INTELLIGENT MEMORY MANAGER: TOWARDS IMPROVING THE LOCALITY BEHAVIOR OF ALLOCATION-INTENSIVE APPLICATIONS

Mehran Rezaei, B.S., M.S.

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

May 2004

APPROVED:

Krishna M. Kavi, Major Professor and
Chair of the Dept. of Computer Science

Robert P. Brazile, Committee Member
and Graduate Coordinator

Steve Tate, Committee Member

Kathleen Swigger, Committee Member

Oscar Garcia, Dean of College of Engineering

Sandra L. Terrel, Interim Dean of
Toulouse School of Graduate School
Rezaei, Mehran, Intelligent Memory Manager: Towards improving the locality behavior of allocation-intensive applications. Doctor of Philosophy (Computer Science), May 2004, 110 pp., 26 tables, 38 figures, references, 79 titles.

Dynamic memory management required by allocation-intensive (i.e., Object Oriented and linked data structured) applications has led to a large number of research trends. Memory performance due to the cache misses in these applications continues to lag in terms of execution cycles as ever increasing CPU-Memory speed gap continues to grow.

Sophisticated prefetching techniques, data relocations, and multithreaded architectures have tried to address memory latency. These techniques are not completely successful since they require either extra hardware/software in the system or special properties in the applications. Software needed for prefetching and data relocation strategies, aimed to improve cache performance, pollutes the cache so that the technique itself becomes counter-productive. On the other hand, extra hardware complexity needed in multithreaded architectures decelerates CPU’s clock, since “Simpler is Faster”.

This dissertation, directed to seek the cause of poor locality behavior of allocation-intensive applications, studies allocators and their impact on the cache performance of these applications. Our study concludes that service functions, in general, and memory management functions, in particular, entangle with application’s code and
become the major cause of cache pollution. In this dissertation, we present a novel
technique that transfers the allocation and de-allocation functions entirely to a separate
processor residing in chip with DRAM (Intelligent Memory Manager). Our empirical
results show that, on average, 60% of the cache misses caused by allocation and de-
allocation service functions are eliminated using our technique.

We also show that internal fragmentation, extra memory over-allocated by the
allocators, counters special locality of applications. We introduce “hybrid,” an exact fit
allocator, which results in 25% cache miss reduction due to minimizing the internal
fragmentation. Moreover, this work indicates that external fragmentation, inability to use
the existing free space, indirectly affects the execution performance. We propose address
ordered and segregated binary tree allocators that exhibit high storage utilization and
moderate execution performance to compare with existing allocators.
Copyright 2004
by
Mehran Rezaei
ACKNOWLEDGMENTS

In the name of Allah, the most merciful, the most compassionate. All praise is due to Allah. My thanks, first and foremost, belongs to Almighty Allah, who has given me wisdom, courage, and strength.

I am filled with joy and excitement. Now, when I look back at the last six years of my life, the thoughts and feelings in my heart are only about the countless people who contributed to my Ph.D. work in variety of ways.

Two individuals have had the main role in guiding and supporting me throughout my Ph.D. work. Krishna Kavi, my advisor, has devoted his intellect and time to form, most importantly, my career’s character. Dr. Kavi has advised and guided me to develop the tendency for research in Computer Architecture. Professor Zabi Rezaee, my dear uncle, whose advice always guided me to make sharp decisions, has supported me by all means during last eight years of my life in America. My very best thanks to them, whom I owe a depth of gratitude.

I am grateful to my mother, Ms. Monireh Rezaei; my aunts, Miss Zahra and Mansoureh Rezaei; and my brother, Mr. Mohammad Rezaei; who have cheered me on the great moments and encouraged me when I needed spiritual boost. My dear brother has been giving persistent care to my mother since I am away from my family. I certainly fail to word my appreciation for them, but I know how blessed I am to have such family.

The Islamic Center of Momin, in Irving (Texas), has been the center of my hope during the past three years. After a week of frustration and hard work, whenever I walked into the center, my dear brothers cheered me. The best times of my life, over the past three years, were when I sat among my brothers in the Islamic Center of Momin to listen to speeches of Mulana Shemshad, commemorate the sad occasions, celebrate the happy events, eat the delicious spicy food, or chat with others. I thank each and every member of the Islamic Center of Momin for things that materialistically cannot be comprehended.

My Ph.D. is the outcome of six years of study and research. The first three years I was a Ph.D. student of Computer Engineering in University of Alabama In Huntsville. I would like
to thank Dr. Reza Adhami, Dr. Tom Martin, Dr. Emil Jovanov, and Dr. Mary Weisskopf who enlightened me with insights about research and Ph.D. work during the time in UAH.

I am especially thankful to Dr. Kathleen Swigger, Dr. Robert Brazile, and Dr. Steve Tate who have given me thoughts for my future research. They have devoted their times and given me the support in Ph.D. committee meetings in UNT.

I am indebted to Dr. Hyong-Shik Kim, Dr. Roberto Giorgi, Dr. Joseph Arul, Dr. Mohammad Aborizka, and the members of Computer Architecture research groups in UAH and UNT. The best part of my time in graduate school was when I discussed the educational, political, and social matters with my colleagues.

I would like to thank the staff of Electrical and Computer Engineering Department of UAH, Ms. Jacqueline Siniard, Pat Simth, and Helen Foster; and the staff of Computer Science and Engineering Department of UNT, Ms. Mary Grumbein, Sally Pettyjohn, Lisa Ayers, Pamela Vincent, and Kathy Bomar for all the times that they patiently took care of administration matters of my Ph.D. and listened to my complaints.

I am grateful to Ms. Threasa Kavi for her consistent encouragement. In our social and family gatherings, she and Dr. Kavi provided such friendly atmosphere where I and my colleagues felt at home. I thank her and Dr. Kavi for their friendship.

Saving the best for the last, I would like to, very especially, thank my father; he was martyred two decades ago, but his warm support and spiritual guidance, which are always within my heart, have helped me form my personal character towards all aspects of life.

My Ph.D. is dedicated to my leaders, Imam Mahdi (ATFS, and AS) and Prophet Jesus (AS), who will lead the world to full justice and freedom.

May this work be a glory to Almighty Allah.
CONTENTS

ACKNOWLEDGMENTS iii

LIST OF TABLES viii

LIST OF FIGURES x

Chapter

1 INTRODUCTION 1

2 ABT AND SBT: NEW MEMORY MANAGEMENT TECHNIQUES 9

2.1 Introduction ................................................................. 9

2.2 Levels of Memory Management System ............................... 10

2.3 Known Allocation Techniques ............................................ 11

2.4 New Allocation Techniques: Address Ordered and Segregated Binary
   Trees ................................................................. 14

2.4.1 Address Ordered Binary Tree (ABT) .............................. 14

2.4.2 Segregated Binary Tree (SBT) ..................................... 17

2.5 Empirical Results .......................................................... 17

2.5.1 Experimental Framework ............................................. 18

2.5.2 Comparison of ABT and SqF ....................................... 20

2.5.3 The Effect of Segregation on Allocators ......................... 29

2.6 Conclusions ............................................................... 34
3 AN EXACT FIT ALLOCATOR: MINIMIZING INTERNAL FRAGMENTATION FOR BETTER CACHE PERFORMANCE

3.1 Introduction ............................................................................. 36
3.2 Summary of Benchmarks and Allocators ................................. 37
3.3 Different Allocation Patterns of Applications ....................... 39
3.4 Memory Fragmentations .......................................................... 42
3.5 Internal Fragmentation and its Impact on Cache Performance ... 46
3.6 An Exact Fit Allocator ............................................................. 50
3.7 Empirical Results ................................................................. 51
3.8 Conclusions ........................................................................... 52

4 INTELLIGENT MEMORY SYSTEM

4.1 Introduction ............................................................................. 56
4.2 Intelligent Memory System ....................................................... 58
4.2.1 Architecture of Intelligent Memory System ...................... 59
4.2.2 Compiler and Operating System of Intelligent Memory System . 62
4.3 Applicability of Intelligent Memory System ........................ 63
4.3.1 Runtime Libraries ............................................................. 64
4.3.2 DSP applications ............................................................. 64
4.3.3 Prefetching, Jump Pointers, Memory Relocation, and Stream Buffers ........................................................................... 65
4.4 Related Work ........................................................................ 66
4.4.1 Tiles of PIMs ................................................................. 66
4.4.2 PIMs as Stand Alone Systems ........................................... 68
LIST OF TABLES

1.1 Improvement of Number of Transistors and Clock Cycle of Alpha Processors over time, Reproduction by permission of the author and Telematik magazine [39] ................................................................. 2
1.2 Memory Speed Improvements over time, Reproduction by permission of Morgan Kaufmann Publishers [24] ................................................................. 2

2.1 Benchmark Description ................................................................. 18
2.2 Benchmark Statistics .................................................................. 19
2.3 Address Ordered Binary Tree, First Fit ......................................... 22
2.4 Sequential Fit, First Fit ................................................................. 22
2.5 Address Ordered Binary Tree, Better Fit ......................................... 24
2.6 Sequential Fit, Best Fit ................................................................. 24
2.7 Address Ordered Binary Tree, First Fit with Coalescing .................. 26
2.8 Sequential Fit, First Fit with Coalescing ......................................... 26
2.9 Address Ordered Binary Tree, Better Fit with Coalescing ................. 28
2.10 Sequential Fit, Best Fit with Coalescing ........................................ 28
2.11 Segregated Binary Tree, First Fit .................................................. 30
2.12 Segregated Fit, First Fit ............................................................... 30
2.13 Segregated Binary Tree, Better Fit ............................................... 32
2.14 Segregated Fit, Best Fit ............................................................... 32
2.15 Segregated Binary Tree, Better Fit with Free Standing List ............. 33
2.16 Segregated Binary Tree, Better Fit with Coalescing ..................... 33

3.1 Benchmark Description ................................................................. 37
5.1 Benchmark Description ....................................................... 77
5.2 Total Number of References for Conventional Configuration ............. 78
5.3 Total Number of References for IMM - Application ........................ 78
5.4 Total Number of References for IMM - Allocator .......................... 79

6.1 Truth table for input and output flip and route signals of each node of CBT,
with collaboration of address and size signals, Reproduction by permission of
IEEE [10] ................................................................................. 94
6.2 Simulated Processor’s Parameters ................................................. 96
6.3 Simulation results for Conv-Conf and IMEM ................................... 97
LIST OF FIGURES

1.1 CPU-Memory Speed Gap (Note that the y axis is log scaled, Reproduction by permission of Morgan Kaufmann Publishers [24] ........................................... 3

2.1 Memory Management Hierarchy, Reproduction by permission of the author and IRWIN publisher [15] ................................................................. 10
2.2 Memory area of a User Process ............................................................. 12

3.1 The Allocation Behavior of “ptc”, Ramp Behavior ................................. 40
3.2 The Allocation Behavior of “espresso”, Peak Behavior ......................... 41
3.3 The Allocation Behavior of “parser”, Plateau Behavior ......................... 41
3.4 Definitions of Fragmentation ............................................................... 42
3.5 Percentage of Fragmentation for boxed-sim with different allocators ....... 45
3.6 Percentage of Fragmentation for vpr with different allocators ............... 45
3.7 Levels of Memory Hierarchy in Conventional Architecture .................... 46
3.8 Internal fragmentation and its impact on locality .................................... 47
3.9 Allocated objects when there is no fragmentation ................................... 49
3.10 Total Number of Cache Misses, Cache size = 16 KBytes, Block size = 32 Bytes 53
3.11 Total Number of Cache Misses, Cache size = 16 KBytes, Block size = 64 Bytes 53
3.12 Total Number of Cache Misses, Cache size = 32 KBytes, Block size = 32 Bytes 54
3.13 Cache Miss Rates, Cache size = 16 KBytes, Block size = 32 Bytes ........... 54

4.1 Intelligent Memory System: Embedded DRAM configuration .................. 58
4.2 Intelligent Memory System: Extended Centralized Controller ................ 58
4.3 Intelligent Memory System Architecture, Extended Controller - eController . 60
4.4 Intelligent Memory System Architecture, using eDRAM instead of DRAM Arrays .......................................................... 60
4.5 Intelligent Memory System Architecture, Extended CPU - eCPU ......... 62
4.6 Typical example of a node architecture in a Tiles of PIMs systems .......... 67
4.7 System Interconnection in a Tiles of PIMs systems, Reproduction by permis-

sion of authors and ACM [65] ................................................. 67
4.8 Active Page Architecture (4 pages), Reproduction by permission of the au-

thors and ACM [56] ................................................................. 70
4.9 FlexRAM architecture, Reproduction by permission of authors and ACM [22] 70

5.1 IMM Framework: The use of two kernel processes for simulating IMM config-

uration .................................................................................. 76
5.2 Conv-Conf Cache Misses, Cache size = 32 KBytes, Cache Block size = 32 Bytes 81
5.3 IMM-application Cache Misses, Cache size = 32 KBytes, Cache Block size = 32 Bytes ......................................................... 81
5.4 Percentage of IMM-application cache miss improvement as to compare with
Conv-Conf , Cache size = 32 KBytes, Cache Block size = 32 Bytes . . . . 82
5.5 Conv-Conf Cache Misses, Cache size 32 KBytes, Cache Block size 64 Bytes . 84
5.6 IMM-application Cache Misses, Cache size 32 KBytes, Cache Block size 64 Bytes ................................................................. 84
5.7 IMM-allocator cache misses, 512 Bytes Direct Mapped Cache, 32 Bytes Block size ................................................................. 85

6.1 4 KByte Heap with 16 Byte Block and its 256 bit Bit-Vector ................. 90
6.2 Or-Gate CBT, to indicate the existence of large enough free chunk, Repro-
duction by permission of IEEE [10] .......................................... 91
6.3 And-Gate CBT, A mechanism which reveals the starting address of available
chunk, Reproduction by permission of IEEE [10] ......................... 92
6.4 Composed and decomposed flip and route signals with collaboration of address and size information, Reproduction by permission of IEEE [10].

6.5 An example of flipping the first four bits of a bit-vector using \textit{flip}, \textit{route}, \textit{address}, and \textit{size} signals, Reproduction by permission of IEEE [10].

6.6 Percentage of Performance Improvement of IMEM-Conf as compared with Conv-Conf.
CHAPTER 1
INTRODUCTION

Gordon E. Moore in 1965 had predicted that by 1975 the number of transistors on a single wafer would exceed 65,000 [52]. In 1965, Moore, presented a graph showing that the number of components on a chip would double every 18 months. His prediction was fortified by the statistical data from 1959 to 1965. It is amazing, Table 1.1 shows that his speculation still holds [39]. Other evidence forecasts that we may be even able to beat Moore’s Law [50]. Stephen Chou from Princeton University claims that his new invention, LADI (Laser Assisted Direct Input) - a new technology that may replace photolithography (the process of transferring geometric shapes onto a mask on the surface of a silicon wafer), increases chip’s density by a factor of 100.

On the other hand, Table 1.2 depicts the pace in memory speed’s improvement over the same period of time [24]. This table shows that the enhancement on Memory Row Access Time is 5% per annum. Figure 1.1 summarizes the observations concluded from the last two tables [24]. This figure shows a very known problem, so-called CPU-Memory Speed gap; how fast CPU can process data is no longer the problem. The main issue of concern in today’s computer system research is how fast data can be provided to CPU.

Based on the locality behavior of applications, to circumvent the CPU-Memory Speed Gap, researchers proposed deeper memory hierarchy and larger caches. But cache miss rate would never become zero, and consequently the Memory Wall a hurdle that limits the execution performance to the memory speed - not CPU speed, has been introduced [75, 78]. The problem is even more pronounced in Object Oriented and linked data structured applications, which do not have the same amount of locality as scientific applications do. In order to fully explore the problem and address the existing solutions, we study the computer’s
<table>
<thead>
<tr>
<th>Chip's name</th>
<th>21064 - EV4</th>
<th>21164 - EV5</th>
<th>21264 - EV6</th>
<th>21364 - EV7</th>
<th>21464 - EV8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>0.75 - 0.5 µm</td>
<td>0.5 - 0.35 µm</td>
<td>0.35 µm</td>
<td>0.18 µm</td>
<td>0.125 µm</td>
</tr>
<tr>
<td>Transistors</td>
<td>1.68 M</td>
<td>9 M</td>
<td>15 M</td>
<td>152 M</td>
<td>250 M</td>
</tr>
<tr>
<td>Frequency</td>
<td>150 - 275 MHz</td>
<td>300 - 600 MHz</td>
<td>0.6 - 1 GHz</td>
<td>1.2 - 1.4 GHz</td>
<td>1.2 - 2 GHz</td>
</tr>
<tr>
<td>Architecture</td>
<td>2-Way In-Order</td>
<td>4-Way In-Order</td>
<td>4-Way Out-Of-Order</td>
<td>4-Way Out-Of-Order System on a Chip</td>
<td>8-Way Out-Of-Order</td>
</tr>
</tbody>
</table>

Table 1.1: Improvement of Number of Transistors and Clock Cycle of Alpha Processors over time, Reproduction by permission of the author and Telematik magazine [39]

<table>
<thead>
<tr>
<th>Year of Introduction</th>
<th>Chip Size</th>
<th>Row Access Strobe (RAS)</th>
<th>Column Access Strobe (CAS)/ data transfer time (ns)</th>
<th>Cycle time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Slowest DRAM (ns)</td>
<td>Fastest DRAM (ns)</td>
<td>time (ns)</td>
</tr>
<tr>
<td>1980</td>
<td>64K bit</td>
<td>180</td>
<td>150</td>
<td>75</td>
</tr>
<tr>
<td>1983</td>
<td>256K bit</td>
<td>150</td>
<td>120</td>
<td>50</td>
</tr>
<tr>
<td>1986</td>
<td>1M bit</td>
<td>120</td>
<td>100</td>
<td>25</td>
</tr>
<tr>
<td>1989</td>
<td>4M bit</td>
<td>100</td>
<td>80</td>
<td>20</td>
</tr>
<tr>
<td>1992</td>
<td>16M bit</td>
<td>80</td>
<td>60</td>
<td>15</td>
</tr>
<tr>
<td>1996</td>
<td>64M bit</td>
<td>70</td>
<td>50</td>
<td>12</td>
</tr>
<tr>
<td>1998</td>
<td>128M bit</td>
<td>70</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>2000</td>
<td>256M bit</td>
<td>65</td>
<td>45</td>
<td>7</td>
</tr>
<tr>
<td>2002</td>
<td>512M bit</td>
<td>60</td>
<td>40</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1.2: Memory Speed Improvements over time, Reproduction by permission of Morgan Kaufmann Publishers [24]
memory under two categories:

**Hardware Oriented Research** which is concentrated on tolerating memory latency - time that the first byte of data needs to travel from memory to CPU, when a memory reference is issued. Multithreaded Architectures, Prefetching Engines, and Stream Buffers are among hardware oriented research trends that aim to hide memory latency. Multithreaded Architectures provide a low cost thread switch when a long latency operation such as a cache miss is encountered [33, 74]. Several Explicit Multithreaded architectures such as Interleaving, Blocking, Non-Blocking, and Simultaneous Multithreading have been proposed among which the latter seems to be more promising and attractive [73]. Explicit Multithreaded Architectures require special care from compiler or higher level programming for thread partitioning. Implicit Multithreaded Architectures, however, receive a window of instructions (similar to single threaded architectures) and try to speculatively schedule them to the threads [74]. As one can imagine, the hardware complexity of such systems prevents them from being adopted by the
industry. On the other hand, Simultaneous Multithreaded architectures, a member of Explicit Multithreading, has been put into practice by many companies [3, 20, 49]. Interleaving Multithreaded Techniques (IMT) issue an instruction from a different thread to the pipeline on each cycle [44]. Blocking Multithreaded (BMT) architectures keep on scheduling instructions from the same thread until a long latency operation or the end of the current thread occur [43]. Non-Blocking multithreaded architectures use fine grained threads so that within a thread there will be no need to context switch [32, 36]. In Simultaneous Multithreading (SMT), multiple instructions from several threads (several windows of instructions already being grouped as threads by compiler) are issued to the several functional units simultaneously [73, 74].

