation. *Commun. ACM*, 17(10):558–562.

[Shore, 1975] Shore, J. E. (1975). On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Commun. ACM*, 18(8):433–440.

[Standish, 1980] Standish, T. A. (1980). *Data Structure Techniques*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Stephenson, 1983] Stephenson, C. J. (1983). New methods for dynamic storage allocation (fast fits). *SIGOPS Oper. Syst. Rev.*, 17(5):30–32.

[Sybase Inc., 2013] Sybase Inc. (2013). Configuration parameters that affect memory allocation. "http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.ase_15.0.sag2/html/sag2/sag274.htm".

[Tao et al., 2008] Tao, J., Kunze, M., and Karl, W. (2008). Evaluating the cache architecture of multicore processors. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 12 –19.

[Tikir and Hollingsworth, 2008] Tikir, M. and Hollingsworth, J. (2008). Hardware monitors for dynamic page migration. *Journal of Parallel and Distributed Computing*, 68(9):1186–1200.

[Vee and Hsu, 1999] Vee, V.-Y. and Hsu, W.-J. (1999). A scalable and efficient storage allocator on shared memory multiprocessors. In *Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks*, ISPAN '99, pages 230–, Washington, DC, USA. IEEE Computer Society.

[Weber, 2001] Weber, F. (2001). Amd's next generation microprocessor architecture. Presented in Microprocessor Forum at San Jose, California.

[Weidendorfer et al., 2004] Weidendorfer, J., Kowarschik, M., and Trinitis, C. (2004). A tool suite for simulation based analysis of memory access behavior. In *In Proceedings of International Conference on Computational Science*, pages 440–447. Springer.

[Wellings et al., 2010] Wellings, A. J., Malik, A. H., Audsley, N. C., and Burns, A. (2010). Ada and cc-numa architectures what can be achieved with ada 2005? *Ada Lett.*, 30(1):125–134.

[Wilson et al., 1995a] Wilson, P., Johnstone, M., Neely, M., and Boles, D. (1995a). Memory allocation policies reconsidered. Technical report, Technical report, University of Texas at Austin Department of Computer Sciences.

[Wilson et al., 1995b] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. (1995b). Dynamic Storage Allocation: A Survey and Critical Review. In *IWMM '95: Proceedings of the International Workshop on Memory Management*, pages 1–116, London, UK. Springer-Verlag.

[Zhou and Petrov, 2011] Zhou, X. and Petrov, P. (2011). Towards virtual memory support in real-time and memory-constrained embedded applications: the interval page table. *Computers Digital Techniques, IET*, 5(4):287 –295.

[Zorn and Grunwald, 1992] Zorn, B. and Grunwald, D. (1992). Empirical measurements of six allocation-intensive c programs. *SIGPLAN Not.*, 27(12):71–80.

[Zorn and Grunwald, 1994] Zorn, B. and Grunwald, D. (1994). Evaluating models of memory allocation. *ACM Trans. Model. Comput. Simul.*, 4(1):107–131.