Prefetching engines prefetch the data of linked nodes ahead of time for linked data structured applications [13]. This is to assure that the data are in the cache when CPU needs to process them.

Stream Buffers are FIFO buffers with prefetching engines for buffering the stream data mainly for floating point applications [31, 51, 57]. These buffers are used to filter the cache data so that the cache pollution caused by stream data is removed.

All of the above techniques, some of which, although effective, require special properties in the applications. For example, to fully utilize multithreaded architecture, the compiler needs to identify thread level parallelism in the applications. Prefetching techniques assume that the prefetching latency can be fully covered by sufficient workload prior to the memory access to the prefetched data. Stream Buffers need to precisely extract stream data and manage the FIFO buffers. In addition to the special properties in the applications needed by these techniques, they add extra hardware to System on Chip implementations.

**Software Related Research.** Research in this category is involved with memory management issues such as allocation/de-allocation, relocation of objects, software prefetching,
and Jump Pointers. These techniques may require modifications to the Instruction Set of the architecture or be completely implemented in software. For example, Memory Forwarding, which relocates the objects of linked lists for better spatial locality, provides new load and store instructions that can be directed to the new locations of the objects transparently [48]. The new instructions, in Memory Forwarding technique, are provided to assure the correctness of the program. When an object is relocated, all the pointers to the object should be modified. It is difficult and rather time consuming, which counters the main purpose of the technique - improving the execution performance, to guarantee that all the pointers to the relocated object are reassigned with the new address. Memory Forwarding marks the previous location (so-called the forwarded bit of the memory location) of the object as indirect and changes its value to the new address of the object. The modified load (store) instruction would load the value from (store the value to) the forwarded location if the object was relocated. Three new instructions are provided to check the forwarded bit, read the value of any address just as normal load, and write to any location just as a normal store in addition to setting the forwarded bit to either zero or one.

Software Prefetching are mainly compiler-based techniques [47, 53]. Compiler, in these techniques, after analyzing the code, inserts the prefetch instructions in place where prefetching the data could eliminate cache misses.

These methods although improve the locality of applications, they do not achieve significant execution performance. For example, software prefetching techniques show 7% performance improvement, on average, for Olden benchmarks [47]. Memory Forwarding technique, on the other hand, adds 64% overhead to applications code [48].

The main goals of all allocators are high execution performance and high storage utilization, which are difficult to reach at the same time. Memory managers also practice relocation for better locality behavior. Memory Management methods have been purely
implemented in software and a very slight effort has been made to get support from hardware for better performance [66].

Our research is originally motivated by the Memory Wall problem, which has, of course, triggered all of the above research trends. We have started by studying the existing problems in memory management techniques. Providing more efficient memory management functions, we have noticed that these functions along with other service functions are the main cause of cache pollution and poor locality behavior of applications. This observation has led us to Processor In Memory Devices and their use to perform the data-intensive service functions for improving the locality of applications.

This dissertation addresses both the hardware and software issues of memory system. In this work, we propose new memory management algorithms which exhibit high storage utilization while maintaining moderate execution performance. We highlight the impact of internal fragmentation on locality behavior of applications and propose an Exact Fit allocator for better locality.

To tolerate memory latency, we propose a form of Processor In Memory Devices (Intelligent Memory Management) which replaces DRAM chips in conventional architecture. We use Intelligent Memory Management for executing Data-Intensive allocation and deallocation functions. The main advantage of Intelligent Memory Management is that it removes the cache pollution caused by memory management service functions. Empirical data in this dissertation show that offloading memory management functions from CPU and executing them by a processor in memory (Intelligent Memory Management) result in 60% cache miss reduction.

The rest of the dissertation is organized as follows:

- **Chapter 2** Introduces our new memory management methods called *Address Ordered and Segregated Binary Trees*. A memory manager (also known as allocator) should keep track of free chunks of memory. In our methods we maintain the free
chunks of memory in binary search trees. The search key in both Address Ordered and Segregated Binary trees is the starting address of the free chunks. When a de-allocation happens, the memory manager traverses the tree to insert the newly freed chunk and checks if the freed chunk can be coalesced with its adjacent nodes in the tree. We choose address ordered trees for better coalescing which improves storage utilization. When an allocation request arrives, the allocator looks for a free chunk in the tree that is at least as large as the request size. Within each node (free chunk of memory) of the tree, we also keep the size of largest nodes of its left and right subtrees. This information improves the allocator’s speed to find the suitable chunk for an allocation request. Address Ordered Binary Tree keeps the free chunks of memory in only one tree, whereas Segregated Binary Tree allocator keeps the free chunks in several trees based on the size of the chunks.

- **Chapter 3** explains memory fragmentation in detail. Memory fragmentation is the inability to use free space. Memory fragmentation consists of internal and external fragmentation. Internal fragmentation is the amount of extra memory allocated for an allocation request. External fragmentation is the actual free memory from which an allocator is unable to satisfy an allocation request. In chapter 3, we show that commonly used allocators perform about equally in terms of external fragmentation. The nuances among allocators in terms of memory fragmentation is due to their differences in internal fragmentation. In this chapter, we also show that internal fragmentation counter the locality behavior of applications. We propose *hybrid*, an exact fit allocator, which minimizes internal fragmentation to improve cache performance. The experimental data in this chapter depict that 25% of the cache misses, in average, are eliminated when using hybrid allocator.

- **Chapter 4** presents the architecture for the Intelligent Memory System. In this chap-
ter, we propose three possible architectures for Intelligent Memory System, eController, eDRAM, and eCPU. eController is an extension to the centralized controller used in memory systems. We place a processor in the heart of memory controller for performing data-intensive service functions. eDRAM is an embedded DRAM that replaces DRAM chips. eCPU adds a small processing engine for executing service functions on chip with the main CPU. Chapter 4 also discusses the software issues related to each of the proposed architectures.

- **Chapter 5** presents a novel idea, Intelligent Memory Management (IMM), that eliminates 60% of the cache pollution caused by the allocation and de-allocation service functions. IMM is the Memory Processor (in our Intelligent Memory System) that performs the allocation and de-allocation service functions. These service functions, when executed by the main CPU, entangle with applications’ working set and become the major cause for cache pollution. Executing the allocation and de-allocation service functions by a processor integrated in DRAM chip removes the cache pollution and leaves the CPU’s cache entirely to the application.

- **Chapter 6** describes existing hardware implementation of almost the fastest memory management technique, Buddy System. In this chapter we discuss the performance issues of Buddy System implemented in hardware. Furthermore, and since Buddy system allocator is known for its speed, in this chapter, we suggest that IMEM uses Buddy System memory allocation technique. Finally, we compare the execution performance of IMEM when using Buddy System allocator with conventional architecture.

- In **Chapter 7** we draw our conclusions and describe our future work.
CHAPTER 2
ABT AND SBT: NEW MEMORY MANAGEMENT TECHNIQUES

2.1 Introduction

The efficiency of memory management algorithms, particularly in object oriented environments, has gained the attention of researchers. A memory manager's task is to organize and track the free chunks of memory as well as memory currently being used by the running process. The primary goals of any efficient memory manager are high storage utilization and execution performance [76]. Current implementations, however, have failed to achieve both aims at the same time. For example, Sequential Fit algorithms show high storage utilization but poor execution performance [30, 60]. On the other hand, Segregated Free lists reveal higher memory fragmentations, yet their execution performances are among the best. Well-known placement policies such as Best Fit and First Fit have been explored with both Sequential Fit and Segregated Free lists for either speed or storage utilization benefit.

We have proposed new variations to Binary Tree allocators, Address Ordered and Segregated Binary Tree memory managers, that report reasonable execution performance while maintaining low fragmentation compared with existing allocators.

In this chapter, first, we describe most commonly used process memory managers\footnote{They are also known as allocation techniques.} and address their shortcomings, in both execution performance and storage utilization terms. In Section 4, we present our new implementations of user process memory manager that address the disadvantages of available allocation techniques. Last sections of this chapter represent empirical results and draw our conclusions.
2.2 Levels of Memory Management System

For fully comprehending and appreciating the memory management system, it is necessary to realize its role in a typical computer system. Figure 2.1 shows the two levels of computer system memory manager: Operating System and user Process Memory Managers.

Operating System (OS) Memory Manager allocates large chunks of memory, called OS pages, to process memory management systems. The size of OS pages, allocated to process memory managers, is fixed. For example, Linux uses 4 KByte pages whereas Alpha Unix page size is 8 KBytes. During the entire period of its execution, a user process acquires no more than 300 OS pages; therefore, OS memory manager’s task is uniform and routine. In contrast, runtime system memory managers (user level processes) are responsible for allocating small chunks of memory to running processes [15]. A typical process allocates more than tens of thousands of objects of different sizes dynamically [60]. The separation of memory management is needed to eliminate too frequent kernel calls when the memory space of a running process grows dynamically.

Usually runtime system\(^2\) uses different primitives to increase and decrease the address space of “heap” and “stack”; “heap” is the memory space that houses dynamically allocated

---

\(^2\)Note that memory manager of runtime system is in our interest.

---

![Figure 2.1: Memory Management Hierarchy, Reproduction by permission of the author and IRWIN publisher [15]]
objects, whereas “stack” is the memory space for keeping the local variables of functions when they are activated. Figure 2.2 depicts how the address space of a running process, especially “heap” and “stack”, grow. For example, “gcc”\(^3\) provides “obstack” (object stack) routines for resizing the stack space and allocation libraries for resizing the “heap”\(^4\). When a user program needs more space, it issues an allocation request. User process memory manager, in response, returns the address of the block of memory as large as the requested size. If the process memory manager cannot successfully respond to the request, it will acquire more memory from Operating System (OS) memory manager via kernel system calls. Unix like OSs provide two families of system calls for such purposes: “brk, sbrk” and “mmap, munmap”. “brk” returns so-called break point of the user process address space\(^5\). Via “sbrk”, user process memory manager is able to either acquire more memory from OS or release some of the unused portion of its available memory back to the OS. One of the main concerns with “sbrk” is that the caller is responsible for page alignment of the returned addresses. The functionality of the other family of OS memory manager system calls, “mmap” and “munmap”, is similar to “sbrk” and “brk” with the difference that the returned addresses are page aligned, and that the caller does not need to maintain the page alignment afterwards. They are also supported in other Operating Systems like Windows families.

2.3 Known Allocation Techniques

Currently used memory allocation schemes can be classified into **Sequential Fit** algorithms, **Buddy Systems**, **Segregated Free Lists**, and **Binary Tree** techniques.

Sequential Fit approach (including First Fit and Best Fit) keeps track of available chunks

---

\(^3\) gnu c compiler

\(^4\) Allocation libraries contain malloc, free, realloc, calloc, and valloc.

\(^5\) Start point of the heap is called break of user process address space.
of memory in a doubly linked list. Known Sequential Fit techniques differ in how they track the memory blocks, how they allocate memory requests from the free blocks, and how they place newly freed objects back into the free list. When a process releases memory, these chunks are added to the free list, either at front or in place, if the list is sorted by addresses (Address Order [76]). When an allocation request arrives, the free list is searched until an appropriate sized chunk is found. The memory is allocated either by granting the entire chunk or by splitting the chunk (if it is larger than the requested size). Best Fit methods try to find the smallest chunk that is at least as large as the request, whereas First Fit methods find the first chunk that is at least as large as the request [41]. Best Fit method may involve delays in allocation while First Fit method leads to more external fragmentation [30]. If the free list is in address order, newly freed chunks may be combined with their surrounding blocks. Such practice, referred to as coalescing, is made possible by employing boundary tags in doubly linked list of address ordered free chunks [41].

In Buddy Systems the size of any memory chunk (live, free, or garbage) is $2^k$ for some $k$ [40, 41]. Two chunks of the same size that are next to each other, in terms of their memory addresses, are called buddies. If a newly freed chunk finds its buddy among free chunks, two
buddies can be combined into a larger chunk of size $2^{k+1}$. During allocation, larger blocks are split into equal sized buddies until a small chunk that is at least as large as the request is created. Large internal fragmentation is the main disadvantage of this technique. It has been reported that as much as 25% of memory is wasted due to internal fragmentation in buddy systems [30]. An alternate implementation, Double Buddy, which creates buddies of equal size but does not require the sizes to be $2^k$, is shown to reduce the fragmentation by half [30, 77].

Segregated Free List approaches maintain multiple linked lists, one for each different sized chunks of available memory. Allocation and de-allocation requests are directed to their associated lists based upon the size of the requests. Segregated Free Lists are further classified into two categories: Simple Segregated Storage and Segregated Fits [76]. No coalescing or splitting is performed in Simple Segregated Storage and the size of chunks remains unaltered. If a request cannot be satisfied from its associated sized list, additional memory from operating system is acquired via `sbrk` or `mmap` system calls. In contrast, Segregated Fit allocator attempts to satisfy the request from a list containing larger sized chunks - a larger chunk is split into several smaller chunks if required. Coalescing is also employed in Segregated Fit allocators for further improvement of storage utilization. Simple Segregated Storage allocators are best known for their high execution performance while Segregated Fit allocators’ edge is their high storage utilization.

In Binary Tree allocators, free chunks of memory are kept in a binary search tree whose search key is the address of the free chunks of memory. Cartesian Tree, which was proposed almost two decades ago, is one of the known Binary Tree Allocators [71]. This allocator is an address ordered binary search tree that forces its tree of free chunks to form a heap in terms of chunk sizes. In other words, Cartesian Tree allocator maintain a binary tree whose nodes are the free chunks of memory with the following conditions:

a. address of descendants on left (if any) $\leq$ address of parent $\leq$ address of descendants
on right (if any)

b. size of descendants on left (if any) \( \leq \) size of parent \( \geq \) size of descendants on right (if any)

The latter that mandates Cartesian Tree to have its largest node at the root, causes the tree to usually become unbalanced and possibly degrade into a linearly linked list.

There exist variety of ad hoc allocators in literature that are not included in this work for several reasons. First and foremost, our study is directed towards general purpose allocators. Secondly, it is not our intention to concentrate on allocators, rather we would like to form a smarter allocator which possesses reasonable performance, high storage utilization, and good locality behavior. More thorough taxonomy of different allocators can be found in the Survey written by Wilson and et al [76].

### 2.4 New Allocation Techniques: Address Ordered and Segregated Binary Trees

In Address Ordered Binary Tree (ABT), the free chunks of memory are maintained in a binary search tree like in Cartesian Tree [34, 60]. To overcome the inefficiency forced by the size condition of Cartesian Tree allocator (condition b), we not only remove this restriction entirely from our implementation, but also replace it with a new strategy that enhances the allocation speed of ABT technique. Similar to Segregated Fit allocator, Segregated Binary Tree keeps several ABTs, one for each class size.

#### 2.4.1 Address Ordered Binary Tree (ABT)

In this specific implementation of Binary Tree algorithms, each node of the tree contains the sizes of the largest memory chunks available in its left and right subtrees. This information can be utilized to improve the response time of allocation requests and used for implemen-
tation of Better Fit policies to improve memory utilization [60, 76]. Binary Tree algorithms whose trees are address ordered are ideally suited for coalescing the free chunks; hence, storage utilization is further improved. In our Address Ordered Binary Tree, while inserting a newly freed chunk of memory, we check if it can be coalesced with existing nodes in the tree. Inserting a new free chunk will require searching the tree with $O(l)$ complexity where $l$ is the tree level bounded by $\log_2(n)$ and $n$; $n$ is the number of nodes in the tree. It is possible that the tree de-generates into a linear list, leading to a linear $O(n)$ insertion complexity. To minimize the insertion complexity, we advocate periodic tree re-balancing, which can be aided by keeping the information about the levels and number of nodes of the left and right subtrees. Note that coalescing of the chunks described above already helps in keeping the tree from being unbalanced. Thus, the number of times a tree should be re-balanced, although it depends on specific application, will be relatively infrequent in our approach.

Algorithm For Inserting a Newly Freed Memory Chunks

The following algorithm shows how a newly freed chunk can be added to Address Ordered Binary Tree of available chunks. The data structure of each node representing the free chunk of memory contains chunk’s size, pointers to its left and right children, a pointer to its parent, and the sizes of largest chunks in its right and left subtrees.

Structure used for each node of the tree:

```c
struct node{
    size_t Size;
    size_t MaxLeft;
    size_t MaxRight;
    struct node *Left;
    struct node *Right;
    struct node *Parent;
};
```

Insertion and Coalescing algorithms.

```c
void INSERT(void *ChunkAddress,size_t ChunkSize,node *Travel){
```
if(((void *)Travel+ Travel->size == ChunkAddress) ||
(ChunkAddress + ChunkSize == (void *)Travel))
    COALESCE(ChunkAddress,ChunkSize,Travel);
else{
    if(ChunkAddress < (void *)Travel){
        if(Travel->Left == NULL){
            Travel->Left=CREATE(ChunkAddress,ChunkSize);
            Travel->MaxLeft=ChunkSize;
            ADJUSTSIZE(Travel);
        }
        else
            INSERT(ChunkAddress,ChunSize,Travel->Left);
    }
    else{
        if(Travel->Right == NULL){
            Travel->Right=CREATE(ChunkAddress,Chunksize);
            Travel->MaxRight=ChunkSize;
            ADJUSTSIZE(Travel);
        }
        else
            INSERT(ChunkAddress,ChunkSize,Travel->Right);
    }
}
void ADJUSTSIZE(node *Travel){
    if(Travel->Parent == NULL)
        return NULL;
    if(Travel->Parent->Left == Travel)
        Travel->Parent->MaxLeft=
            MAX(Travel->Size,Travel->MaxLeft,Travel->MaxRight);
    else
        Travel->Parent->MaxRight=
            MAX(Travel->Size,Travel->MaxLeft,Travel->MaxRight);
    ADJUSTSIZE(Travel->Parent);
}
void COALESCE(void *ChunkAddress,size_t ChunkSize,node *Travel){
    if(ChunkAddress > (void *) Travel)
        Travel->size+=ChunkSize;
    else{
        ChunkAddress=(node *)Travel;
        ChunkAddress->size+=ChunkSize;
    }
    ADJUSTSIZE(Travel);
Complexity Analysis

INSERT is very similar to binary tree traversal and its time complexity depends on \( l \) (level of the tree). COALESCE’s complexity depends on ADJUSTSIZE function that traverses the tree upwards; therefore, both COALESCE and ADJUSTSIZE possess \( O(l) \) time complexity. The other functions used in the implementation of ABT are SEARCH and DELETE. SEARCH is a search binary tree that is improved with keeping MaxLeft and MaxRight; hence, its upper bound time complexity is \( O(l) \). DELETE function only visits one node but it calls ADJUSTSIZE and therefore its time complexity is also \( O(l) \).

2.4.2 Segregated Binary Tree (SBT)

In a manner similar to Segregated Fit technique, Segregated Binary Tree keeps several Address Ordered Binary Trees, one for each chunk size [61]. Each tree is typically small, thus reducing the search time while retaining the memory utilization advantage of Address Ordered Binary Tree. In our implementation, SBT contains 8 binary trees; Memory chunks less than 64 bytes and greater than 512 are kept in the first and the last binary trees respectively. Each binary tree is responsible for keeping chunks of a size range, and sizes range in 64 byte intervals. For example, the second binary tree’s range is \([64, 128)\) (viz., if a chunk’s size is \( x \) then \( 64 \leq x < 128 \)).

2.5 Empirical Results

In order to evaluate the benefits of our approach to memory management, we developed simulators that accept requests for memory allocation and de-allocation. We have studied 4 different implementations for tracking memory chunks: Address Ordered Binary Tree (ABT), Sequential Fit (SqF), Segregated Binary Tree (SBT), and Segregated Fit (SgF). We have investigated the impact of different placement policies - in Sequential and Segregated Fits we have employed First Fit and Best Fit; in Address Ordered and Segregated Binary First Fit
and Better Fit. To observe the impact of Segregation on Memory, we have compared SBT with Segregated Fit (SgF). We have conducted our experiments with and without coalescing to investigate its impact on different allocators, especially ABT and SBT. In this section, we will first explain our framework and then show the data collected while performing our experiments.

2.5.1 Experimental Framework

For our experiments, we have used Java Spec98 benchmarks since java programs are allocation intensive [29]. Applications with large amount of live data (dynamically allocated) are worthy benchmark candidates for memory management algorithms, because they expose the memory allocation speed and memory fragmentation clearly.

Java Spec98 benchmarks are instrumented using ATOM on HP/Digital Unix, to collect traces indicating memory allocation and de-allocation requests [21]. Table 2.1 summarizes the benchmarks’ descriptions while Table 2.2 reports the benchmarks’ statistics.

Table 2.2 shows that:

- On average, Java applications allocate more than 70,000 objects.
- About ten thousand objects are not de-allocated; they lead to the memory leaks. Note that Java run time system uses an automatic memory manager so that programmer is
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total No. Allocation Requests</th>
<th>Total No. Deallocation Requests</th>
<th>Total Requested (Mbytes)</th>
<th>Average Request (Bytes)</th>
<th>Average Live (Mbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>46,665</td>
<td>40,991</td>
<td>4.25</td>
<td>96</td>
<td>0.82</td>
</tr>
<tr>
<td>compress</td>
<td>41,239</td>
<td>35,849</td>
<td>3.71</td>
<td>94</td>
<td>0.77</td>
</tr>
<tr>
<td>db</td>
<td>43,176</td>
<td>37,727</td>
<td>3.86</td>
<td>94</td>
<td>0.79</td>
</tr>
<tr>
<td>jack</td>
<td>80,814</td>
<td>73,854</td>
<td>6.83</td>
<td>86</td>
<td>1.12</td>
</tr>
<tr>
<td>javac</td>
<td>132,745</td>
<td>123,222</td>
<td>11.01</td>
<td>87</td>
<td>1.68</td>
</tr>
<tr>
<td>jess</td>
<td>81,857</td>
<td>74,298</td>
<td>7.25</td>
<td>93</td>
<td>1.29</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>97,430</td>
<td>91,095</td>
<td>7.18</td>
<td>77</td>
<td>1.23</td>
</tr>
<tr>
<td>mtrt</td>
<td>54,682</td>
<td>48,662</td>
<td>4.73</td>
<td>91</td>
<td>0.92</td>
</tr>
<tr>
<td>average</td>
<td>71,148</td>
<td>64,603</td>
<td>5.98</td>
<td>89.2</td>
<td>1.05</td>
</tr>
</tbody>
</table>

Table 2.2: Benchmark Statistics

not responsible for de-allocating the objects. Automatic memory manager deals with three types of objects: live (application is currently using them), free (which are in the free list of the allocator), or garbage (application is not using them anymore, but they have not been freed yet). Every so often, when memory is exhausted, garbage collector, which is a part of the automatic memory manager, in some way, identifies the garbage objects. If any object is marked as garbage, the garbage collector will, then, issue a de-allocation request to move them to the free list.

- Total memory requested by applications is about 6 MBytes.
- The average amount of live memory is 1 MBytes - 128 pages if page size is 8 KBytes\(^6\). Normally about 50% of memory is wasted; hence, an allocator needs 260 pages of memory to perform reasonably in terms of fragmentation.
- Average request size is 90 Bytes, which means there are 12,000 live objects at any given time during the application’s run.

\(^6\)In our simulators we have also used 8 KByte pages.
For comparing different allocators, we collected statistical data for the following items:

**Average Number of Free Chunks** measures the memory overhead of a memory management algorithm. It is also an indication of how the memory space is fragmented; the more the number of the free chunks, the higher the memory fragmentation.

**OS Pages Consumed.** This item shows how many pages of memory are acquired by the allocator. Each page is 8 KBytes; hence, this figure multiplied by 8 equals the amount of memory an allocator consumed in KBytes. This number also reports how many times operating system kernel is called to increase the heap size. Kernel system calls are expensive and cost a lot of execution cycles; therefore, *OS Pages Consumed* is an indirect indication of execution performance.

**Internal Fragmentation** measures the excess memory allocated by an allocator as compared to the actual memory requested by the user program.

**Average Numbers of Nodes Searched at Allocation and De-allocation** measure the execution complexity for each algorithm while allocating or de-allocating a chunk of memory. These numbers provide a measure of the allocator’s execution efficiency.

**Maximum Numbers of Nodes Searched at Allocation and De-allocation** are the worst-case execution time for allocation and de-allocation.

**Coalescence Frequency** measures how often a newly freed chunk of memory can be combined with other free nodes. “Less fragmentation will result if an implementation immediately coalesces free chunks” [76].

### 2.5.2 Comparison of ABT and SqF

We have considered two cases for comparing these allocators: with and without coalescing. Coalescing illuminates the properties of allocators, and assists us in judging them justly.
Table 2.3 and Table 2.4 depict our experimental results for Address Ordered Binary Tree (ABT) and Sequential Fit (SqF) allocators with First Fit policy. In almost all the categories shown in the tables, ABT outperforms SqF. What follows summarizes our observations of data from these tables:

**Storage Utilization** *Average Number of Nodes* in SqF is about 3.5 times more than in ABT, which means SqF fragments memory 3.5 times more than ABT does. In addition, *Total Number of OS Pages* consumed by ABT is about 110 fewer than that is consumed by SqF; another figure that verifies fragmentation in its general sense. Therefore, ABT better utilizes storage than SqF.

*Internal Fragmentation* in ABT is higher than SqF, since the internal memory overhead in ABT is more than in SqF. Each free chunk in Address Ordered Binary Tree keeps three sizes\(^7\) and three addresses\(^8\) in its header. If the computer system is 32 bits\(^9\), each of these items will be 4 bytes; consequently, 24 bytes is needed for keeping each chunk in the tree (“3 addresses + 3 sizes” \(*4 = 24\) ). This means that any allocation request smaller than 24 bytes should be rounded up to 24. If the request sizes are small (which is the case for Java benchmarks), allocating bigger chunks for small sized requests causes higher internal fragmentation. For example, Sequential Fit allocators, need two addresses (previous and next pointers) and a size (chunk’s size), which add up to 12 Bytes overhead. That is why the internal fragmentation reported by SqF in Table 2.4 is less than what Table 2.3 shows for ABT.

**Execution Performance** Total number of *OS memory pages consumed* by ABT is 110

---

\(^7\)Chunk’s size, MaxLeft (i.e., size of the largest chunk in the left subtree), and MaxRight (i.e., size of the largest chunk in the right subtree)  
\(^8\)Parent (i.e., a pointer to the parent), Left (i.e., a pointer to the left child), Right (i.e., the right child pointer)  
\(^9\)Note that we collected traces on 64 bit machine (i.e., Alpha 21164), but run our simulators on 32 bit machine (i.e., Intel Pentium IV).
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average No. Nodes</th>
<th>OS Pages Consumed</th>
<th>Internal Frag. Kbytes</th>
<th>Nodes Searched At Allocation Avg.</th>
<th>Nodes Searched At Deallocation Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>2089</td>
<td>194</td>
<td>314</td>
<td>26</td>
<td>41</td>
</tr>
<tr>
<td>compress</td>
<td>3077</td>
<td>181</td>
<td>269</td>
<td>27</td>
<td>41</td>
</tr>
<tr>
<td>db</td>
<td>3175</td>
<td>184</td>
<td>274</td>
<td>28</td>
<td>47</td>
</tr>
<tr>
<td>jack</td>
<td>5347</td>
<td>263</td>
<td>478</td>
<td>39</td>
<td>75</td>
</tr>
<tr>
<td>javac</td>
<td>5116</td>
<td>392</td>
<td>874</td>
<td>83</td>
<td>100</td>
</tr>
<tr>
<td>jess</td>
<td>3316</td>
<td>295</td>
<td>578</td>
<td>33</td>
<td>38</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>13606</td>
<td>373</td>
<td>536</td>
<td>77</td>
<td>125</td>
</tr>
<tr>
<td>mtrt</td>
<td>3781</td>
<td>208</td>
<td>346</td>
<td>31</td>
<td>46</td>
</tr>
<tr>
<td>average</td>
<td>4938</td>
<td>261</td>
<td>459</td>
<td>43</td>
<td>65</td>
</tr>
</tbody>
</table>

Table 2.3: Address Ordered Binary Tree, First Fit

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average No. Nodes</th>
<th>OS Pages Consumed</th>
<th>Internal Frag. Kbytes</th>
<th>Nodes Searched At Allocation Avg.</th>
<th>Nodes Searched At Deallocation Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>10123</td>
<td>249</td>
<td>215</td>
<td>153</td>
<td>8644</td>
</tr>
<tr>
<td>compress</td>
<td>9271</td>
<td>237</td>
<td>188</td>
<td>159</td>
<td>8868</td>
</tr>
<tr>
<td>db</td>
<td>9852</td>
<td>244</td>
<td>196</td>
<td>151</td>
<td>6514</td>
</tr>
<tr>
<td>jack</td>
<td>19361</td>
<td>387</td>
<td>371</td>
<td>358</td>
<td>15773</td>
</tr>
<tr>
<td>javac</td>
<td>32198</td>
<td>656</td>
<td>607</td>
<td>85</td>
<td>41898</td>
</tr>
<tr>
<td>jess</td>
<td>19654</td>
<td>442</td>
<td>382</td>
<td>135</td>
<td>12641</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>26081</td>
<td>482</td>
<td>431</td>
<td>282</td>
<td>26404</td>
</tr>
<tr>
<td>mtrt</td>
<td>12581</td>
<td>289</td>
<td>252</td>
<td>121</td>
<td>13327</td>
</tr>
<tr>
<td>average</td>
<td>17390</td>
<td>373</td>
<td>330</td>
<td>181</td>
<td>16759</td>
</tr>
</tbody>
</table>

Table 2.4: Sequential Fit, First Fit
fewer than what SqF consumed, on average. For each extra page that an allocator needs, it issues an “sbrk” or “mmap” system call. Kernel system calls are very expensive; hence, the more system calls the worse the performance.

Nodes Searched at Allocation and De-allocation directly reflect the execution performance of allocators. SqF is a LIFO (Last In First Out) list; newly freed chunks are put at the front of the list, which counts for only one node de-allocation search. For a fair comparison, one needs to add the average number of nodes searched at allocation and de-allocation of ABT and compare it with the average number of nodes searched at allocation of SqF. Doing so, we have come to realize that ABT performs 1.7 times faster than SqF. Only in one application (i.e., Java compiler) SqF outperforms ABT, but it issues twice as many “sbrk” system calls, which counters this performance benefit.

Table 2.5 and 2.6 depict the results when using Better Fit placement policy for ABT and Best Fit for SqF. No significant improvement has been observed when using Better Fit ABT; in fact, First Fit performs better. In contrast, there is a lot of improvement in SqF performance with Best Fit. Average number of nodes dramatically changed as an indication of much better storage utilization. This is also supported by the fact that the total number of OS memory pages consumed, on average, is dropped down from 373 (SqF First Fit) to 254 (SqF Best Fit). Average number of nodes searched at allocation, however, is increased, which reflects that the execution performance of Best Fit is worse than First Fit in SqF allocators.

In terms of speed, Better Fit ABT outperforms Best Fit SqF. In terms of storage utilization, though, Best Fit SqF is best among all allocators. Best Fit placement policy, because of its high storage utilization, has also been used in Segregated Fit allocators. Speedup resulted from segregation will balance the poor execution performance of Best Fit, and high storage utilization benefit of it will counteract fragmentation caused by segregation if Segregated Fit allocators use Best Fit placement policy.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average No. Nodes</th>
<th>OS Pages Consumed</th>
<th>Internal Frag. Kbytes</th>
<th>Nodes Searched At Allocation Avg.</th>
<th>Max</th>
<th>Nodes Searched At Deallocation Avg.</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>1896</td>
<td>215</td>
<td>315</td>
<td>33</td>
<td>1331</td>
<td>46</td>
<td>1568</td>
</tr>
<tr>
<td>compress</td>
<td>2976</td>
<td>195</td>
<td>264</td>
<td>37</td>
<td>1127</td>
<td>53</td>
<td>1130</td>
</tr>
<tr>
<td>db</td>
<td>3106</td>
<td>198</td>
<td>271</td>
<td>35</td>
<td>1311</td>
<td>49</td>
<td>1310</td>
</tr>
<tr>
<td>jack</td>
<td>5880</td>
<td>287</td>
<td>479</td>
<td>43</td>
<td>1376</td>
<td>80</td>
<td>1266</td>
</tr>
<tr>
<td>javac</td>
<td>5138</td>
<td>416</td>
<td>805</td>
<td>62</td>
<td>689</td>
<td>76</td>
<td>671</td>
</tr>
<tr>
<td>jess</td>
<td>3284</td>
<td>316</td>
<td>555</td>
<td>43</td>
<td>1397</td>
<td>47</td>
<td>1396</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>13501</td>
<td>388</td>
<td>524</td>
<td>68</td>
<td>2561</td>
<td>106</td>
<td>2561</td>
</tr>
<tr>
<td>mtrt</td>
<td>3741</td>
<td>229</td>
<td>343</td>
<td>43</td>
<td>1270</td>
<td>62</td>
<td>1269</td>
</tr>
<tr>
<td>average</td>
<td>4940</td>
<td>281</td>
<td>445</td>
<td>46</td>
<td>1383</td>
<td>65</td>
<td>1396</td>
</tr>
</tbody>
</table>

Table 2.5: Address Ordered Binary Tree, Better Fit

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average No. Nodes</th>
<th>OS Pages Consumed</th>
<th>Internal Frag. Kbytes</th>
<th>Nodes Searched At Allocation Avg.</th>
<th>Max</th>
<th>Nodes Searched At Deallocation Avg.</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>1377</td>
<td>188</td>
<td>208</td>
<td>307</td>
<td>6650</td>
<td></td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>2533</td>
<td>175</td>
<td>182</td>
<td>451</td>
<td>6819</td>
<td></td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>2647</td>
<td>179</td>
<td>189</td>
<td>419</td>
<td>6794</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jack</td>
<td>4856</td>
<td>256</td>
<td>345</td>
<td>666</td>
<td>8849</td>
<td></td>
<td></td>
</tr>
<tr>
<td>javac</td>
<td>4466</td>
<td>380</td>
<td>559</td>
<td>535</td>
<td>6668</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jess</td>
<td>2647</td>
<td>288</td>
<td>365</td>
<td>379</td>
<td>6718</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpegaudio</td>
<td>13215</td>
<td>366</td>
<td>410</td>
<td>1644</td>
<td>25967</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtrt</td>
<td>3309</td>
<td>202</td>
<td>238</td>
<td>482</td>
<td>6798</td>
<td></td>
<td></td>
</tr>
<tr>
<td>average</td>
<td>4381</td>
<td>254</td>
<td>312</td>
<td>610</td>
<td>9408</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.6: Sequential Fit, Best Fit

24
On the whole, while retaining high storage utilization, First Fit ABT outperforms the other allocators we have studied thus far, in terms of speed, which is why we strongly believe that binary search trees are better candidates than linked lists for Segregated Allocators. Note that Segregated Allocators keep several linked lists, one for each size.

Coalescence Impact

Table 2.7 and Table 2.8 show the results of our experiments when we allowed coalescing in First Fit ABT and SqF allocators. Again, the goals of all allocators are high storage utilization and execution performance; likewise, we classify our remarks based on these two goals:

Storage Utilization  Bear in mind that we compare these two tables with their correspondents without coalescing (Table 2.3 and 2.4). *Average Number of Nodes*, *OS Pages Consumed*, and *Internal Fragmentation* are the table items which we draw our conclusions from to make our comments about Storage Utilization. Using Coalescing, when possible, we observe 72 to 78% reduction in *Average Number of Nodes*. This is particularly significant for ABT, since the need for re-balancing the tree is minimized. In other words, allowing Coalescing, as we expected, elicits a more balanced Address Ordered Binary Tree. First Fit ABT and SqF report 8 to 16% reduction in the number of *OS Pages Consumed*, Coalescing does not help allocators to achieve less fragmentation\(^\text{10}\). This conclusion is also supported by the fact that enhancement in Internal Fragmentation is negligible. Overall, the memory space is less fragmented, which is shown by the improvements in *Average Number of Nodes*\(^\text{11}\), using coalescing.

Execution Performance  Using coalescing, we notice that the *Average Number of Nodes Searched at Allocation and De-allocation* decreased by a factor of two to three; therefore, the major impact of coalescing is on the execution performance of allocators. The

\(^{10}\) Fragmentation, in its general concept, means inability to use free space.

\(^{11}\) Number of Nodes of allocator is the number of free chunks.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>918</td>
<td>189</td>
<td>296</td>
<td>0.43</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>compress</td>
<td>942</td>
<td>166</td>
<td>258</td>
<td>0.46</td>
<td>19</td>
<td>24</td>
</tr>
<tr>
<td>db</td>
<td>959</td>
<td>173</td>
<td>270</td>
<td>0.44</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>jack</td>
<td>1128</td>
<td>235</td>
<td>487</td>
<td>0.49</td>
<td>25</td>
<td>28</td>
</tr>
<tr>
<td>javac</td>
<td>1457</td>
<td>371</td>
<td>888</td>
<td>0.34</td>
<td>35</td>
<td>37</td>
</tr>
<tr>
<td>jess</td>
<td>1131</td>
<td>269</td>
<td>543</td>
<td>0.36</td>
<td>32</td>
<td>36</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>1273</td>
<td>335</td>
<td>518</td>
<td>0.69</td>
<td>48</td>
<td>65</td>
</tr>
<tr>
<td>mtrt</td>
<td>954</td>
<td>186</td>
<td>351</td>
<td>0.43</td>
<td>23</td>
<td>28</td>
</tr>
<tr>
<td>average</td>
<td>1095</td>
<td>241</td>
<td>451</td>
<td>0.46</td>
<td>28</td>
<td>34</td>
</tr>
</tbody>
</table>

Table 2.7: Address Ordered Binary Tree, First Fit with Coalescing

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>3194</td>
<td>220</td>
<td>205</td>
<td>0.6</td>
<td>76</td>
<td>3756</td>
</tr>
<tr>
<td>compress</td>
<td>2482</td>
<td>202</td>
<td>182</td>
<td>0.58</td>
<td>66</td>
<td>3153</td>
</tr>
<tr>
<td>db</td>
<td>2610</td>
<td>210</td>
<td>188</td>
<td>0.59</td>
<td>49</td>
<td>2578</td>
</tr>
<tr>
<td>jack</td>
<td>5419</td>
<td>345</td>
<td>353</td>
<td>0.61</td>
<td>114</td>
<td>4837</td>
</tr>
<tr>
<td>javac</td>
<td>9870</td>
<td>543</td>
<td>584</td>
<td>0.55</td>
<td>35</td>
<td>14514</td>
</tr>
<tr>
<td>jess</td>
<td>6759</td>
<td>387</td>
<td>365</td>
<td>0.56</td>
<td>62</td>
<td>11089</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>4805</td>
<td>358</td>
<td>418</td>
<td>0.67</td>
<td>104</td>
<td>4740</td>
</tr>
<tr>
<td>mtrt</td>
<td>3484</td>
<td>246</td>
<td>242</td>
<td>0.57</td>
<td>45</td>
<td>2965</td>
</tr>
<tr>
<td>average</td>
<td>4828</td>
<td>314</td>
<td>317</td>
<td>0.59</td>
<td>69</td>
<td>5954</td>
</tr>
</tbody>
</table>

Table 2.8: Sequential Fit, First Fit with Coalescing
execution overhead of Coalescing is negligible when compared with the search time. Each Coalescence, at most, needs two comparisons, three pointer modifications, and an addition - a total of 6 integer operations. If there is no chance of Coalescence, though, it will only need a comparison. As shown in the tables, the frequency of Coalescing is 46 to 59%; thus, at most 4 integer operations are added on each De-allocation\textsuperscript{12}. Comparing coalescing overhead with the reduction of \textit{Nodes Searched at Allocation}, (43 – 28 = 15)\textsuperscript{13} for ABT and (181 – 69 = 112)\textsuperscript{14} for SqF, which happens for each Allocation request, we certainly promote Coalescing. Moreover, since we propose the use of a separate processor for memory allocations and de-allocations, Coalescing should not impact CPU response time.

Most importantly, we observe a great improvement on \textit{Maximum Number of Nodes Searched at Allocation and De-allocation}. These numbers reflect the worst-case execution time, and they are significant for real time systems. Although in real time systems, dynamically allocating memory is not typically used, Coalescence First Fit ABT may be a good candidate if one is forced to allocate memory dynamically in these systems.

Finally, we have studied the influence of coalescing on Best Fit and Better Fit placement policies. Table 2.9 and 2.10 show the collected data for Better Fit ABT and Best Fit SqF with coalescing respectively. Comparing Better Fit ABT without and with coalescing, Table 2.5 and 2.9, we notice the consistency of coalescence impact concluded thus far among allocators.

One major significance is that Better Fit ABT with coalescing reveals the lowest \textit{Maximum Number of Nodes Searched at Allocation and De-allocation}, which makes it very suitable for real time systems.

The merit of Coalescence Best Fit SqF is its low \textit{Average Number of Nodes}. SqF allocator needs to keep two addresses (pointers to previous and next nodes in the list) and,

\begin{align*}
\text{\textsuperscript{12}} (0.59 \times 6) + ((1 - 0.59) \times 1) &= 3.95 \\
\text{\textsuperscript{13}} \text{Average Nodes Searched At Allocation (Avg.) in Table 2.3 and Table 2.7} \\
\text{\textsuperscript{14}} \text{Average Nodes Searched At Allocation (Avg.) in Table 2.4 and Table 2.8}
\end{align*}
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>713</td>
<td>220</td>
<td>293</td>
<td>0.47</td>
<td>23</td>
<td>27</td>
</tr>
<tr>
<td>compress</td>
<td>702</td>
<td>192</td>
<td>255</td>
<td>0.51</td>
<td>22</td>
<td>113</td>
</tr>
<tr>
<td>db</td>
<td>720</td>
<td>203</td>
<td>269</td>
<td>0.47</td>
<td>23</td>
<td>170</td>
</tr>
<tr>
<td>jack</td>
<td>1134</td>
<td>269</td>
<td>490</td>
<td>0.52</td>
<td>31</td>
<td>152</td>
</tr>
<tr>
<td>javac</td>
<td>1423</td>
<td>408</td>
<td>884</td>
<td>0.38</td>
<td>48</td>
<td>125</td>
</tr>
<tr>
<td>jess</td>
<td>1067</td>
<td>301</td>
<td>559</td>
<td>0.37</td>
<td>38</td>
<td>106</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>1484</td>
<td>360</td>
<td>519</td>
<td>0.69</td>
<td>38</td>
<td>273</td>
</tr>
<tr>
<td>mtrt</td>
<td>754</td>
<td>220</td>
<td>339</td>
<td>0.5</td>
<td>26</td>
<td>123</td>
</tr>
<tr>
<td>average</td>
<td>1000</td>
<td>272</td>
<td>451</td>
<td>0.49</td>
<td>31</td>
<td>145</td>
</tr>
</tbody>
</table>

Table 2.9: Address Ordered Binary Tree, Better Fit with Coalescing

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>455</td>
<td>186</td>
<td>204</td>
<td>0.29</td>
<td>173</td>
<td>1131</td>
</tr>
<tr>
<td>compress</td>
<td>501</td>
<td>159</td>
<td>179</td>
<td>0.34</td>
<td>198</td>
<td>1128</td>
</tr>
<tr>
<td>db</td>
<td>559</td>
<td>162</td>
<td>188</td>
<td>0.32</td>
<td>206</td>
<td>1186</td>
</tr>
<tr>
<td>jack</td>
<td>1218</td>
<td>244</td>
<td>345</td>
<td>0.31</td>
<td>360</td>
<td>2508</td>
</tr>
<tr>
<td>javac</td>
<td>1415</td>
<td>367</td>
<td>579</td>
<td>0.18</td>
<td>345</td>
<td>2599</td>
</tr>
<tr>
<td>jess</td>
<td>710</td>
<td>264</td>
<td>368</td>
<td>0.18</td>
<td>224</td>
<td>1549</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>1332</td>
<td>328</td>
<td>412</td>
<td>0.5</td>
<td>549</td>
<td>2383</td>
</tr>
<tr>
<td>mtrt</td>
<td>646</td>
<td>178</td>
<td>239</td>
<td>0.29</td>
<td>233</td>
<td>1302</td>
</tr>
<tr>
<td>average</td>
<td>855</td>
<td>236</td>
<td>314</td>
<td>0.3</td>
<td>286</td>
<td>1723</td>
</tr>
</tbody>
</table>

Table 2.10: Sequential Fit, Best Fit with Coalescing
of course, the node size information. If we use a 32 bit machine (e.g., Pentium IV), SqF will add 12-Byte header \( (3 \times 4 = 12) \) to each memory chunk, which is a node in the free list. If we separate the headers of memory chunks from the actual space used for keeping data, we should be able to cache the headers. If the number of the nodes in the free list is small so that headers of all nodes fit into the allocator’s cache, all allocator accesses to the nodes (actually headers of the memory chunks) will be cache hits. Number of Nodes in Coalescence Best Fit Sqf is 855, which means that with a small cache \( (855 \times 12 = 10 \text{ KBytes}) \), say 16 KBytes, the headers of all nodes fit into the cache, and consequently all the allocator’s accesses will be cache hits\(^{15}\).

### 2.5.3 The Effect of Segregation on Allocators

In order to observe the speed-up that segregated allocators are meant to reach, we have carried out our experiments on Segregated Binary Tree (SBT) and Segregated Fit (SgF) allocators. The Segregated Fit allocator used in these experiments is similar to our SBT in structure, but the memory chunks are kept in segregated doubly linked lists instead of trees. Table 2.11 and 2.12 show the collected data for First Fit SBT and SgF.

Comparing First Fit SBT with ABT (Table 2.3 and 2.11), we find that Number of Nodes Searched at Allocation, on average, is reduced, and Average Number of Nodes Searched at De-allocation is increased. Further, Maximum Number of Nodes Searched at Allocation and De-allocation is increased when segregation is used in our Binary Tree allocator. These numbers show that trees in SBT are not well balanced. Later on, we will see that Coalescing, which has been effective for ABT, causes more balanced SBTs. We have explored the use of a small Free Standing List with SBT which also results balanced SBT. The Free Standing List keeps recently freed objects. This list is searched first on an allocation request.

First Fit SgF, to compare with First Fit SqF, reports higher storage utilization as well\(^{15}\) if we separate the headers from nodes of free list, the header of each node will become 16 Bytes (i.e., because of an extra pointer from header to the memory chunk), however the size the thesis remains legitimate.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average No. Nodes</th>
<th>OS Pages Consumed</th>
<th>Internal Frag. Kbytes</th>
<th>Nodes Searched At Allocation Avg.</th>
<th>Max</th>
<th>Nodes Searched At Deallocation Avg.</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>1352</td>
<td>192</td>
<td>313</td>
<td>13</td>
<td>2340</td>
<td>40</td>
<td>2341</td>
</tr>
<tr>
<td>compress</td>
<td>2338</td>
<td>180</td>
<td>260</td>
<td>19</td>
<td>2281</td>
<td>62</td>
<td>2281</td>
</tr>
<tr>
<td>db</td>
<td>2436</td>
<td>184</td>
<td>270</td>
<td>14</td>
<td>2451</td>
<td>58</td>
<td>2442</td>
</tr>
<tr>
<td>jack</td>
<td>4542</td>
<td>264</td>
<td>469</td>
<td>17</td>
<td>2366</td>
<td>112</td>
<td>2363</td>
</tr>
<tr>
<td>javac</td>
<td>4676</td>
<td>402</td>
<td>747</td>
<td>107</td>
<td>2687</td>
<td>148</td>
<td>2684</td>
</tr>
<tr>
<td>jess</td>
<td>2647</td>
<td>295</td>
<td>550</td>
<td>20</td>
<td>2045</td>
<td>44</td>
<td>2054</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>12906</td>
<td>373</td>
<td>514</td>
<td>29</td>
<td>6693</td>
<td>134</td>
<td>6691</td>
</tr>
<tr>
<td>mtrt</td>
<td>3038</td>
<td>207</td>
<td>332</td>
<td>14</td>
<td>2327</td>
<td>50</td>
<td>2327</td>
</tr>
<tr>
<td>average</td>
<td>4242</td>
<td>262</td>
<td>432</td>
<td>29</td>
<td>2899</td>
<td>81</td>
<td>2898</td>
</tr>
</tbody>
</table>

Table 2.11: Segregated Binary Tree, First Fit

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average No. Nodes</th>
<th>OS Pages Consumed</th>
<th>Internal Frag. Kbytes</th>
<th>Nodes Searched At Allocation Avg.</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>3165</td>
<td>197</td>
<td>231</td>
<td>37</td>
<td>2556</td>
</tr>
<tr>
<td>compress</td>
<td>3895</td>
<td>183</td>
<td>199</td>
<td>49</td>
<td>2800</td>
</tr>
<tr>
<td>db</td>
<td>4099</td>
<td>186</td>
<td>207</td>
<td>49</td>
<td>2793</td>
</tr>
<tr>
<td>jack</td>
<td>7196</td>
<td>270</td>
<td>372</td>
<td>67</td>
<td>2921</td>
</tr>
<tr>
<td>javac</td>
<td>8981</td>
<td>418</td>
<td>655</td>
<td>67</td>
<td>4034</td>
</tr>
<tr>
<td>jess</td>
<td>6550</td>
<td>311</td>
<td>423</td>
<td>51</td>
<td>2996</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>15184</td>
<td>376</td>
<td>437</td>
<td>56</td>
<td>12579</td>
</tr>
<tr>
<td>mtrt</td>
<td>5107</td>
<td>214</td>
<td>259</td>
<td>56</td>
<td>2869</td>
</tr>
<tr>
<td>average</td>
<td>6772</td>
<td>269</td>
<td>348</td>
<td>54</td>
<td>4194</td>
</tr>
</tbody>
</table>

Table 2.12: Segregated Fit, First Fit
as better execution performance. SgF shows 60% improvements in the *Number of Nodes* and 28% reduction in the *OS Pages Consumed*; thus, it shows lesser fragmentation. It also reveals 3-fold decrease in search times as compared to SqF.

If you notice, SgF outperforms SBT in speed, since Segregated Binary Tree, without coalescing, generates unbalanced trees. In a multithreaded system, in which threads are running in parallel, one thread can be responsible for executing the memory management functions (memory management thread), while another can run the application code (application thread). Note that only the de-allocation functions of memory management thread can be run in parallel with application thread, viz., the application thread is only blocked when it issues an allocation request. In such a system, the allocation search time of an allocator is the dominant factor in execution performance; hence, SBT is a better choice than SgF for multithreaded systems.

Next set of tables, Table 2.13 and 2.14, compare Better Fit SBT with Best Fit SgF. These tables show that storage utilization reported by SgF is higher than SBT, while execution performance of SBT is higher than SgF. Still trees in SBT are unbalanced. Number of Nodes Searched at Allocation in SBT, on average, is 22, which is the fastest allocation time reported in this chapter thus far.

The impact of Free Standing List and Coalescing on SBT is shown in Table 2.15 and 2.16. The data shown in these two tables confirm that using Free Standing List or Coalescing makes the trees in SBT more balanced. The impact of coalescing is much more effective than Free Standing List; the Coalescence Frequency is very high, and the average number of nodes reported is low. Number of Nodes Searched at Allocation and De-allocation is small, and storage utilization is also high. On the whole, the goals of allocation techniques, i.e., high storage utilization and high execution performance, are reached when SBT implements coalescing.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average No. Nodes</th>
<th>OS Pages Consumed</th>
<th>Internal Frag. Kbytes</th>
<th>Nodes Searched At Allocation Avg.</th>
<th>Nodes Searched At Allocation Max</th>
<th>Nodes Searched At Deallocation Avg.</th>
<th>Nodes Searched At Deallocation Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>1308</td>
<td>195</td>
<td>312</td>
<td>9</td>
<td>2437</td>
<td>33</td>
<td>2437</td>
</tr>
<tr>
<td>compress</td>
<td>2312</td>
<td>181</td>
<td>257</td>
<td>13</td>
<td>2491</td>
<td>48</td>
<td>2555</td>
</tr>
<tr>
<td>db</td>
<td>2407</td>
<td>183</td>
<td>265</td>
<td>14</td>
<td>2361</td>
<td>48</td>
<td>2353</td>
</tr>
<tr>
<td>jack</td>
<td>4425</td>
<td>262</td>
<td>459</td>
<td>19</td>
<td>1962</td>
<td>142</td>
<td>2123</td>
</tr>
<tr>
<td>javac</td>
<td>4338</td>
<td>403</td>
<td>742</td>
<td>49</td>
<td>1462</td>
<td>87</td>
<td>1464</td>
</tr>
<tr>
<td>jess</td>
<td>2622</td>
<td>297</td>
<td>542</td>
<td>18</td>
<td>2324</td>
<td>39</td>
<td>2350</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>12822</td>
<td>373</td>
<td>511</td>
<td>42</td>
<td>9275</td>
<td>218</td>
<td>9621</td>
</tr>
<tr>
<td>mtrt</td>
<td>3011</td>
<td>207</td>
<td>330</td>
<td>10</td>
<td>623</td>
<td>52</td>
<td>2411</td>
</tr>
<tr>
<td>average</td>
<td>4156</td>
<td>263</td>
<td>427</td>
<td>22</td>
<td>2867</td>
<td>83</td>
<td>3164</td>
</tr>
</tbody>
</table>

**Table 2.13: Segregated Binary Tree, Better Fit**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average No. Nodes</th>
<th>OS Pages Consumed</th>
<th>Internal Frag. Kbytes</th>
<th>Nodes Searched At Allocation Avg.</th>
<th>Nodes Searched At Allocation Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>1887</td>
<td>190</td>
<td>220</td>
<td>141</td>
<td>3699</td>
</tr>
<tr>
<td>compress</td>
<td>2946</td>
<td>178</td>
<td>192</td>
<td>214</td>
<td>3716</td>
</tr>
<tr>
<td>db</td>
<td>3081</td>
<td>181</td>
<td>200</td>
<td>192</td>
<td>3707</td>
</tr>
<tr>
<td>jack</td>
<td>5591</td>
<td>261</td>
<td>360</td>
<td>222</td>
<td>5981</td>
</tr>
<tr>
<td>javac</td>
<td>5546</td>
<td>395</td>
<td>594</td>
<td>255</td>
<td>3816</td>
</tr>
<tr>
<td>jess</td>
<td>3580</td>
<td>292</td>
<td>393</td>
<td>182</td>
<td>3862</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>13832</td>
<td>370</td>
<td>427</td>
<td>550</td>
<td>15721</td>
</tr>
<tr>
<td>mtrt</td>
<td>3834</td>
<td>205</td>
<td>251</td>
<td>221</td>
<td>3728</td>
</tr>
<tr>
<td>average</td>
<td>5037</td>
<td>259</td>
<td>330</td>
<td>247</td>
<td>5529</td>
</tr>
</tbody>
</table>

**Table 2.14: Segregated Fit, Best Fit**
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>582</td>
<td>191</td>
<td>311</td>
<td>0.35</td>
<td>9</td>
<td>0.44</td>
<td>9</td>
<td>51</td>
<td>13</td>
<td>153</td>
<td>13</td>
<td>153</td>
</tr>
<tr>
<td>compress</td>
<td>608</td>
<td>168</td>
<td>250</td>
<td>0.44</td>
<td>8</td>
<td>0.44</td>
<td>8</td>
<td>51</td>
<td>12</td>
<td>153</td>
<td>12</td>
<td>153</td>
</tr>
<tr>
<td>db</td>
<td>626</td>
<td>171</td>
<td>261</td>
<td>0.42</td>
<td>9</td>
<td>0.49</td>
<td>12</td>
<td>138</td>
<td>19</td>
<td>153</td>
<td>19</td>
<td>153</td>
</tr>
<tr>
<td>jack</td>
<td>1056</td>
<td>245</td>
<td>460</td>
<td>0.49</td>
<td>12</td>
<td>0.62</td>
<td>17</td>
<td>222</td>
<td>31</td>
<td>313</td>
<td>31</td>
<td>313</td>
</tr>
<tr>
<td>javac</td>
<td>975</td>
<td>281</td>
<td>542</td>
<td>0.34</td>
<td>13</td>
<td>0.44</td>
<td>10</td>
<td>105</td>
<td>15</td>
<td>153</td>
<td>15</td>
<td>153</td>
</tr>
<tr>
<td>jess</td>
<td>1315</td>
<td>351</td>
<td>505</td>
<td>0.62</td>
<td>17</td>
<td>0.44</td>
<td>10</td>
<td>105</td>
<td>15</td>
<td>153</td>
<td>15</td>
<td>153</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>1671</td>
<td>391</td>
<td>783</td>
<td>0.4</td>
<td>17</td>
<td>0.44</td>
<td>10</td>
<td>105</td>
<td>15</td>
<td>153</td>
<td>15</td>
<td>153</td>
</tr>
<tr>
<td>mtrt</td>
<td>1894</td>
<td>421</td>
<td>737</td>
<td>0.44</td>
<td>12</td>
<td>0.44</td>
<td>12</td>
<td>110</td>
<td>17</td>
<td>173</td>
<td>17</td>
<td>173</td>
</tr>
<tr>
<td>average</td>
<td>950</td>
<td>250</td>
<td>430</td>
<td>0.44</td>
<td>12</td>
<td>0.44</td>
<td>12</td>
<td>110</td>
<td>17</td>
<td>173</td>
<td>17</td>
<td>173</td>
</tr>
</tbody>
</table>

Table 2.15: Segregated Binary Tree, Better Fit with Free Standing List

Table 2.16: Segregated Binary Tree, Better Fit with Coalescing
2.6 Conclusions

We have proposed new memory management algorithms, ABT and SBT, that maintain the available chunks of memory in binary search trees. The search key in ABT and SBT is the starting address of the free chunks of memory. In addition, we keep track of the sizes of largest chunk of memory in the left and right subtrees. This information is used to speed up the search phase of allocation.

We have used Java applications to compare ABT and SBT with the existing allocators, Sequential and Segregated Fit algorithms, since Java applications allocate several tens of thousands of objects of varied sizes. From Java Spec98 Benchmarks, we have collected the allocation and de-allocation traces and fed them to the memory management simulators. We have designed memory management simulators that report data on memory fragmentation and search time of allocation and de-allocation requests. Our simulation data show that:

In general, allocators perform the best when they are allowed to explore coalescing. ABT and SBT are address ordered; hence, coalescing, specifically, helps these allocators outperform others in allocation time. In today’s multithreaded architecture, one thread can be scheduled to execute the application’s code (application thread) while the other runs allocator’s code (memory management thread). The application thread can run fully in parallel with the memory management thread when processing the de-allocation requests; therefore, what matters the most is allocation search time the execution of the application thread is blocked. Since the fastest allocation search time among all allocators studied in this chapter has been achieved by SBT with coalescing, it is the best candidate for the memory management thread.

*Maximum Number of Nodes Searched at Allocation and De-allocation* matters the most for the real time systems, requiring bounded execution times. As shown in the tables, SBT with coalescing reports the lowest worst case search time (Max number of Nodes
Searched at Allocation is 110 for Better Fit SBT with coalescing; this makes SBT a good choice for real time system Memory Manager.

Best Fit Sequential Fit with Coalescing behaves the best among allocators in terms of Storage Utilization. It shows about 14% improvement in terms of fragmentation when compared with Better Fit SBT with Coalescing\(^\text{16}\). However, the execution performance improvement of Better Fit SBT compared with Best Fit SqF is 90% (17 + 12 = 29 compared with 286).

On the whole, the data represented in this chapter show that Address Ordered and Segregated Binary Trees’ execution performance is far better than Sequential and Segregated Fits’, while in terms of Storage Utilization all the allocators perform about the same.

\(^{16}\)We have averaged Number of Nodes, OS Pages Consumed, and Internal Fragmentation of both Best Fit SqF and Better Fit SBT to conclude this improvement.
CHAPTER 3
AN EXACT FIT ALLOCATOR: MINIMIZING INTERNAL FRAGMENTATION FOR BETTER CACHE PERFORMANCE

3.1 Introduction

Two main objectives of all allocators are speed and storage utilization [76]. It is very difficult to design an allocator that meet both goals for all applications. In this chapter, we focus on storage utilization, and we show that there is a close relationship between speed and storage utilization. In fact, we have already seen, in the previous chapter, that allocator with poor storage utilization issue more system calls (\textit{sbrk}, \textit{mmap}). Kernel system calls are expensive, and thus issuing extra system calls degrades overall performance. But in this chapter, we elicit a different aspect of allocators with poor storage utilization that incur poor execution performance.

Memory fragmentation is an indication of low storage utilization. Memory fragmentation is allocator’s inability to use free space. Johnson and Wilson claim that memory fragmentation problem is solved [30]. We also believe that memory fragmentation, in general, is not a major issue in terms of storage requirements. Internal fragmentation, however, is one of the reasons of cache misses in allocation-intensive applications; therefore, internal fragmentation indirectly impacts execution performance.

Our work indicates that widely used allocators behave the same in terms of overall fragmentation. But when we distinguish between internal and external fragmentation, their differences become clear. Allocators are different in terms of internal fragmentation because of two factors; one is the memory overhead that an allocator adds to each object it allocates, and the other is the excess memory added (i.e., over-allocation) to the actual request’s size due to allocator’s policy for not keeping very small free chunks. Whatever the cause, internal
fragmentation plays a role in harming and eluding the locality behavior of allocation-intensive applications.

We propose a solution which minimizes internal fragmentation, and, consequently, improves the locality of allocation-intensive applications. Our empirical results show that, on average, 25% of cache misses are eliminated when using our method.

The rest of this chapter is organized as follows: Section 2 explains the benchmarks and allocators used in this chapter. We will describe possible allocation patterns of applications in Section 3. Section 4 provides definition of fragmentation types. Section 5 describes the potential impacts of internal fragmentation on locality behavior of applications. We present our solution, i.e., an exact fit Allocator, that reduces internal fragmentation for better locality in Section 6. Experimental results that support the thesis of our approach are shown in Section 7. At the end, we draw our conclusions in Section 8.

3.2 Summary of Benchmarks and Allocators

Table 3.1 briefly explains the Benchmarks that we have used in this chapter. Three of these benchmarks belong to SPEC2000int [25], and three are chosen from a set of widely used benchmarks for memory management evaluations.

In this work six general purpose allocators are studied for their behaviors based on different allocation strategies. One of these allocators, "hybrid: An Exact Fit Allocator", 

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>gnu zip data compressor</td>
<td>test/input.compressed 2</td>
</tr>
<tr>
<td>parser</td>
<td>english parser</td>
<td>test.in</td>
</tr>
<tr>
<td>vpr</td>
<td>FPGA Circuit Placement and Routing</td>
<td>test/lindain.raw</td>
</tr>
<tr>
<td>boxed-sim</td>
<td>balls and box simulator</td>
<td>-n 10 -s 1</td>
</tr>
<tr>
<td>ptc</td>
<td>pascal to C convertor</td>
<td>mf.p</td>
</tr>
<tr>
<td>espresso</td>
<td>PLA optimizer</td>
<td>largest.espresso</td>
</tr>
</tbody>
</table>

Table 3.1: Benchmark Description
will be explained later on in Section 6, and the rest are described here.

**Address Ordered Binary Tree** ABT keeps the free chunks of memory in a binary search tree [60]. The search key of ABT is the starting address of free chunks. Sizes of the largest chunks in the left and right subtrees of each node are also kept with the node for further speed improvement (see Chapter 2 for more details). The placement policy employed by ABT used in this chapter is so-called heuristic Best Fit or Better Fit. “abt” in the figures is a reference to Address Ordered Binary Tree.

**BSD allocator** ¹ This allocator is an example of a Simple Segregated Storage technique [79]. It is among the fastest allocators but it reports high memory fragmentation. In the figures this allocator is referred to as “bsd”.

**Doug Lea’s allocator** Perhaps the most widely used allocator is Doug Lea’s. We have used version 2.7.0, an efficient allocator that has benefited from a decade of optimizations [45]. For request sizes greater than 512 bytes it uses a LIFO Best Fit. For requests less than 64 bytes it uses pools of recycled chunks. For sizes in between 64 and 512 bytes it explores a self adjusting strategy to meet the two main objectives of any allocator: speed and high storage utilization. For very large size requests (greater than 128 Kbytes), it directly issues `mmap` system call. “lea” is used to reference data for this allocator in our figures.

**Segregated Binary Tree** SBT contains 8 binary trees each for a different class size [61]. Memory chunks less than 64 bytes and greater than 512 bytes are kept in the first and the last binary tree respectively. Each binary tree is responsible for keeping chunks of a given size range, and sizes range in 64 byte intervals. For example, the second binary tree’s range is [64,128) (viz., if a chunk’s size is x then $64 \leq x < 128$). In the figures, “sbt” is used to refer to the data set belonging to this allocator.

¹known as Chris Kingsley’s allcoator - 4.2 BSD Unix
**Segregated Fit** We have written our own version of Segregated Fit algorithm referred to as “sgf”. In the structure, this allocator is similar to our SBT but the memory chunks are kept in segregated doubly linked lists instead of trees. LIFO Best Fit is chosen for placement policy of each list.

3.3 Different Allocation Patterns of Applications

According to the behavior of a variety of applications, their allocation gesture can be classified as *Ramp*, *Peak*, *Plateau*, or a combination [76].

Programs that accumulate data monotonically over time have a *Ramp* allocation pattern. This happens with applications that perform no de-allocation. Some programmers are reluctant to de-allocate objects after using them. Moreover, the programmer may need to build a large set of data structure gradually. “pte” allocation behavior is an example of *Ramp*, which is shown in Figure 3.1. Allocation time, on X-axis, is the accumulated amount of allocated memory (in byte) at each allocation. Y-axis, also in byte, is the amount of memory requested by the allocator. For example, if a sequence of requests for an application is:

<table>
<thead>
<tr>
<th>request</th>
<th>object name</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate</td>
<td>object 1</td>
<td>20 Bytes</td>
</tr>
<tr>
<td>allocate</td>
<td>object 2</td>
<td>40 Bytes</td>
</tr>
<tr>
<td>de-allocate</td>
<td>object 1</td>
<td></td>
</tr>
<tr>
<td>allocate</td>
<td>object 3</td>
<td>10 Bytes</td>
</tr>
<tr>
<td>de-allocate</td>
<td>object 2</td>
<td></td>
</tr>
</tbody>
</table>

The instances on its allocation behavior graph will be:

point 1  \( X : 20 \ & \ Y : 20 \)
point 2  \( X : 60 = 20 + 40 \ & \ Y : 60 = 20 + 40 \)
point 3  \( X : 70 = 60 + 10 \ & \ Y : 50 = 20 + 40 - 20 + 10 \)
Some applications allocate large data structures, use them for very short periods of time, and de-allocate them. This pattern, known as Peak, is a challenge for allocators. Memory will be rapidly fragmented if the allocator does not group the data structures that are allocated and freed together. “espresso”, shown in Figure 3.2, reports sharp Peaks in its allocation pattern. Another example of Peak is “gcc” (gnu C Compiler). “gcc” uses obstacks (object stacks) for procedure calls. Object stacks are the objects which are allocated incrementally and freed together; they are also called arena allocations.

Many applications tend to allocate large data structures together and use them for a long period of time. “parser”, shown in Figure 3.3, and “perl” interpreter, when running a script, reveal such pattern that is called Plateau.

Knowing the allocation pattern of an application, we can provide a special allocator that serves best for that pattern. This is becoming very common nowadays. “Perl” package, for example, provides its own allocator; “gcc” also uses its own allocator. This method can be adopted for PLA optimizers, like “espresso”, if we know the allocation pattern in advance.

![Figure 3.1: The Allocation Behavior of “ptc”, Ramp Behavior](image-url)
Figure 3.2: The Allocation Behavior of “espresso”, Peak Behavior

Figure 3.3: The Allocation Behavior of “parser”, Plateau Behavior
3.4 Memory Fragmentations

Fragmentation, generally, means inability to use available resources. Specifically, memory fragmentation is the inability to use free memory. If an allocator is unable to satisfy an allocation request, it will acquire more memory from Operating System (OS). The means for requesting memory from OS is a system call (either `sbrk` or `mmap`). System calls are very expensive; therefore, fragmentation indirectly impacts the performance, i.e., an allocator that suffers from high fragmentation reports degraded performance because of issuing more OS memory manager system calls.

It is claimed that well-known allocators behave the same in terms of fragmentation [30]. There exists a variety of measures of fragmentation. Johnson and Wilson, alone, present four measures of fragmentation shown in Figure 3.4 [30]. This figure shows the total memory requested from OS by an allocator, and the memory used by the application at each instant of time for “vpr” with “abt” allocator.

The first and perhaps mostly used definition of fragmentation is the average of frag-
mentations at all points in time. Fragmentation, if considered problematic, it is an issue for
abrupt changes in the allocation behavior of application. Averaging the fragmentation over
time will hide the response of an allocator to peaks or critical moments.

Another measure of fragmentation addressed by Johnson and Wilson, is the total amount
memory an allocator uses relative to the amount of live memory, at the time when the amount
of live memory is highest (point 1 relative to point 2 in Figure 3.4 - which is about 24%
fragmentation).

The worst case scenario happens when the amount of memory used by the allocator is
maximum (point 3), using this amount, fragmentation can be measured in two ways:

- The maximum amount of memory used by the allocator (point 3) relative to the amount
  of live memory at the same allocation time (point 4) - which is about 220% fragmenta-
  tion

- The maximum amount of memory used by the allocator (point 3) relative to the amount
  of live memory when it is maximum (point 2) - which is about 44% fragmentation

Johnson and Wilson collected statistics for both of these measures, for variety of allocators
using allocation-intensive applications. The statistical data assert that the memory frag-
mentation is not really a problem. This chapter rephrases their claim and illustrates that,
while the external fragmentation problem is not a serious concern in terms of storage needed,
internal fragmentation is of a great concern, in terms of execution performance.

Figure 3.5 and 3.6 show the percentage of fragmentation during application’s execution,
for two applications (boxed-sim and vpr) with three different allocators (abt, bsd, and sgf).
These Figures indicate that allocators exhibit similar patterns. They all tend to reach 50%
fragmentation before the end of execution. Thus, most researchers feel that memory frag-
mentation is not serious problem. In this chapter, we study the elements of fragmentation,
internal and external fragmentations, and show the differences that allocators reveal due to
the nuances in fragmentation among them.

Memory fragmentation is classified into two categories: internal and external fragmentations. Internal fragmentation is the extra bytes (i.e., over-allocation) allocated when a request is responded by the allocator. For example, if a 64 byte chunk is carved from free memory for a request of size 50, 14 bytes is wasted and that is the amount of internal fragmentation. On the other hand, external fragmentation is the actual existing memory in the free list from which a request cannot be satisfied (because no single chunk is large enough). Internal fragmentation is strongly associated with allocator’s policy and memory overhead to maintain data structures. In their implementations, allocators enforce allocation of extra memory for the sake of faster allocation. Also the size of the smallest object allocated depends on allocator’s memory overhead. For instance, Sequential Fit allocators, which maintain the free chunks of memory in doubly liked lists, need to reserve space for at least two pointers (pointers to previous and next free chunks in the list). In addition, all allocators need to keep the size information of a chunk in its header. Keeping this information, Sequential Fit allocators cannot allocate objects of size smaller than 12 bytes for 32-bit machines (and 24 bytes for 64-bit machines). For any object smaller than 12 bytes, therefore, there is an over-allocation or internal fragmentation in Sequential Fit allocators.

In most applications, however, external fragmentation dominates internal fragmentation by a large factor. In fact, external fragmentation behavior of allocators is the reason for the similarities in storage utilization, discussed above. The small differences observed in the overall amounts of fragmentation, shown in Figure 3.5 and 3.6, are primarily due to internal fragmentation caused by the allocators. These differences will be overlooked if we aggregate external and internal fragmentations. In the next section, we will show how these nuances in fragmentation (basically differences in internal fragmentation) can severely harm the locality properties of applications.
Figure 3.5: Percentage of Fragmentation for boxed-sim with different allocators

Figure 3.6: Percentage of Fragmentation for vpr with different allocators
3.5 Internal Fragmentation and its Impact on Cache Performance

To fully understand the impact of internal fragmentation on locality behavior of applications, we should appreciate the principle of locality and cache organization. The principle of locality conveys that the data which are accessed recently by the CPU are very likely to be accessed again in the near future [24]. This principle is the conclusion of an observation that the access pattern of applications is not uniform. Locality is further classified into two categories. Spatial Locality exists if the neighbors of the data that are accessed now will be accessed in the near Future. Temporal Locality suggests that the data that is accessed now is likely to be accessed again in the near future. Together with the conclusion that underlie the RISC philosophy, “making the common case fast”, principle of locality suggests the design of memory hierarchy, shown in Figure 3.7.

The size of memory at any level in the memory hierarchy is larger (but its access time is slower) than the size of memory at a lower level - farther from CPU. Most memories are inclusive in that data at a lower level also exist at higher level of memory. The size restriction and inclusive property of the memory hierarchy require a mapping mechanism

Figure 3.7: Levels of Memory Hierarchy in Conventional Architecture
between any two adjacent levels of the memory hierarchy. The mapping mechanism between memory and cache depends on the design of cache and could be either direct mapped or set associative [24, 67]. In direct mapped cache, several blocks of the main memory are mapped to one line of the cache. Note that memory block and cache line have the same size, and that the number of memory blocks which map to a cache line depends on the size of the cache and memory. For example, if a cache has 8 lines (each 32 bytes) and the main memory has only 32 blocks, every four blocks of memory (32 divided by 8) compete for a cache line. In fact, memory block 0, 8, 16, and 24 map to cache line 0. In set associative caches, several blocks of memory map into more than one line of cache. For example, two-way set associative cache organization maps a set of memory blocks to two lines of cache based on a replacement policy. Cache replacement policy can be Least Recently Used, First In First Out, or Random.

There are two memory operations that CPU performs. It either loads a data item from memory to a register (known as read or load), or stores a data item from a register to memory (known as store or write); in general, these two operations are called memory references. On each memory reference, the data item is either in the cache (in which case the data item is also in the memory) or, only, in the memory. If the datum is not in the cache, resulting in a cache miss, it will be brought into the cache from the memory before it
is loaded to register for processing. Cache memories are divided into cache lines (or blocks) that are a multiple of memory bus width. On any cache miss, the whole block of data will be brought into the cache from the memory. Cache misses are classified into three categories:

**Compulsory Misses** also known as Cold Misses. This happens when the cache is empty. This normally happens at the beginning of the program execution, or after a context switch if the Operating System is multitasking.

**Capacity Misses.** If the cache cannot contain all the data that a program needs during its execution, some data may be evicted from the cache (cache miss) and retrieved later on. These type of misses that are due to the limited size of the cache are called Capacity Misses.

**Conflict Misses.** A Conflict Miss happens when two data items that reside in two separate blocks of memory, and fall to the same cache line, are accessed after each other. When one of these data has been already brought into the cache, accessing the second one will cause a conflict cache miss.

Figure 3.8 visualizes internal fragmentation and how it forms the holes in objects. It is obvious from the figure that internal fragmentation lessens spatial locality. If each block is 64 bytes, accessing object 1, 2, and 3 shown in Figure 3.8 causes two compulsory misses. In applications, usually each object consists of several data items. For example, o11, o12, and o13, shown in the figure, belong to object 1. When The CPU accesses o11, as mentioned before, the entire memory block will be fetched to the cache by the memory system. Therefore, consequent accesses to o12, o13, or even data of object 2 will be cache hits. If the CPU accesses any datum of object 3, however, for example o31, another cache miss will occur.

If there is no fragmentation, shown in Figure 3.9, accessing the objects for the first time will only cost one cache miss. By reducing internal fragmentation, an allocator allows applications to enjoy greater spatial locality. In fact, if applications use adequate allocators
which have information about the access pattern of the applications, their temporal locality can be improved by the allocator by converting temporal locality to spatial locality.

If the memory space of an application is reduced, some of the Capacity Misses will also be eliminated. Last two Figures show that removing internal fragmentation reduces the memory space of applications. An application that allocates 1000 objects of size 40 Bytes (average size), for instance, will occupy 64 KBytes if its allocator over-allocates 24 Bytes extra for each object. The same application with a better allocator that only allocates 10 extra Bytes for each object will occupy 50 KBytes of memory.

Luk and Mowry have been investigating the possibility of memory relocations for eliminating the Conflict Misses [48]. In their work, they analyze the memory space of applications to find the data that cause Conflict Misses. They relocate data items that cause Conflict Misses to eliminate conflicts. Reducing the internal fragmentation shrinks the memory space of application; hence, the analysis phase of memory relocation will be less complex and faster.

In conclusion, allocators are responsible for constructing spatial locality for objects that are accessed together. If an allocator is able to recognize objects’ access patterns, it can allocate them from contiguous address spaces. We believe that internal fragmentation will lessen the spatial locality. In other words, even if the allocator is smart enough to allocate these objects from contiguous address space, internal fragmentation may still defeat spatial
locality. For an application to achieve good cache performance, its allocator should not only promote spatial locality but also reduce its internal fragmentation.

3.6 An Exact Fit Allocator

This allocator allocates the memory objects based on the actual size of the requests. It only adds 8 bytes (on 64-bit machine) for holding the size information of each object. The allocator consists of Simple Segregated Storage lists for chunks less than 512 bytes, and Segregated Binary Trees for chunks greater than 512 bytes (note that the exact fit allocation can also be implemented with other allocation techniques described in chapter 2). Each list or binary tree is associated with a size. Lists are 8 bytes apart, and trees are 64 bytes apart. Therefore, there are 63 lists with the smallest size being 8 bytes, and 9 binary trees with the largest size being 1024. Chunks greater than 1024 bytes are kept under a separate binary tree. No coalescing is performed in the lists, and hence all the chunks in the list are of the same size. Exact Fit allocator always keeps a pool of free memory. In the first attempt for allocating a new object, it looks for an available chunk in the segregated lists or binary trees. If it fails to find the appropriate chunk from the segregated bins, it will carve chunks from the pool of memory. If that is not successful, it will call “sbrk” or “mmap” system calls for acquiring more memory from OS.

Note that, Simple Segregated Storage lists are linear linked lists, and only one pointer is required to keep each chunk in any list. This means that the smallest chunk in the lists will be 8 bytes, which is the main reason that the Simple Segregated Storage lists start from 8 bytes. The size information is kept in the header of each chunk no matter whether the chunk is free or in-use by the application; therefore, we do not count this mandatory memory overhead as a part of the chunk’s size.

The closest work to our Exact Fit allocator in the literature is Tadman’s Fast Fit allocators.

\[\text{From now on, we will simply call them segregated bins.}\]
tor [70, 72]. Tadman’s allocator uses an array of free lists for small sizes, and an AVL binary tree for larger sizes for speed. Our Exact Fit Allocator also takes advantage of some of the features of Lazy Fit allocator [14].

3.7 Empirical Results

We have compared the cache performance of three applications when they run with different allocators. These applications are “boxed-sim”, “vpr”, and “gzip”; and the allocators are FreeBSD allocator (bsd), Doug Lea’s version 2.7.0 allocator (lea) [45], Segregated Binary Tree (sbt), Segregated Fit allocator (sgf), and the new Exact Fit allocator that is referred to as “hybrid” in the figures.

We have conducted our experiments on Alpha 21264 machine running Tru64 operating system [38], and instrumented the applications using ATOM [21]. ATOM tool consists of Instrumentation and Analysis routines. Instrumentation routines detect the required items of executable file and place probes on these items based on the desire of the user. Then, it will call Analysis routines to perform necessary analysis. We have used Instrumentation routines to detect the load and store instructions of running application, and Analysis routines to simulate different caches to collect the cache information.

Figure 3.10 depicts the number of cache misses for the applications with different allocators. The cache size here is 16 Kbytes with 32 Byte block size. This figure shows that, on average, “hybrid” reports 50 million misses fewer than the best allocator, which is “sbt”, and about 250 million misses fewer than the worst allocator among the rest, which is “bsd”. This is, indeed, about 25% miss reduction, on average.

Figure 3.11 shows the cache misses for 64 byte cache block size. Note that, the cache size is still 16 Kbytes. As the cache block size is increased, the applications reveal more cache misses. This is because allocation-intensive applications do not possess high spatial locality. But our hybrid allocator still performs very well.
Figure 3.12 shows that as cache size is doubled the number of misses dropped by about 100 millions (28% improvement), and “hybrid”, the Exact Fit allocator, still exhibits better locality behavior. Of course, when the cache size is increased, we expect fewer misses, for all allocators and applications.

Figure 3.13 depicts the miss rates. About 3 to 4 percent cache miss rates are reported for these applications. For only one application, “boxed-sim”, the hybrid allocator shows a higher miss rate than the others. This is because the total number of references in “boxed-sim” with other allocators is much higher than with hybrid allocator. This also shows that hybrid allocator is also efficient since it generates fewer instructions.

The Exact Fit allocator differs from other allocators in only one aspect, it tries to minimize the internal fragmentation by allocating chunks with almost no over-allocation. Note that all the allocators used in this work share the same implementation characteristics such as segregation, placement policy, and the absence of coalescing. Hence, the differences in cache behaviors of allocators, shown in this chapter, are primarily due to their internal fragmentation.

3.8 Conclusions

Fragmentation, if considered in its general form, the inability to use free space, can be declared solved. Well known general purpose allocators exhibit similar storage utilization behavior. However, the impact of fragmentation on locality behavior of applications will be overlooked, simply because it is viewed that all allocators perform almost the same with respect to fragmentation. If we distinguish between internal and external fragmentation, however, the effect of fragmentation on cache performance of applications will become clear.

In this chapter, we have shown that the impact of internal fragmentation on storage utilization of allocators is negligible. In fact, external fragmentation is the dominant factor in measuring storage utilization. By contrast, we have shown that the impact of internal
Figure 3.10: Total Number of Cache Misses, Cache size = 16 KBytes, Block size = 32 Bytes

Figure 3.11: Total Number of Cache Misses, Cache size = 16 KBytes, Block size = 64 Bytes
Figure 3.12: Total Number of Cache Misses, Cache size = 32 KBytes, Block size = 32 Bytes

Figure 3.13: Cache Miss Rates, Cache size = 16 KBytes, Block size = 32 Bytes
fragmentation on locality behavior of applications is very significant. Indeed, internal fragmentation counters spatial locality of allocation-intensive applications. We have designed an allocator that is different with existing ones in only one aspect “it minimizes internal fragmentation”.

We have conducted our experiments on Alpha processor and used ATOM for simulating different caches. The data in this chapter show that Exact Fit allocators, with low internal fragmentation, report 25% cache miss reduction, on average, when compared with other types of allocators.

Reusability is another feature, not practiced by existing allocators, that strengthens spatial locality. Reusability can be improved in allocators using LIFO lists. If an allocator explores segregation for speed, it can use a small list, say a Free Standing List, to improve reuse. When objects are freed, they will be placed in this list if the list is not full. An allocator that exploits Free Standing List, first searches this list to find the appropriate chunk of memory for satisfying the allocation request. It is likely that the recently freed objects in the free standing list may still be in CPU cache - thus reducing some cache misses in accessing a newly allocated object from the free standing list. It is clear that if an allocator practices reusability properly, it can improve the temporal locality.
4.1 Introduction

The speed gap between CPU and memory continues to widen and memory latency continues to grow. In the next two decades LADI processor may be able to beat Moore’s law [50], consequently memory speed will lag farther behind and memory latency will become even more pronounced.

Standard techniques such as deeper memory hierarchies and larger on chip caches fail to tolerate memory latency, mainly because application’s data sizes are growing and programming styles are changing. Nowadays, programmers practice the use of linked data structures, which requires dynamic memory allocation. The proximity of storage layout of such applications does not imply the same degree of spatial locality that array based applications’ does.

More recent approaches such as Multithreading [2, 33, 36, 54, 73], Prefetching [13, 47, 53], Jump Pointers [64], Memory Forwarding [48], and Stream Buffers [31, 51, 57] have been explored to address memory latency in pointer based applications. Multithreading tends to combat latency by passing the control of execution to other threads when a long latency operation is encountered. Prefetching tries to predict the access patterns of data ahead of time and bring what is needed into the cache. Jump Pointers provide direct access to non-adjacent nodes in linked data structures to alleviate the shortcomings of prefetching when insufficient work is found to cover the prefetching latency. Memory Forwarding relocates non-adjacent nodes to adjacent memory spaces. Stream Buffers are FIFO buffers added to the memory hierarchy between two adjacent levels, parallel with first level cache, or as a substitution of second level cache to filter out data accesses to streams. Multithreading requires parallelism
in the applications and extra hardware that manages threads and switching among threads. Software-controlled multithreading relaxes the hardware complexity by shifting the responsibility of switching time decision from hardware to software and enhances the performance when application lacks parallelism. Prefetching, Jump Pointers, and Memory Forwarding do not require parallelism in an application, but hardware or software overhead is added to the system. Hardware overhead decelerates the clock (“simpler is faster”), whereas software overhead uses CPU cycles and pollutes the cache. Stream Buffers are used for prefetching stream of data into a buffer; therefore, stream buffers are useful only when applications possess high spatial locality; for example, scientific applications.

Our approach of tolerating memory latency is originally motivated by a different trend in the design of systems, viz., Intelligent Memory Devices such as Berkeley IRAM [58], which is a stand alone vector processor on chip with DRAM for multimedia applications. Our research, aimed for better performance for general purpose systems, is also motivated by a different trend in the field that suggests distribution of processing powers for higher parallelism. Similar to Active Pages [56], within the DRAM chip, in our work we use a small processor as an aid to the main CPU, particularly for dealing with data intensive operations. These data intensive operations include memory management, prefetching of data, management of jump pointers, relocation of data (i.e., memory forwarding) to improve locality, and array based operations (if we include a vector processor like IRAM). In this chapter we will explain the architecture and applicability of our system - Intelligent Memory System.

The remainder of this chapter is organized as follows: Section 2 describes our Intelligent Memory System. Section 3 presents potential applicability of Intelligent Memory System. Section 4 is dedicated to the related work in Process In Memory devices. Finally, we conclude our discussion about Intelligent Memory System in Section 5.
4.2 Intelligent Memory System

We propose two streams of design for Intelligent Memory System: an extension to the centralized controllers\textsuperscript{1}\cite{16, 26, 27, 37}, and Embedded DRAM \cite{46, 56, 58}. Embedded DRAM limits the amount of memory on DRAM chip. With current gigabit DRAM technology, we can mount no more than 256 MByte DRAM and reasonable logic (powerful enough to perform operations such as memory management functions) on a single chip. Centralized controller design suffers from poor execution speed since it needs to communicate with DRAM chips via common bus. In these two designs conventional memory bus interface needs to change. Figure 4.1, 4.2 depict high level design of these two configurations.

It is reported that 5 Gbit DRAM technology can accommodate up to 32 Kbits internal

\textsuperscript{1}Non-interleaved SDRAMs, or SLDRAMs in Rambus configuration
bus width with 1.7 GHz speed for embedded DRAM, and 0.8 GHz 128 bit wide external bus for standard DRAM [28]. Using embedded DRAM or centralized controller, we feel that IMEM is feasible with current technologies. The functionality required to implement the logic of IMEM can be achieved by either using ASIC or more traditional pipelined execution engines. In the following subsections we discuss the architecture and software of Intelligent Memory System.

4.2.1 Architecture of Intelligent Memory System

As mentioned earlier, we suggest two implementations of Intelligent Memory System: an extension to centralized controller (eController), and the use of embedded RAM instead of DRAM arrays (eRAM). Figure 4.3 and 4.4 depict the architectures for these implementations respectively. In these figures, we assume that the main task of the Intelligent Memory System is to perform Memory Management. In Section 3 we will explain the use of Intelligent Memory System for other applications; the goal of this section, however, is to provide a simple design for a chosen purpose. We have chosen Memory Management algorithms since we are interested in both the hardware and software aspects of Memory System. Nonetheless, the same architecture with slight modifications can be adopted for offloading other data-intensive service functions such as prefetching.

In eController, the requests and control signals are buffered via five queues shown in Figure 4.3. Queue 1 and 3 buffer the requests’ data, that are Read, Write, Allocation, and De-allocation data; queue 2 is used to buffer the control signals. For Write and De-allocation requests, on the successful completion, the controller needs to only inform the CPU that the requests progressed and completed without errors. For Read and Allocation requests, however, the controller needs to provide CPU with data, which are retrieved from DRAM array. The Intelligent Memory System uses queue 3 to buffer Read and Allocation data. Intelligent Memory System Processor employs queue 4 and 5 for final Write and Read
Figure 4.3: Intelligent Memory System Architecture, Extended Controller - eController

Figure 4.4: Intelligent Memory System Architecture, using eDRAM instead of DRAM Arrays
data sent to or retrieved from DRAM arrays.

eDRAM design of Intelligent Memory System uses the same method of buffering the requests’ data and control signals. Queue 4 and 5 are no longer needed, and DRAM interface uses its own buffers to manage such traffic. The rest of the system is similar to the eController design of Intelligent Memory System.

In both designs, the processor can employ split or unified instruction and data cache. In the next chapter, we will show that 1 KBytes is sufficient for the data cache of the processor. One can consider 512 Byte cache for caching the instructions of Intelligent Memory System processor, since the size of code segment run by the processor in Intelligent Memory System is small. Intelligent Memory system processor reorders the requests for better performance. It can also decide to prefetch data if it recognizes a pattern of the data requests. It also provides other functionalities that a memory controller is responsible for.

To conclude this subsection, we would like to mention that it is also possible to place the Intelligent Memory System Processor on chip with the CPU; we call it Extended CPU (eCPU) shown in Figure 4.5. The main advantage of this configuration is that the System Bus does not need to change. Compiler has also a simpler task in terms of scheduling the service functions to the Memory Processor. In this design, the only requests between CPU and Memory Processor are those of the service functions. For example, for Memory Management functions, the requests are only Allocations and De-allocations. Second Level cache is shared by both CPU and Memory Processor, which adds to the complexity of the Second Level Cache Controller.

In any of the proposed architectures for Intelligent Memory System, we can use either programmable (pipelined) or ASIC design. If we choose simple five-stage RISC pipelined design, service functions of any form can be executed by the Memory Processor. This, of course, requires changes to system software. The changes, as we will point out in the next subsection, will be transparent to the programmer. Designing ASIC Memory Processor, we
can provide higher speed and less complex system software. ASIC Memory Processor can only serve one purpose for which it is designed.

### 4.2.2 Compiler and Operating System of Intelligent Memory System

Software issues in Process In Memory devices (PIM - a general form of Intelligent Memory System) are not different from those in multiprocessor systems. Although PIM devices have been proposed in mid 1990s, their software aspects have not been studied completely. Researchers have simulated different types of PIMs based on specific applications. This subsection briefly explains software issues of PIM devices in general and for variety of applications.

Kernel is shared by Processor in Memory and CPU, which creates and modifies the process table. For synchronization purposes, system hardware provides hardware primitives.
such as Load Linked and Stored Conditional. Task scheduling in PIM operating system needs special care, since tasks running on Memory Processor are of special types, which will be explained in Section 3. CPU provides beginning address of the code and data segments of the tasks needed to be executed by Memory Processor.

Memory Processor is a simple integer unit; therefore, compiler needs to partition application’s code to memory and CPU tasks. Tasks that are meant to run on Memory Processor, preferably, should be data-intensive; which is also the responsibility of task partitioning portion of the compiler. In ASIC design of Intelligent Memory System, where a part of the application’s code, for example service functions such as memory management, is chosen to run on Memory Processor, both compiler and operating system will become much simpler. In fact, the only difference of ASIC Intelligent Memory System and uniprocessor system is the CPU instruction set. Minor changes, therefore, are needed in pipeline design of the main CPU. In chapter 5 we will explain an example of such system for running memory management functions.

A number of other methods exist and can be adopted for software portion of Intelligent Memory System. For example, the communication between CPU and Memory Processor can be made possible via traditional Mapped I/O and the Memory Processor’s code can be considered as an Interrupt Service Handler. The ease of such methods, however, comes as the expense of their efficiency.

4.3 Applicability of Intelligent Memory System

The main goal of PIM-Based systems is to offload some part of application’s code to Memory Processor to achieve better performance. Memory Processor can be scheduled to run specific functions, say service functions, or a portion of application’s code in general. However we schedule the code, it should possess certain properties. Memory Processor is on DRAM chip; therefore, its code needs to be data-intensive. Any fragment of the application’s code run by
Memory Processor should not have floating point operations, since Memory Processor is a simple integer processor. CPU and Memory Processor should be utilized to run in parallel, and hence data dependency between their codes should be minimized. In the following subsections we study three types of applications that are well suited for incorporation in Intelligent Memory Systems.

4.3.1 Runtime Libraries

Runtime Libraries such as dynamic memory allocation and de-allocation are used in any of today’s applications. For example, in Object Oriented applications, constructing and deconstructing an object invoke allocation and de-allocation routines either implicitly by the runtime system or explicitly by the programmer. These Runtime Libraries do not have any floating point operations, and they are very simple in nature. As shown in Chapter 2, for example, allocation and de-allocation routines are invoked hundreds of thousand times in Java applications. Furthermore, de-allocation routines, if executed by Memory Processor, can run fully in parallel with CPU’s code. With a little help from compiler, allocation routines can also run in parallel, although partially, with CPU’s code. Runtime Library routines, when run on CPU, entangle with the application’s code and become the main cause of cache pollution. In chapter 5, we will show that removing allocation and de-allocation runtime routines from application’s code and running them separately on Memory Processor can reduce application’s cache misses by as much as 60%.

4.3.2 DSP applications

DSP applications are mainly divided into two groups: image and speech processing. Image processing involves two dimensional array operations, whereas speech processing involves one dimensional transformation such as Fast Fourier and cosine transforms. Parallelism in these applications are of Single Instruction Multiple Data type (i.e., SIMD); therefore, Vector...
Processors are the best choice for running such applications. Furthermore, DSP applications are very data-intensive, and they possess almost no temporal locality. All of these properties make them suitable for running on Memory Processor. Memory Processor for such applications, however, should be implemented as a vector processor for better performance. Berkeley IRAM is a stand alone Vector Processor to serve for DSP applications [58].

4.3.3 Prefetching, Jump Pointers, Memory Relocation, and Stream Buffers

Applications can be classified into two categories in terms of locality: they exhibit either high spatial locality and very week, if any at all, temporal locality (stream data and floating point programs), or high temporal locality and week spatial locality (Object Oriented programs). Whatever locality exhibited by an application, memory hierarchy, in general, and cache, in particular, fail to completely address the Memory Wall problem [65, 75, 78]. Stream Buffers and their associated prefetching engines are proposed to buffer stream of data that are accessed only once and discarded soon after [31, 51, 57]. Prefetching techniques try to prefetch the nodes of linked lists ahead of time so that the memory latency can be hidden [13, 47, 53]. Jump Pointers exploit pointers not to the next node but to the second or third nodes and prefetch them ahead of time when the workload is not sufficient to cover the prefetching latency [64]. Memory relocation tends to relocate the nodes of linked lists to create greater spatial locality [48]. For all of these techniques, we either need to add hardware assistance (prefetching engine) or to use CPU cycles if they are implemented in software. Latter will itself pollute the cache so that the main goal - better cache locality - will become defeated. Memory Processor, however, can assist CPU in implementing these techniques.

The use of Intelligent Memory System is not limited to the above applications. Indeed, it is our prediction that in the near future companies substitute commodity DRAM products with Intelligent Memory Systems.
4.4 Related Work

The first appearance of the term “Memory Wall” problem in the literature was due to two papers in ACM Computer Architecture News in 1995 [75, 78]. Since then almost all researchers in the field have tried to confront the problem in different ways. Process In Memory devices’ trend has become one of the most attractive and promising solution studied ever since. Research in Process In Memory (PIM) devices can be classified into three categories: tiles of PIMs as small yet powerful computing engines, PIMs as stand alone systems, and PIMs as substitutions of DRAM chips in the conventional architectures. Following subsections explain these categories.

4.4.1 Tiles of PIMs

Modern high performance computers use sophisticated techniques such as register scoreboard and out-of-order issue to hide memory latency. Tiles of PIMs, however, is another approach that suggests simple PIMs tied together via fast interconnect network to form a supercomputer [65]. Figure 4.6 shows the architecture of a node in such systems, and Figure 4.7 shows the system interconnection. One of the advantages of using several memory banks on chip together with a processor is that the sense amplifiers of the memory banks can serve as fully set associative cache for Memory Processor. For example, Saulsbury and et. al. have used 16 independent DRAM banks and 3 buffers for each bank [65]. Memory banks are two dimensional arrays of bits, and each buffer is able to cache one row of data. If a memory bank is 16 Mbits, each buffer will be 4096 bit wide\(^2\), and 16 banks of memory result in \(3 \times 16 \times 4096\) bits of cache (fully set associative). This design provides a flow of enormous data to Memory Processor, and hides memory latency since all accesses occur on chip.

\(^2\)4096 \times 4096 = 16M
Figure 4.6: Typical example of a node architecture in a Tiles of PIMs systems

Figure 4.7: System Interconnection in a Tiles of PIMs systems, Reproduction by permission of authors and ACM [65]
4.4.2 PIMs as Stand Alone Systems

Some researchers have utilized PIMs in vector processing as stand alone processors; Berkeley IRAM is an example of this group [58]. Such an IRAM consists of a vector processor and a simple scalar processing engine on chip with DRAM modules. IRAM is designed for DSP and multimedia applications which contain significant amount of vector level parallelism. The Berkeley IRAM project demonstrates that even when operating at a moderate clock rate of 500 MHz, Vector-IRAMs can achieve 4 Gflops, while Cray machines achieved only 1.5 Gflops [7]. Recent efforts in IRAM project converged to a new architecture called CODE which addresses the shortcomings of Vector-IRAM [42].

Embedded DRAM (eRAM) is another family of stand alone Processor In Memory Devices [55]. M32R, the main core of eRAM which contains a CPU and DRAM, has been used in variety of applications. M32R/D with an off-chip I/O ASIC is used for multimedia applications, for example, JPEG compression/decompression. Open Core M32R which integrates CPU, SRAM, DRAM, and versatile peripherals into a single chip can be used for portable multimedia devices. M32R media Turbo, which includes a vector processor and a super-audio processor along with a CPU and DRAM, is promoted for applications that demand high performance when dealing with large streams of data (such as speech recognition and image processing). M32R core is based on an extensible dual-issue, VLIW instruction set.

4.4.3 PIMs as Substitutions for DRAM Chips

This class of PIM devices covers a wide range of applications. We have chosen two architectures of this group called Active Pages [56] and FlexRAMs [22].

Active Pages consist of pages of memory and a set of associated functions for each page to improve processing power. RADram\(^3\) is the basis of Active Pages that gives flexibility in customizing functionality for each application. RADram replaces DRAM chips in conven-

---

\(^3\)Reconfigurable Architecture Dram
tional architecture with some additional control logic and control lines. Figure 4.8 depicts Active Page architecture for a quad page of memory. Each page of DRAM in Active Page architecture is associated with reconfigurable Computing Logic that can be reconfigured to run any form of code.

FlexRAM replaces DRAM chips with PIM devices but leaves some of the DRAM chips as commodity DRAMs in the system assuming that not all the data in the memory need computing power. Figure 4.9 shows the architecture of FlexRAM which consists of 64 PIMs. Each PIM is a simple pipelined processor with its designated cache and 1 MBytes of DRAM. This configuration of FlexRAM is particularly hard to program; researchers, however, have made substantial progress to ease the programming aspect of FlexRAM. Fraguela and et. al. have proposed a set compiler directives\textsuperscript{4} and Intelligent Memory Operations\textsuperscript{5} for programming FlexRAMs [22].

4.5 Conclusions

Memory latency is the great issue of concern for today’s computer system researchers. Although the number of transistors on chip doubles every 18 months, for last 15 years, we have failed to utilize this transistors’ power to hide memory latency. Every so often a layer of hardware complexity, but hardware added to the computer system requires special properties in applications to gain performance. For example, superscalars need sufficient ILP\textsuperscript{6}, whereas Multithreaded architectures require coarse grain parallelism (so-called Threaded Level Parallelism).

We have proposed a viable solution to hide latency that is Intelligent Memory System for offloading library service functions which, in conventional architectures, entangle with the applications code and become the major cause of cache pollution. In this chapter, we have

\textsuperscript{4}CFlex, high level compiler directives
\textsuperscript{5}highly optimized Library calls
\textsuperscript{6}Instruction Level Parallelism
Figure 4.8: Active Page Architecture (4 pages), Reproduction by permission of the authors and ACM [56]

Figure 4.9: FlexRAM architecture, Reproduction by permission of authors and ACM [22]
illustrated Intelligent Memory System in three different configurations, eController, eDRAM, and eCPU; all require small changes in system bus and programming. We have shown that Intelligent Memory Systems can be used for variety of applications.

It is our goal to use Intelligent Memory Systems for performing data intensive library functions such as dynamic allocation and de-allocations. Data intensive library functions when executed by main CPU become the major cause of eluding the temporal locality of applications. Transparent to programmer, we can offload these service functions to Memory Processor for execution. Unlike other Intelligent Memory System research, our technique requires no change in the programming language nor does it require any special properties in applications.
In this chapter, we present a novel idea called Intelligent Memory Management that utilizes Intelligent Memory Systems for executing allocation and de-allocation service functions. The goal of Intelligent Memory Management system is to minimize the cache pollution caused by memory management algorithms when they are executed by the CPU. We have chosen memory management algorithms for executing by a separate processor in memory for several reasons. First, memory management functions are used in almost every program. In Object Oriented and linked data structured applications, especially, allocation and de-allocation functions are invoked very frequently. These memory management functions are themselves very data intensive, but require only integer arithmetic. Therefore, a simple integer processor embedded inside DRAM chip offers a viable option for migrating allocation/de-allocation functions from main CPU to DRAM, and eliminates the cache pollution caused by those functions. In this chapter we will show that moving memory management functions from main CPU to Intelligent DRAM eliminates, on average, 60% of cache misses (as compared to the conventional method where allocation/de-allocation functions are performed by the main CPU). Furthermore, we compare the performance of a variety of well known general purpose allocators. We also show that Binary Tree allocators result in better cache localities when compared to other allocators [34, 60, 61].

In this chapter, first we introduce our Intelligent Memory Management’s architecture. Section 3 represents the framework used to empirically validate the claim of this chapter. Final sections illustrate the results and draw the conclusions based upon the results.
5.2 Intelligent Memory Management System

In the previous chapter, we have given a general view of Intelligent Memory System. In this chapter, we will focus on a subset of Intelligent Memory Systems called Intelligent Memory Manager (IMM), which is a specialized processor embedded in DRAM chip for performing memory management functions. This section highlights the characteristics of IMM which replaces DRAM chips in conventional architectures.

Architecture of IMM can be ASIC, reconfigurable logic, or traditional pipelined design. ASIC IMM is the fastest, since it is designed to serve only one purpose. It also benefits from the simple interface and it needs very little software support. The main problem with this design is that only one memory management algorithm (for example Segregated Binary Tree) can be implemented to meet all the applications, thus removing the flexibility of adapting memory management based on application’s needs. Reconfigurable and pipelined architectures are more flexible, since they can be programmed. The programmer can choose a memory management technique which suits best for her application and reconfigure the reconfigurable IMM for the chosen memory management technique. Changing to another memory management technique, however, needs new reconfiguration. In contrast, programming a pipelined IMM for memory management algorithms can be deferred to the load time.

Any chunk of memory is used by either the running application (live objects) or the memory manager (free chunks of memory); therefore, no chunk is accessed at the same time by the application and the memory manager. This trivial observation simplifies the design of IMM for which no synchronization is required.

In any of the designs that we suggested above, system control lines must be changed. We propose the addition of two functions to the standard memory interfaces.

- allocation and de-allocation interfaces (additional interfaces)
  - allocate(size)
– \text{de-allocate}(\text{virtual-address})

• standard conventional interfaces

– \text{read}(\text{virtual-address})

– \text{write}(\text{virtual-address},\text{data})

On any allocation request, CPU places the size of the request on the data bus and asserts the allocation control line. IMM, which is programmed for a specific memory management algorithm, receives the size and looks for a free chunk of memory at least as large as the size requested. After finding the appropriate chunk, IMM places the address of the chunk on the data lines and informs the CPU that the result is ready. CPU, in turn, receives the address via data lines and proceeds with its execution. If compiler schedules the allocation requests ahead of time before CPU needs the objects, IMM can run in parallel with CPU and provide the addresses of the objects before they are used. For de-allocating an object, CPU places the address of the object on the data lines and asserts the de-allocation control line. When IMM acknowledges CPU that the address of the object has been observed, CPU can proceed with its execution; therefore, de-allocation of an object by IMM runs fully in parallel with CPU.

5.3 Experimental Framework

To confirm the claims we have made (viz., separating memory managements from CPU results in fewer cache misses), and also to compare the cache performance of different memory managers (allocators), we have conducted two stems of experiments on Alpha 21264 running Tru64 operating system [38]. First, a single process is used to execute both application and the memory management functions. This scenario simulates conventional systems using a single CPU for both application and memory manager. Next, a pair of processes are used to execute application and its service functions separately. This simulates the use of a sepa-
rate processor for memory management functions, which can potentially be embedded in a DRAM chip. The latter experiment exploits shared memory segment for interprocess communication. These processes are instrumented using ATOM instrumentation and analysis routines [21]. Instrumentation routines detect the memory references and call analysis routines, which simulate different cache organizations. The use of shared memory interprocess communication adds a considerable amount of system overhead and consequently blurs the aim of this work. To avoid such artifact, using instrumentation routines, to the extent possible, we have discarded the references made by interprocess communication system calls. We have also separated the application heap and analysis routines’ heap so that ATOM activities do not impact the locality behavior of the applications. Figure 5.1 depicts IMM’s framework.

To illustrate the wide applicability of our claim we have employed two sets of benchmarks, a subset of SPEC CINT2000 [25] and a subset of benchmarks commonly used to evaluate memory allocators. They are briefly explained in Table 5.1.

In this work four general purpose allocators are studied for their locality behaviors based on different allocation strategies. The allocators studied in this work are:

- **BSD allocator**\(^1\). This allocator is an example of a Simple Segregated Storage technique [79]. It is among the fastest allocators but it reports high memory fragmentation. In the figures this allocator is referred to as “bsd”.

- **Doug Lea’s allocator**. Perhaps the most widely used allocator is Doug Lea’s. We have used version 2.7.0, an efficient allocator that has benefited from a decade of optimizations [45]. For request sizes greater than 512 bytes it uses a LIFO Best Fit method. For requests less than 64 bytes it uses pools of recycled chunks. For sizes in between 64 and 512 bytes it explores a self adjusting strategy to meet the two main objectives of any allocator: speed and high storage utilization. For very large size

\(^1\)known as Chris Kingsley’s allocator - 4.2 BSD Unix
requests (greater than 128 Kbytes), it directly issues `mmap` system call. “lea” is used to reference data for this allocator in our figures.

- **Segregated Binary Tree.** SBT contains 8 binary trees each for a different class size. Memory chunks less than 64 bytes and greater than 512 bytes are kept in the first and the last binary tree respectively. Each binary tree is responsible for keeping chunks of a unique size range, and sizes range in 64 byte intervals. For example, the second binary tree’s range is [64,128) (viz., if a chunk’s size is x then $64 \leq x < 128$). In the figures “sbt” is used to refer to the data set belonging to this allocator.

- **Segregated Fit.** We have written our own version of Segregated Fit algorithm referred to as “sgf”. In the structure, this allocator is similar to our SBT but the memory chunks are kept in segregated doubly linked lists instead of binary trees. LIFO Best Fit is chosen for placement policy of each list.
As mentioned in the last section, we have carried out our experiments under two scenarios:

- **Conv-Conf**: Conventional Configuration, in which case both application and its allocator are running on the main CPU using a single cache.

- **IMM**: Intelligent Memory Manager, where a separate processor executes memory management functions. IMM consists of two parts; IMM-application that is the application code running on the main CPU, and IMM-allocator that is the memory management operations running on the processor embedded in DRAM chip (with a separate cache).

Table 5.2, 5.3, and 5.4 show total number of references for Conv-Conf, application part of IMM (total number of loads and stores issued by main CPU when running the applications), and allocator part of IMM (total number of references issued by DRAM logic when running
the allocator portion of the applications) respectively.

Several observations must be made with the data shown in these tables. From Table 5.3,

<table>
<thead>
<tr>
<th>Bench/Alloc.</th>
<th>bsd</th>
<th>lea</th>
<th>sbt</th>
<th>sgf</th>
</tr>
</thead>
<tbody>
<tr>
<td>boxed-sim</td>
<td>2.54e+09</td>
<td>2.54e+09</td>
<td>2.54e+09</td>
<td>2.54e+09</td>
</tr>
<tr>
<td>cfrac</td>
<td>2.78e+09</td>
<td>2.79e+09</td>
<td>2.79e+09</td>
<td>2.79e+09</td>
</tr>
<tr>
<td>espresso</td>
<td>1.41e+08</td>
<td>1.41e+08</td>
<td>1.41e+08</td>
<td>1.41e+08</td>
</tr>
<tr>
<td>gzip</td>
<td>1.01e+10</td>
<td>1.02e+10</td>
<td>1.02e+10</td>
<td>1.02e+10</td>
</tr>
<tr>
<td>parser</td>
<td>9.28e+08</td>
<td>9.29e+08</td>
<td>9.29e+08</td>
<td>9.29e+08</td>
</tr>
<tr>
<td>ptc</td>
<td>9.2e+07</td>
<td>9.2e+07</td>
<td>9.2e+07</td>
<td>9.2e+07</td>
</tr>
<tr>
<td>twolf</td>
<td>2.91e+08</td>
<td>2.92e+08</td>
<td>2.92e+08</td>
<td>2.92e+08</td>
</tr>
<tr>
<td>vpr</td>
<td>1.8e+10</td>
<td>1.8e+10</td>
<td>1.8e+10</td>
<td>1.8e+10</td>
</tr>
</tbody>
</table>

Table 5.3: Total Number of References for IMM - Application

it should be noted that the number of references made by the application (IMM-application) are approximately the same for all allocators, across all benchmarks. The exception is for “cfrac” and “parser”. The “bsd” allocator seems to have fewer references, since these appli-
cations fit better with the allocation strategies used by “bsd”. In almost all benchmarks with different allocators, represented data demonstrate that the number of memory references of Conv-Conf is equal to the sum of memory references of IMM-application and IMM-allocator. For “sbt” and “sgf” allocators we observe a decrease in the number of references for IMM. We suspect the reason for such a behavior due to our unoptimized algorithms. When compared with more established allocators that have benefited from years of fine-tuning, our allocators provide great opportunities for hardware based optimizations such as out of order execution and branch predictions. We have conducted our experiments on Alpha 21264, an out-of-order microprocessor that is able to fetch four instructions per cycle [38]. It employs a sophisticated branch prediction and speculative instruction fetch/execute. While such hardware optimizations are also available for other allocators, since the implementations are already higher optimized, separating the allocator functions have not shown significant improvements, unlike our allocators. Among benchmarks used throughout this chapter, “ptc” disagrees with others in showing fewer memory references when application and its allocator functions are separately executed by two processes (IMM). This happens because “ptc” con-

<table>
<thead>
<tr>
<th>Bench/Alloc.</th>
<th>bsd</th>
<th>lea</th>
<th>sbt</th>
<th>sgf</th>
</tr>
</thead>
<tbody>
<tr>
<td>boxed-sim</td>
<td>1.36e+08</td>
<td>1.11e+08</td>
<td>1.01e+08</td>
<td>1.02e+08</td>
</tr>
<tr>
<td>cfrac</td>
<td>318459</td>
<td>268149</td>
<td>248351</td>
<td>249147</td>
</tr>
<tr>
<td>espresso</td>
<td>1.3e+08</td>
<td>1.86e+08</td>
<td>1.01e+08</td>
<td>1.01e+08</td>
</tr>
<tr>
<td>gzip</td>
<td>1.4e+06</td>
<td>2.9e+06</td>
<td>1.2e+06</td>
<td>1.2e+06</td>
</tr>
<tr>
<td>parser</td>
<td>1.53e+08</td>
<td>1.97e+08</td>
<td>1.21e+08</td>
<td>1.23e+08</td>
</tr>
<tr>
<td>ptc</td>
<td>5.9e+06</td>
<td>9.9e+06</td>
<td>6.2e+06</td>
<td>6.2e+06</td>
</tr>
<tr>
<td>twolf</td>
<td>676653</td>
<td>675783</td>
<td>542219</td>
<td>548174</td>
</tr>
<tr>
<td>vpr</td>
<td>580947</td>
<td>675783</td>
<td>542219</td>
<td>512907</td>
</tr>
</tbody>
</table>

Table 5.4: Total Number of References for IMM - Allocator
tains only allocation requests and no de-allocation. Although we have partially removed the references associated with interprocess communication overhead due to our framework, this overhead for “ptc” tends to dominate the impact of separating the execution of application and its service functions in terms of number of references. Nonetheless, we feel that the cache miss reduction achieved by excising the allocator functions from the application is still verified by our data.

5.4.1 Comparison of Cache Performance

It is our aim to show the improvement in cache performance obtained using Intelligent Memory Manager. This improvement holds valid for all cache levels. However, first level cache activity attracts more interest because of its influence on CPU execution time. Hence, in this subsection we include data for first level cache only. We have chosen the cache sizes and block sizes based on modern systems.

Figure 5.2 and 5.3 show the total number of cache misses for Conv-Conf and IMM-application with 32 KBytes cache and 32 Bytes blocks. In almost all benchmarks, it is very clear that IMM configuration has removed the cache pollution caused by memory management service functions. This is better shown by Figure 5.4, which reports the percentage of IMM-application cache miss improvement. The data shows 60% reduction in the number of cache misses on average. Similar reductions will result for separating the execution of any service function from the application if the service function is invoked very frequently. Memory management operations are just examples of such service functions.

As mentioned before “ptc” behaves somewhat differently than other applications in our benchmark suite. This is partially because “ptc” contains only allocations. In both “twolf” and “boxed-sim”, computation core of the application dominates execution time requiring fewer memory management calls. Thus, these applications show insignificant improvements on cache performance. It should be noted that negative impact (i.e., increase in cache misses)
Figure 5.2: Conv-Conf Cache Misses, Cache size = 32 KBytes, Cache Block size = 32 Bytes

Figure 5.3: IMM-application Cache Misses, Cache size = 32 KBytes, Cache Block size = 32 Bytes
Figure 5.4: Percentage of IMM-application cache miss improvement as to compare with Conv-Conf, Cache size = 32 KBytes, Cache Block size = 32 Bytes

is primarily due to the artifact of our experiments involving the shared memory interprocess communication. Although we have done our best to eliminate such memory references, it is impossible to be certain that all references caused by such communication have been removed entirely.

5.4.2 Impact of Cache Parameters

In the next experiments we doubled the cache line size to view the impact of changing this cache parameter. Figure 5.5 and 5.6 depict the results for Conv-Conf and IMM-application. Figure 5.5 reports fewer misses for Conv-Conf when cache block size is increased. Figure 5.6 shows that on average the number of misses increased in the case of IMM, when the cache block is enlarged, albeit slightly.

When application with its memory management functions is running on the main CPU (Conv-Conf), it certainly possesses higher spatial locality as compared to the case of IMM. Each chunk of memory, free or live, contains its size information (normally the first four or eight bytes of the chunk). When the chunk becomes free, it will also contain pointers used
by the allocator to track the list of the free chunks. These items are kept in the header of every memory chunk. The role of allocator obliges it to visit free chunks of memory and hence either reading or modifying the information kept in each chunk. When the execution of memory management functions (allocation and de-allocation) entangled with application, its behavior elevates spatial locality and lessens temporal locality of the application. Separating memory management functions from application improves temporal locality (on average 60% as shown previously), and decreases spatial locality of the application very slightly. Increasing cache block size for Conv-Conf results in fewer cache misses (comparison of Figure 5.2 and 5.5), because spatial locality of the applications is more utilized. On the other hand, it results in more misses for IMM-application since it circumvents the temporal locality of the application\(^2\) (comparison of Figure 5.3 and 5.6).

### 5.4.3 Comparing cache behavior of Allocators

Storage Utilization analysis as well as execution performance of different allocators have been adequately studied by others \([6, 30, 76]\). Surprisingly, locality behavior of allocators has not been reported as widely. This subsection is an effort, although for only a subset of allocators, that represents the cache behavior of such useful service functions of computer system.

Cache data shown here belongs to the allocator portion of IMM configuration that runs on a separate logic integrated with DRAM chip. A small cache is considered to serve the IMM-allocator processor, because the chip area and number of transistors on chip are limited. Figure 5.7 illustrates cache performance of different allocators for 512 Bytes direct mapped cache with 32 Bytes block size. “lea” allocator shows the worst performance due to its complexity and hybrid nature. Mixing `sbrk` and `mmap` system calls, which is prac-

\(^2\)Larger cache block size with fixed cache size means fewer blocks. This favors spatial locality. Certainly, it is not favorable for temporal locality since fewer blocks of memory can be mapped to the cache at the same time.
Figure 5.5: Conv-Conf Cache Misses, Cache size 32 KBytes, Cache Block size 64 Bytes

Figure 5.6: IMM-application Cache Misses, Cache size 32 KBytes, Cache Block size 64 Bytes
Figure 5.7: IMM-allocator cache misses, 512 Bytes Direct Mapped Cache, 32 Bytes Block size

noticed by "lea" allocator for different object class sizes, may be a cause for such behavior.
"lea" allocator is followed with "bsd", which also benefited from strong segregation ("lea" allocator, as mentioned before, keeps several free lists, one for each class size; it also uses different policies for different free lists) for speed. This segregation causes "bsd" to reveal poor locality behavior.

Both "sbt" and "sgf" perform almost the same as they benefit strongly from memory chunk reusability. Their implementation leads to the reuse of recently freed objects. Reusability seeds temporal locality, which is the main advantage of these two allocators.

It is quite obvious that cache performance of an allocator is directly associated with its storage utilization. Allocators with higher storage utilization report better cache performance.
5.5 Conclusions

In this chapter, we presented the cache data collected from experiments on two schemes. First we conducted our experiment when both application and its memory management functions are executed on the main CPU (Conv-Conf). We have also carried out our work by separating the execution of memory management functions and application (IMM). The cache data resulted from the latter shows 60% improvement on average. In the case of IMM, our experimental framework caused some additional overhead due to the interprocess communication, which we tried to remove by disregarding the references caused at the time of communication - albeit not completely. We believe that if the interprocess communication overhead is completely removed (i.e., when a separate processor is used for IMM), we will achieve even more cache miss reduction with IMM configuration.

We have also studied the amounts of cache pollution caused by different memory allocation techniques. Some techniques have resulted in more pollution while maintaining their goal of high execution performance. For instance, Simple Segregated Storage techniques are the best in terms of speed, but as we have shown in this work, they illustrate poor cache performance and high cache pollution. Since employing a separate hardware processor eliminates the cache pollution caused by an allocator, we can consider the use of more sophisticated memory managers. Other dynamic service functions such as Jump Pointers to perfectch linked data structures and relocation of closely related objects to improve localities can also cause cache pollution if a single CPU is used - such service functions drag the objects through the processor cache. These functions can also be off-loaded to the allocator processor of Intelligent Memory Manager in order to benefit from their performance advantages, while maintaining low cache miss rate.
CHAPTER 6
HARDWARE IMPLEMENTATION AND EXECUTION PERFORMANCE BENEFIT OF IMEM

6.1 Introduction

On the verge of new millennium, computer system’s researchers’ main agenda is shaped with the emergence of the fact that VLSI technology allows 1 billion transistors fabricated on a single chip with multiples of GHzs clock speed [35]. We ought to utilize this enormous processing power to design a better computer system. “A better computer system” is an undefined abstract term and defining its aspects is well beyond the scope of this work. However, grasping the first stem of thoughts would suggest to design an SOC (System On Chip) composed of designated processing engines for executing different tasks. Having considered durable Amdahl’s law [4] - making the common case fast [24] - one would prefer to design some of these processing engines as special purpose processors to execute frequently invoked functions.

For object oriented applications, studies have shown that about 20 to 30% of the CPU execution time is spent on allocating and de-allocating the objects dynamically [5, 11, 17]. It has been also shown that about two-third of the memory management time is spent on allocation [60]. This is very well convincing that memory management functions can be considered as common case in today’s applications (i.e., object oriented applications).

In Chapter 4 of this work, we have presented three variations of IMEM, eController, eDRAM, and eCPU, analogous with respect to the fact that they all share a designated processing engine for executing memory management functions. In Chapter 5, we have studied the cache miss reduction as a result of executing memory management functions using
IMEM. In this chapter, however, we present existing hardware designs of memory management system. We also show the execution performance benefit of IMEM. For hardware design of memory management functions, we have considered the simplest allocation technique, Buddy System [41], to achieve a high speed for IMEM, since “simpler is faster”. To evaluate the performance benefit of our proposed IMEM system, we have extended the SimpleScalar simulator tool set [8], which ultimately includes a separate processor in the memory module of the simulator. We have run a set of three benchmarks and achieved, in best case, a 10% performance speedup.

The rest of this chapter is organized as follows: Section 2 demonstrates the existing hardware implementation of Buddy System allocator. Section 3 explains the simulation framework for CPU and IMEM. We also present the simulation results of IMEM in this section. Finally, in Section 4 we address the conclusion remarks and possible future trends that can reach more convincing results.

6.2 Buddy System Allocator and its Hardware Design

As mentioned in Chapter 2, Section 3, in Buddy System allocator the size of memory blocks is power of 2, and two adjacent memory chunks with the same size are called buddies. In the same manner as Segregated Free List allocator, the free chunks of the same size can be kept under the same list. Any two free buddies can, in principle, be coalesced and form a chunk twice as large. This process can be delayed to prevent the oscillation phenomenon [18]. Oscillation problem occurs when a series of allocations and de-allocations cause unnecessary splits and coalescences, in which case deferred coalescing can resolve the problem to some extent.

However the coalescing of the buddies are performed, deferred coalescing or immediate, Buddy System allocators are known for their two main properties. They perform very poor in terms of fragmentation and storage utilization, but they are very suitable for hardware
implementation. The restriction that all the memory chunks’ sizes are $2^k$ for some $k$ mandates the beginning addresses of the memory chunks to be multiples of $2^k$; therefore, with a simple logic, the beginning address of any chunk can be determined if its buddy has been freed. This is a particular procedure that any allocator needs to perform in the times of de-allocation. In the following subsection, we explain a simple hardware implementation of Buddy System that amortizes the bit-vectors for indicating free or allocated chunks [10, 59].

6.2.1 Bit-Map Buddy System

In Bit-Map Buddy System, the entire heap address space is divided into fixed size blocks. This fixed size is the smallest size of any object ever allocated by the allocator. For example, if the heap address space is 4 KBytes and if the smallest object size is 16 Bytes, the heap will be divided into 256 contiguous blocks of 16 Bytes. As name implies, Bit-Map Buddy System exploits a bit-vector associated with these blocks; each bit of the bit-vector corresponds to each block of heap. State zero of any bit of the bit-vector indicates that its correspondent block is free, otherwise the bit is set to one. Figure ref:figone illustrates the heap and its bit-vector for an allocator. In this hypothetical example, only first and last 4 blocks of the heap are in use by the running application (viz., they are allocated).

When an allocation request with size $s$-bytes arrives, the Bit-Map Buddy System allocator should perform three tasks in the following order:

1. It should first verify that if it can successfully satisfy the request. In other words, it needs to determine the existence of contiguous free space at least as large as $s$-bytes.

2. If such free chunk exists, the allocator should then find the starting address of the chunk.

3. Finally, it needs to update a portion of the bit-vector associated with either starting or ending $s$-bytes of the found free chunk\(^1\)

\(^1\)This depends on the strategy of design to either allocate the objects from lower or higher addresses of
Figure 6.1: 4 KByte Heap with 16 Byte Block and its 256 bit Bit-Vector

In this process, all the request and chunk sizes are assumed to be of a power of 2.

In our work, we exploit a modified version of the hardware implementation proposed by Puttkamer [59] and Chang-Gehringer [10, 11, 12], since they use an efficient design for Bit-Map Buddy System which consists of only combinational logic circuit. Implementation of each step of the allocation process follows:

Step 1: Is there any free chunk large enough to satisfy the request?

We use a Complete Binary Tree (CBT) of or-gates for bit-vector to verify the existence of such chunk. Figure 6.2 depicts or-gate CBT for a page of 8 blocks\(^2\) of memory. In the example shown by the figure, a bit-vector and its or-gate for a small portion of heap memory are shown. Any level of CBT is responsible for verifying the availability of free chunks of a unique size. For example, if there is an allocation request for 4 blocks of memory, the allocator checks the results of or-gates in the third level (level number 2)\(^3\). If any of the or-gates found free chunk.

\(^2\)Each block is, by assumption, 16 Bytes
\(^3\)\(2^2\) is the unique size.
If any of the Or-gates evaluated zero, a chunk with size $2^l$ blocks is available.

Figure 6.2: Or-Gate CBT, to indicate the existence of large enough free chunk, Reproduction by permission of IEEE [10]

gates generates zero output, the allocator will then determine that there exists large enough available chunk to satisfy the request.

Step2: The allocator needs to reveal the beginning address of the available chunk.

To indicate the starting address of the free chunk, whose availability was verified in the previous step, Chang - Gehringer introduced two extra bit-vectors, Propagate and Address Vectors (P-Vector and A-Vector) [10, 11]. P-bit, any bit of the P-Vector, is the result of and-gate CBT, and A-bit, in any node of the tree, is the P-bit of its left child. Figure 6.3 illustrates the use of P and A bits to find the address of the free chunk. If the A-bit of a given node is zero, it means that the p-bit of its left child is zero. Therefore, the A-bit of the left child should be selected by the corresponding multiplexor. The outputs of the multiplexors form the address of the first available chunk of requested size. Note that in the previous step (Step 1), the existence of such free chunk in the heap memory was verified.
Figure 6.3: And-Gate CBT, A mechanism which reveals the starting address of available chunk, Reproduction by permission of IEEE [10]

Step 3: The allocator should change the bit-vector bits that correspond to the found free chunk; this step is called Bit-Flipper [10].

The inputs to this phase of the allocator are the address and size of the free chunk found in two previous steps. The action of the allocator is to flip the values of the searched chunk’s bits of the bit-vector to 1. For this step, we also exploit the CBT explained in prior phases accompanied with two new signals, flip and route. If the flip signal of any node ‘N’ is set to one, the signal will be propagated through the CBT from this node, and it will result in flipping all the bits at the leaves of the subtree whose root is the node ‘N’. The route signal, however, has lesser priority and if any node N’s route signal is asserted, it will indicate that some of the bits at the leaves of the subtree with root N should be flipped to one. The assurance of which bits to flip is provided by the address and size information of the found free chunk. Figure 6.4 depicts each node of the CBT used in this phase, and Table 6.1 shows
the truth table for each node’s outputs. Finally, Figure 6.5 illustrates a simple example of address and size correlation with \textit{flip} and \textit{route} signals. In this example, the root’s \textit{flip} signal is 0 and its \textit{route} is one, which means that the bit-vector should be partially flipped. At this point, since the corresponding address bit is 0, the movement is towards the left subtree. The associated size bit of root in this example is one, which indicates that the \textit{flip} signal should be asserted, which in its turn makes sure that all the leaves of the left subtree will be flipped.

De-allocation process is very similar to allocation’s last step. Note that de-allocation requests deliver the address of the object to be freed. The allocator, after receiving the request, needs to determine the size of the object. For this purpose, we suggest the simple method of recording the size of the chunk in its header at a time of allocation [41]. This information can be read from the header during de-allocation procedure. Finally, the allocator uses \textit{size}, \textit{address}, \textit{flip}, and \textit{route} signals and flipping methods to flip the corresponding portion of the bit-vector for the freed chunk.

![Diagram](image)

Figure 6.4: Composed and decomposed flip and route signals with collaboration of address and size information, Reproduction by permission of IEEE [10]
\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\textbf{Inputs} & & & \textbf{Outputs} & & & \\
\hline
\text{flip} & \text{route} & \text{size} & \text{address} & \text{control} & \text{control} & \text{flip} & \text{route} & \text{flip} & \text{route} \\
\hline
1 & X & X & X & 1 & X & 1 & X & 0 & 0 \\
0 & 0 & X & X & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & X & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & X & X \\
\hline
\end{tabular}
\caption{Truth table for input and output flip and route signals of each node of CBT, with collaboration of address and size signals, Reproduction by permission of IEEE [10]}
\end{table}

Figure 6.5: An example of flipping the first four bits of a bit-vector using \textit{flip}, \textit{route}, \textit{address}, and \textit{size} signals, Reproduction by permission of IEEE [10]
6.3 Simulation Framework and Execution Performance of IMEM

The design of memory management system presented in the last section is composed of combinational circuits. The performance bottleneck of such design is when the allocator needs to examine the bit-vectors. Nevertheless, the heap memory can be partitioned into pages of 4 KBytes with the bit-vectors for each page. Finally, the allocator needs to keep the bit-vector for each page in designated registers to elevate the speed of allocation process. De-allocation, however, can be fully performed in parallel with the execution of the application. This is based on the assumption that, for each de-allocation request, the allocator has enough time to address the request before any new allocation request arrives.

Recent studies in evaluating the performance of Buddy System allocators, implemented in hardware, have shown that each allocation takes at most 10 memory cycles [18, 19]. Based on today’s technology, each memory cycle is about 5 CPU cycles. For example, typical Pentium IV has 2 GHz clock speed and the IMEM can be implemented in the heart of 400 MHz synchronous DRAM. Therefore, the worst case allocation speed is about 50 CPU cycles.

In our study, we have used SimpleScalar tool sets [8] to compare the performance of IMEM with conventional architecture. We have simulated the two systems with the following configurations:

Conv-Conf This system resembles the conventional architecture, in which we have simulated a superscalar as the main CPU. In Conv-Conf, both allocator’s functions and application’s code are running on the main CPU. Table 6.2 shows the system parameters for Conv-Conf scenario.

IMEM-Conf For IMEM System, we have excluded the execution cycles of allocation and de-allocation portion of the main CPU. For a fair comparison, however, we have considered the communication overhead for every allocation or de-allocation request to be the same as a system call. Other than communication overhead, for every de-allocation
request IMEM-Conf execution overhead is zero cycles, since de-allocation function in IMEM can be run in parallel with the application code executed by the main CPU. For each allocation request, in this configuration, we have considered almost 50 cycle overhead, since each allocation takes almost 10 memory cycles (recall that each memory cycle is about 5 CPU cycles).

<table>
<thead>
<tr>
<th>Issue Width</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUUs</td>
<td>16</td>
</tr>
<tr>
<td>LSQ</td>
<td>16</td>
</tr>
<tr>
<td>Int/FP/LS</td>
<td>4/4/2</td>
</tr>
<tr>
<td>I-Cache/D-Cache (L1)</td>
<td>32KB/2-Way/32B block</td>
</tr>
<tr>
<td>Unified L2-Cache</td>
<td>512KB/4-Way/64B block</td>
</tr>
<tr>
<td>I/D TLB</td>
<td>64 entries/Fully Associative/4KB pages</td>
</tr>
<tr>
<td>TLB miss latency</td>
<td>70 cycles</td>
</tr>
<tr>
<td>Memory Latency</td>
<td>100 cycles (first access)</td>
</tr>
</tbody>
</table>

Table 6.2: Simulated Processor’s Parameters

Table 6.3 shows the simulation results for Conv-Conf and IMEM-Conf using three benchmarks with small input sets. The description of the benchmarks can be found in chapter 5. Figure 6.6 depicts the percentage of execution performance speedup achieved by IMEM-Conf with respect to Conv-Conf. Although the overall speedup is a bit less than 4%, IMEM reveals over 10% performance improvement for espresso benchmark. This is because espresso allocates and de-allocates dynamic objects very actively. Note that since the SimpleScalar simulator runs quite slow, we were not able to execute these benchmarks with large input sets. Hence, the genuine benefit of Intelligent Memory Management System is not yet accurately exposed.

For Conv-Conf, we have used the allocator provided by the SimpleScalar tool set [8].
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Input Set</th>
<th>Conv-Conf Execution time (cycle)</th>
<th>IMEM-Conf Execution time (cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfrac</td>
<td>26 digit integer</td>
<td>90209199</td>
<td>89780879</td>
</tr>
<tr>
<td>espresso</td>
<td>Z5xp1.espresso</td>
<td>17738534</td>
<td>15902298</td>
</tr>
<tr>
<td>gzip</td>
<td>test.in</td>
<td>2421528916</td>
<td>2421528916</td>
</tr>
</tbody>
</table>

Table 6.3: Simulation results for Conv-Conf and IMEM

Figure 6.6: Percentage of Performance Improvement of IMEM-Conf as compared with Conv-Conf

The SimpleScalar tool set employs the `gnu` allocator written by Mike Haertel [79]. Although the fastest allocator that can be used in Conv-Conf is a buddy system allocator, but `gnu` allocator is also known for its speed. Having considered that mostly all applications written in C and compiled by `gnu` C compiler are linked with the `gnu` allocator, we believe that we have used the best allocator in Conf-Conv for fair comparison with IMEM-Conf.
In this chapter, we have presented the existing hardware implementation of buddy system allocator, Bit-Map Buddy System [10, 11]. Buddy system allocators are well known for their speed [76]; therefore, their hardware implementations are also considered the fastest among other allocators [18, 19]. Any logic circuit is composed of datapath and control system, and the speed of the circuit depends on the simplicity of its datapath and control system. The control unit of the Bit-Map Buddy System allocator is fully combinational; hence, it is very fast. The datapath of the Bit-Map Buddy System consists of its bit-vectors. Although each page of heap memory\textsuperscript{4} has at least two bit-vectors, but they can be cached for better speed.

In this study, we have also shown the execution performance benefit of IMEM-Conf, which consists of a CPU for executing the application code and a simple logic on chip with DRAM for executing the memory management functions. We have compared IMEM-Conf design with Conv-Conf which resembles the conventional architecture, an out-of-order issue CPU for executing both application code and memory management functions. Based on Bit-Map Buddy System design and on the top of communication overhead\textsuperscript{5}, we have added 50 CPU cycles for each allocation to actual CPU speed. De-allocation execution time, however, has been deemed negligible in our design. This is because de-allocation requests can be fully parallelized with application’s code.

For verifying the execution performance benefit of IMEM-Conf, we have used SimpleScalar tool set [8] and a small set of benchmarks. Out-Of-Order issue simulator, the most time consuming yet cycle by cycle accurate execution driven simulator of SimpleScalar tool set, obliges the researchers to use the smallest input data sets, and thus our data sets are also small. Running into difficulties of executing the benchmarks using Out-Of-Order

\textsuperscript{4}In this study, each memory page is considered 4 KBytes.
\textsuperscript{5}In MEM-Conf, for each allocation and de-allocation request, we have included the actual function call overhead as communication overhead.
issue simulator, we have succeeded to conduct our experiments only with three benchmarks. The results of our experiments, however, as shown in this chapter are very promising. We have shown that IMEM-Conf outperforms Conv-Conf by almost 4%, in average.

What can be done next? There is plenty of space for improvement in this work. One is to design accurate cycle by cycle execution driven IMEM simulator. It is only then that we can compare the execution performance of IMEM with different allocator implementations. There is also a great demand for improving the time that allocator spends allocating objects. In this study, we have reported 50 cycles for each allocation based upon other studies. We should be very well able to enhance the allocation execution time to about 10 cycles. Moreover, if the allocation requests scheduled early enough, the allocation execution on IMEM can be overlapped with the application execution on the main CPU. Then we can claim that allocation execution time is only the overhead for the communication between the main CPU and IMEM, similar to de-allocation.
CHAPTER 7
CONCLUSIONS

As the performance gap between processors and memory units continues to grow, memory accesses continue to limit performance on modern processors. While memory hierarchy and cache memories can alleviate the performance gap to some extent, cache performance is often adversely affected by service functions such as dynamic memory allocations and de-allocations. Modern applications rely heavily on linked lists and object-oriented programming. This requires sophisticated dynamic memory management, including allocation, de-allocation, garbage collection, data perfetching, Jump Pointers, and object relocation (Memory Forwarding). Using a single CPU (with its cache) for executing both service related functions and application code leads to poor cache performance. Sophisticated service functions need to traverse user data objects - and this requires the objects to reside in cache even when the application is not accessing them.

Motivation of the Work: The motivation of our work is three-fold:

The need for more efficient memory management algorithms has grown because the programming paradigm has changed. Object Oriented and linked data structured applications invoke memory management functions very frequently. For example, Java programs allocate and de-allocate more than 100,000s of objects dynamically in their execution period. It has been shown that about 30% of Object Oriented and linked data structured applications’ execution time is spent on allocating and de-allocating the objects. This observation has led us to design new memory management functions that are efficient for Object Oriented applications.

High execution performance and storage utilization are the main objectives of all mem-
ory management algorithms, but it is very difficult to reach both goals. Moreover, these two goals have been studied separately in the literature. We believe that they are very related; in fact, allocators with poor storage utilization perform poorly in terms of execution performance.

Frequently used service functions such as allocations and de-allocations when mixed with the execution of application code become the major cause of cache pollution. The cache pollution can be removed by separating the execution of these functions from application code and migrating them to a different processor. Service functions are also very data intensive, the feature that made them suitable for execution in a processor integrated with DRAM in a single chip. This is yet another observation that motivated Intelligent Memory Management research and directed us towards Intelligent Memory Devices (viz., eRAM, Active Pages, and IRAM).

**Dissertation’s Contributions:** This dissertation explores the space of possible solutions into the existing trends that tolerate memory latency. In this work, we show that data-intensive and frequently used service functions such as memory allocation and de-allocation entangle with application’s working set and become a major cause of cache misses. In this dissertation we have proposed new Memory Management techniques - ABT, SBT, and hybrid allocator - which are aimed to reach high execution performance, high storage utilization, and low memory overhead (over-allocated memory). The latter objective, low memory overhead, is the outcome of the observation that internal fragmentation counters locality behavior of applications.

We have also presented a novel technique that transfers the allocation and de-allocation functions’ computations entirely to a separate processor residing on chip with DRAM, called Intelligent Memory Management. This technique eliminates the **execution overhead** of the service functions from application’s code. The empirical results in this dissertation show that more than half of the cache misses caused by allocation and
de-allocation service functions are eliminated when using Intelligent Memory Management.

**Future Work:** Internal fragmentation is a great threat to cache performance of applications. Although we have minimized internal fragmentation in our hybrid allocator, it still keeps the size information with each object (live object or free chunk). If an allocator keeps two lists, one for maintaining live objects and one for free chunks, it will become possible to eliminate internal fragmentation entirely. Having live object and free chunk lists, an allocator can automatically perform de-allocations on behalf of the application. This is referred to as Garbage Collection in the literature. With this scenario, Intelligent Memory Management will gain more attention if we migrate allocation and automatic garbage collection functions to the processor in the memory. On the other hand, the hardware constraint of Process In Memory devices needs more study. In the future, we would like to expand our Intelligent Memory Management for performing garbage collection and object relocations (for better cache performance). We will also design the hardware of Intelligent Memory Management System.
BIBLIOGRAPHY


